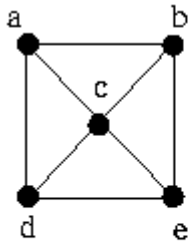
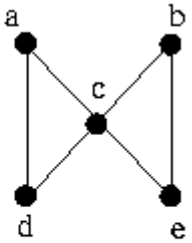
Problemas Hamiltonianos

Definição: Um **circuito hamiltoniano** em um grafo conexo *G* é definido como um caminho elementar, fechado passando em cada vértice de *G* exatamente uma vez. Um grafo que admite um circuito hamiltoniano é um **grafo hamiltoniano**.

Evidentemente nem todo grafo é hamiltoniano. A figura abaixo ilustra dois grafos, o primeiro é hamiltoniano (a) e o segundo não (b).



(a)



(b)

Portanto, um circuito hamiltoniano de um grafo de *n* vértices consiste de exatamente *n* arestas. Um circuito hamiltoniano equivale a uma permutação dos vértices, assim, o número máximo de caminhos hamiltonianos de um grafo com n vértices é igual a *n*!.

Os problemas desta classe são, de modo geral, de dificuldade maior que os problemas eulerianos. Há um grande número de resultados úteis, ao lado de muitas questões em aberto; por outro lado os algoritmos para descobrir um circuito hamiltoniano são em geral, muito mais trabalhosos. Enquanto que um grafo euleriano pode ser verificado em tempo linear, não se conhece algoritmo polinomial para verificar se um grafo é hamiltoniano, exceto para casos particulares. Por exemplo, não é complicado verificar que um grafo completo possui um circuito hamiltoniano em tempo $O(n^2)$.

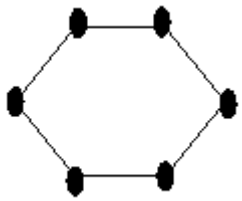
Propriedades para grafos hamiltonianos

Existem muitos resultados acerca de grafos hamiltonianos, porém, há poucos teoremas relativamente úteis. A seguir são apresentados dois teoremas.

Teorema de Ore. Uma condição suficiente (mas não necessária) para que um grafo G seja hamiltoniano é que a soma dos graus de cada par de vértices não adjacentes seja no mínimo n.

Teorema de Dirac: Uma condição suficiente (mas não necessária) para que um grafo simples G possua um ciclo hamiltoniano, é que o grau de cada vértice em G seja pelo menos igual a n/2, onde n é o número de vértices em G.

Observe que estes teoremas não são suficientes para determinar se um grafo é hamiltoniano. Observe o grafo abaixo que é um grafo hamiltoniano que não obedece as condições dos teoremas acima. O grau de cada vértice é 2, que é menos que 6/2=3 e, além disso, a soma dos graus de qualquer par de vértices não adjacentes é sempre 4, que é menor que *n*=6 (vértices).



A seguir é apresentado um algoritmo que determina os circuitos hamiltonianos.

Método Exato: Método "Composição Latina"

Trata-se de um método algébrico para enumeração de caminhos hamiltonianos. Este método gera todos os caminhos simples por multiplicação de matrizes.

Considere a matriz $B(n \times n)$ da seguinte forma:

$$B_{ij} = \begin{cases} v_j, \text{se existe a aresta } (v_i, v_j) \\ 0, \text{caso contrário} \end{cases}$$

Passos do Algoritmo

P1. Construa a matriz B e a matriz de adjacência A do grafo dado.

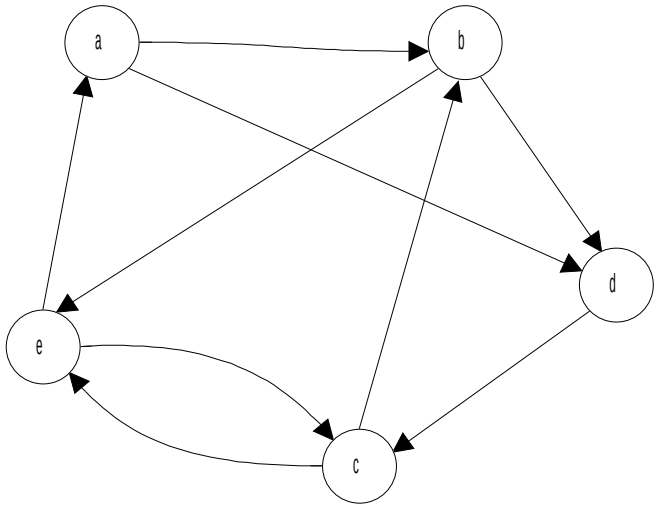
P2. Faça $P_1 \leftarrow A$;

P3. Para $i = 1,2,....,n\text{-}2$ faça
$P_{i+1} \leftarrow B \times P_i$

Obs:

1) Os elementos da matriz P são cadeias de caracteres (vértices) e não números. A operação *multiplicação* significa a concatenação do caracteres e a *soma* representa a divisão de duas ou mais cadeias de caracteres.
2) Cada elemento da matriz P_i representa um caminho hamiltoniano de comprimento $i + 1$ entre os vértices **s** e **t.**
3) Para todo P_{i+1} a diagonal é zero, assim como todo caminho de **s** até **t** contendo **s**.

Exemplo:
Determinar as caminhos hamiltonianos do grafo abaixo:



Solução pelo método da composição latina.

Passo 1: $A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix}$ $B = \begin{bmatrix} 0 & b & 0 & d & 0 \\ 0 & 0 & 0 & d & e \\ 0 & b & 0 & 0 & e \\ 0 & 0 & c & 0 & 0 \\ a & 0 & c & 0 & 0 \end{bmatrix}$

Passo 2: $P_1 = A$

Passo 3: $P_2 = B \text{x} P_1$; $P_3 = B \text{x} P_2$; $P_4 = B \text{x} P_3$

Eliminam-se nas matrizes P_i os termos sublinhados, pois são caminhos de ***s*** até ***t*** que contem ***s***. Faz-se também a diagonal igual a zero.

$$P_2 = \begin{bmatrix} 0 & 0 & d & b & b \\ e & 0 & d+e & 0 & 0 \\ e & 0 & \underline{e} & b & b \\ 0 & c & 0 & 0 & c \\ 0 & a+c & 0 & a & \underline{c} \end{bmatrix} = \begin{bmatrix} 0 & 0 & d & b & b \\ e & 0 & d+e & 0 & 0 \\ e & 0 & 0 & b & b \\ 0 & c & 0 & 0 & c \\ 0 & a+c & 0 & a & 0 \end{bmatrix}$$

$$P_3 = \begin{bmatrix} \underline{be} & dc & bd+be & 0 & dc \\ 0 & \underline{bc+ea+ec} & 0 & ea & dc \\ be & ea+\underline{ec} & \underline{bd+be} & ea & 0 \\ ce & 0 & 0 & \underline{cb} & cb \\ ce & 0 & ad & ab+cb & \underline{ab+cb} \end{bmatrix}$$

$$P_4 = \begin{bmatrix} \underline{dce} & 0 & 0 & \underline{bea} & bdc + dcb \\ dce & 0 & ead & \underline{eab} + \underline{ecb} & \underline{dcb} \\ 0 & 0 & \underline{ead} & bea + eab + \underline{ecb} & \underline{bdc} + eab + \underline{ecb} \\ cbe & cea & 0 & \underline{cea} & 0 \\ \underline{cbe} & adc + \underline{cea} & abd + \underline{abe} & \underline{cea} & \underline{abc} \end{bmatrix}$$

Os caminhos hamiltonianos são: abdce, adcbe, bdcea, baedc, cbead, ceabd, dcbea, dceab, eadcb, eabdc.

Se as arestas são valoradas, então pode-se determinar o caminho de menor custo.

Os ciclos disjuntos contendo todos os vértices são: **abdcea , adcbea**

Obs: A maioria dos algoritmos exatos são normalmente da classe "*Branch – and- Bound*". Trata-se de buscas em árvores, caracterizadas pelo particionamento do conjunto de soluções por um critério dado. Em problemas práticos estes algoritmos são inviáveis, pois, o tempo de computação cresce rapidamente com a dimensão do problema.

Problema do Caixeiro Viajante (PCV)

(*Traveling Salesman Problem*)

É um problema de grande importância prática que consiste na determinação da rota de menor custo para uma pessoa que parta de uma cidade e deva visitar diversas outras, passando pelo menos uma vez em cada cidade e retornando ao ponto de partida.

O custo pode ser medido em termos de dinheiro, tempo, distância, etc, e as opções existentes para as diferentes etapas de viagens correspondem as arestas de um grafo valorado.

Se o grafo não for muito grande, e especialmente, se não tiver muitos arcos, será possível enumerar os circuitos hamiltonianos e depois achar o valor de cada um deles; mas no caso geral, isso é impossível na prática.

Algoritmos Heurísticos para o PCV

Os métodos exatos para resolver o PCV não são computacionalmente eficientes, o que tem levado os pesquisadores a utilizarem algorítimos heurísticos, que geram soluções satisfatórias, com consumo de tempo computacional muitas vezes menor do que um algoritmo exato.

Uma heurística é um processo que procura boas soluções de um problema a um custo computacional razoável, porém, sem ser capaz de garantir a otimalidade ou até, em muitos casos, de estabelecer quão perto uma dada solução viável está da solução ótima. Portanto, não existe prova de corretude desses algoritmos heurísticos.

Os algoritmos heurísticos encontrados na literatura podem ser divididos nas seguintes classes:

- Heurística de Construção;
- Heurística de Melhoramento;
- Heurística mista.

<u>Heurística de Construção</u>

As heurísticas de construção mais conhecidas são:

- Vizinho mais próximo;
- Inserção mais próxima;
- Inserção mais distante;
- Inserção mais barata;
- Algoritmos das Economias (Clark-Wright).

<u>Heurística de Melhoramento</u>

As heurísticas de melhoramento (ou de melhoria iterativa) mais conhecidas são:

- K-opt ou K-melhoramento; (em sala de aula)
- *Simulated Annealing;*
- Algoritmos Genéticos;
- Busca Tabu;
- etc.

Heurística de Melhoramento K-Opt

Trata-se de uma estratégia de troca de arestas que podem ser usadas para melhorar uma solução obtida por algum algoritmo de construção. A heurística K-Opt elimina K arestas não adjacentes do

roteiro do PCV e reconectam esses k caminhos em ordem diferentes através de outras k arestas. Entretanto, as experiências computacionais com K=2 e K=3 têm apresentado os melhores resultados.

A Heurística 2-OPT

A heurística 2-opt é uma das técnicas mais conhecida para melhorar uma solução para o problema do caixeiro viajante. Inicia-se com um ciclo *H*; retira-se 2 arestas de *H*, assim produzindo 2 grafos desconectados. Os dois grafos são reconectados de tal maneira que produza outro ciclo *H'*. Assim *H* e *H'* diferem em exatamente 2 arestas. Calcula-se o custo *w*(*H'*) do ciclo *H'*. Se *w*(*H'*)< *w*(*H*), o ciclo *H* é substituído por *H'* e repete-se o processo; caso contrário, um outro par de arestas é trocado. As mudanças continuam até que nenhum melhoramento poder se feito trocando 2 arestas. A solução final que não pode ser melhorada mudando apenas 2 arestas é chamada de uma solução 2-opt (Figura 1).

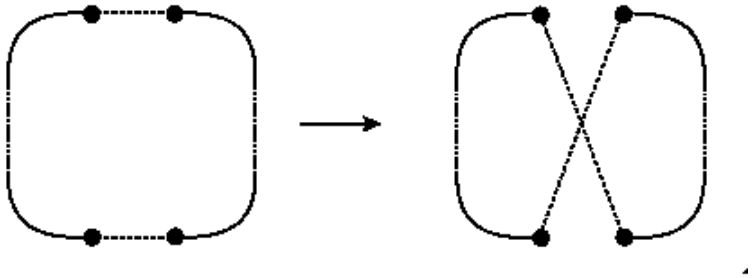


Figura 1. Ilustração da heurística 2-Opt

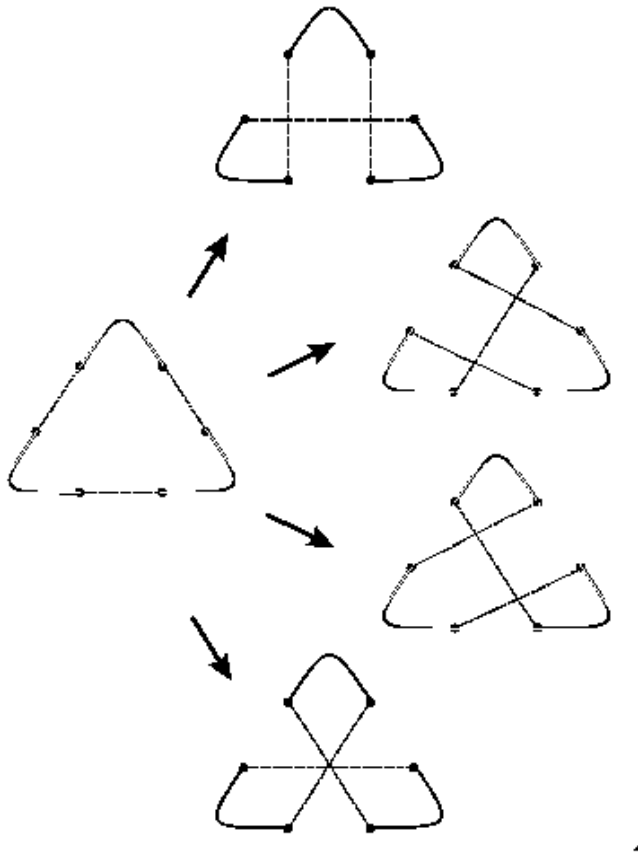


Figura 2. Ilustração da heurística 3-Opt.

De maneira análoga a heurística 2-Opt pode se criar a heurística 3-Opt. Neste caso, a mudança de 3 arestas podem ser combinadas em 4 formas diferente, conforme a Figura 2.

Formalização da Heurística 2-Opt

Dado um ciclo inicial contendo o seguinte conjunto de arestas $H=\{x_1, x_2, ..., x_n\}$ na ordem $x_1, x_2, ..., x_n$. Define-se $w(H)$ o custo associado ao ciclo H. Seja $X=\{x_i,x_j\}$ um conjunto de 2 arestas de H é apagado e substituído por outro conjunto de arestas $Y=\{y_p,y_q\}$, obtendo-se um novo ciclo $H'= (H - X) \cup Y$. Se $w(H')<w(H)$ então H é substituído por H'.

Observe que:

1) as duas arestas x_i, x_j de X não podem ser adjacentes;
2) uma vez que X tenha sido escolhido, o conjunto Y é determinado.

Assim, é possível gerar $\frac{n(n-3)}{2}$ ciclos H' a partir de um dado ciclo H.

Denota-se δ como sendo o melhoramento

$\delta = w(H) - w(H') = (w(x_i) + w(x_j)) - (w(y_p) + w(y_q))$.

O algoritmo examina todos os ciclos H' a fim de obter um ciclo com o máximo valor de δ. O valor máximo é denotado por δ_{max}.

A seguir é descrito o algoritmo deste método para alcançar uma solução 2-opt para o problema do caixeiro viajante através de sucessivas trocas de 2 arestas (Syslo *et al*, 1983)

Procedure 2-OPT

```
begin
    < seja H= (x1, x2, ..., xn) o ciclo corrente >;
    repeat
        δmax :=0;
        for i:=1 to (n-2) do
            for j:=(i+1) to n  < ou (n-1) quando i=1 > do
                if (w(xi) + w(xj) - w(yp) - w(yq))> δmax then
                    begin
                        δmax:= (w(xi) + w(xj) - w(yp) - w(yq));
                        < guardar i e j >;
                    end;
        if  δmax > 0 then H:= H- {xi , xj} ∪{yp ,yq};
    until δmax=0;
end;
```

Exercício 1. Faça a análise de complexidade para cada algoritmo heurístico construtivo apresentado.

Exercício 2. Faça a análise de complexidade para os algoritmos heurísticos melhorativo 2-Opt e 3-Opt.