




# Metaheurísticas e Aplicações

Celso Carneiro Ribeiro

Departamento de  
Ciência da Computação (UFF)

Outubro 2007

# Conteúdo

- Origens
- Teoria da complexidade
- Solução de problemas NP-difíceis
- Motivação
- Métodos construtivos
- Representação de soluções 
- Vizinhanças
- Busca local
- Metaheurísticas
  - *Simulated annealing*
  - Busca tabu 
  - GRASP
  - VNS/VND
  - Algoritmos genéticos 
  - Colônias de formigas
- Aplicações
- Otimização multicritério
- Paralelização

# Origens

- Algoritmos aproximados para o problema do caixeiro viajante: métodos construtivos, busca local 2-opt, heurística de Lin-Kernighan (1973)
- Técnicas de inteligência artificial para problemas de busca em grafos:
  - Demonstração automática de teoremas, trajeto de robôs, problemas de otimização vistos como busca em grafos
  - Algoritmo A\* (“versão” IA de B&B): Nilsson (1971)
  - Aplicações pioneiras no SE brasileiro (anos 70): modelos TANIA (expansão da transmissão) e VENUS (planejamento da expansão)



# Teoria da complexidade

- Problemas de decisão: “existe uma determinada estrutura satisfazendo certa propriedade?”  
*Resultado*: “sim” ou “não”
- Problemas de otimização: “dentre todas as estruturas satisfazendo determinada propriedade, obter aquela que otimiza certa função de custo.”  
*Resultado*: uma solução viável ótima
- Exemplo: problema do caixeiro viajante  
*Entrada*:  $n$  cidades e distâncias  $c_{ij}$   
*Problema de decisão*: “dado um inteiro  $L$ , existe um ciclo hamiltoniano de comprimento menor ou igual a  $L$ ?”  
*Problema de otimização*: “obter um ciclo hamiltoniano de comprimento mínimo.”

# Teoria da complexidade

- Exemplo: Problema da mochila  
*Entrada*:  $n$  itens, peso máximo  $b$ , lucros  $c_j$  e pesos  $a_j$  associados a cada item  $j=1, \dots, n$   
*Problema de decisão*: “dado um inteiro  $L$ , existe um subconjunto  $S \subseteq \{1, \dots, n\}$  tal que  $\sum_{j \in S} a_j \leq b$  e  $\sum_{j \in S} c_j \geq L$ ?”  
*Problema de otimização*: “obter um conjunto  $S^*$  maximizando  $\sum_{j \in S} c_j$  entre todos os conjuntos  $S \subseteq \{1, \dots, n\}$  tais que  $\sum_{j \in S} a_j \leq b$ .”
- Estudo da teoria da complexidade baseado nos problemas de decisão
- Um algoritmo (determinístico) para um problema de decisão  $A$  é polinomial se sua complexidade de pior caso é limitada por um polinômio no tamanho  $L(I_A)$  de sua entrada  $I_A$

# Teoria da complexidade

- Classe P: problemas de decisão para os quais são conhecidos algoritmos polinomiais
- Exemplos de problemas da classe **P**:  
ordenação  
caminhos mais curtos em um grafo  
árvore geradora de peso mínimo  
fluxo máximo  
programação linear
- Algoritmos não-determinísticos: são construídos usando as primitivas  
**Choice** - “chuta” uma solução (*oráculo*)  
**Check** - verifica (em tempo polinomial) se uma proposta de solução (certificado) leva ou não a uma resposta “sim”  
**Success** - algoritmo responde “sim” após aplicar **Check**  
**Fail** - algoritmo não responde “sim”

# Teoria da complexidade

- Resultado: se **Choice** “chuta” uma solução levando a uma resposta “sim” (e ele tem capacidade para isto), então o algoritmo executa em tempo polinomial.
- Algoritmos não-determinísticos: pode-se também dizer que algoritmos fazem uso de uma instrução especial de desvio “duplo”  
**GO TO Label\_A,Label\_B**  
que funciona como se criasse duas cópias do fluxo de execução, como se estivesse em um ambiente de paralelismo ilimitado, cada cópia correspondendo a um diferente “chute” possível

# Teoria da complexidade

- Exemplo: problema da mochila  
**for**  $j = 1$  **to**  $n$  **by** 1  
    **go to** A,B  
**A:**  $x_j \leftarrow 0$   
    **go to** C  
**B:**  $x_j \leftarrow 1$   
**C:** **continue**  
**if**  $a.x \leq b$  **and**  $c.x \geq L$  **then** “sim”
- Um algoritmo não-determinístico é polinomial se o primeiro ramo que leva a uma resposta “sim” termina em tempo polinomial.
- Classe NP: problemas de decisão para os quais são conhecidos algoritmos não-determinísticos polinomiais (problemas de decisão para os quais qualquer certificado pode ser verificado em tempo polinomial para uma resposta “sim”)



# Teoria da complexidade

- Resultado:  $\mathbf{P} \subseteq \mathbf{NP}$
- Em aberto:  $\mathbf{P} = \mathbf{NP}$  ou  $\mathbf{P} \subset \mathbf{NP}$  ?
- Transformação polinomial: um problema de decisão A se transforma polinomialmente em outro problema de decisão B se, dada qualquer entrada  $I_A$  para o problema A, pode-se construir uma entrada  $I_B$  para o problema B em tempo polinomial no tamanho  $L(I_A)$  da entrada  $I_A$ , tal que  $I_A$  é uma instância “sim” para A se e somente se  $I_B$  é uma instância “sim” para B.
- Problemas **NP**-completos: um problema de decisão  $A \in \mathbf{NP}$  é **NP**-completo se todos os outros problemas de **NP** se transformam polinomialmente em A

# Teoria da complexidade

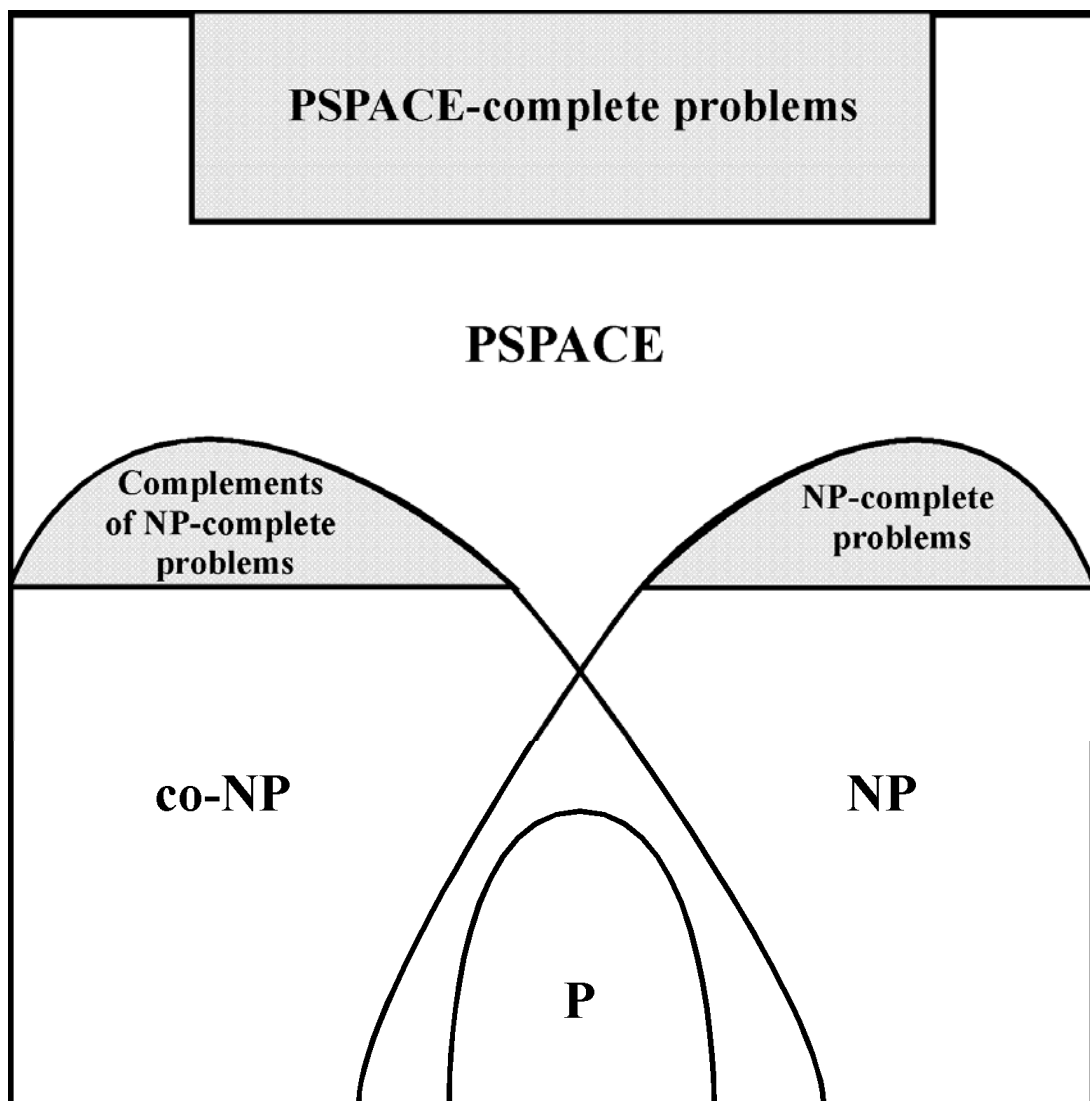
- Se existir um algoritmo (determinístico) polinomial para a resolução de algum problema **NP**-completo, então todos os problemas da classe **NP** também poderão ser resolvidos em tempo polinomial.
- Exemplos de problemas **NP**-completos: caixeiro viajante, mochila, coloração de grafos, programação inteira, problema de Steiner em grafos
- Não existem ou ainda não foram encontrados algoritmos polinomiais para os problemas **NP**-completos.
- Classe **Co-NP**: problemas de decisão cujo complemento pertence à classe **NP**  
*Exemplo*: “dado um grafo  $G$ , este grafo não possui um ciclo hamiltoniano?”

# Teoria da complexidade

- Problemas NP-árduos ou NP-difíceis: problemas de otimização cujo problema de decisão associado é NP-completo
- Classe PSPACE: problemas de decisão que podem ser resolvidos utilizando uma quantidade polinomial de memória  
 $P \subseteq PSPACE$   
 $NP \subseteq PSPACE$

# Teoria da complexidade

- Visão simplificada do “mundo” dos problemas de decisão:



# Solução de problemas NP-difíceis

- Algoritmos exatos não-polinomiais:
  - programação dinâmica*
  - branch-and-bound*
  - backtracking*
- Algoritmos pseudo-polinomiais:
  - polinomiais no tamanho da instância e no valor do maior dado da entrada
  - problema da mochila
- Processamento paralelo: aceleração na prática, sem redução da complexidade
- Casos especiais polinomiais
  - coloração de grafos de intervalos
- Algoritmos aproximativos: encontram uma solução viável com custo a distância máxima garantida do ótimo
  - bin packing*

# Solução de problemas NP-difíceis

- Algoritmos probabilísticos:
  - convergência em valor esperado
  - convergência em probabilidade
- Heurísticas: qualquer método aproximado projetado com base nas propriedades estruturais ou nas características das soluções dos problemas, com complexidade reduzida em relação à dos algoritmos exatos e fornecendo, em geral, soluções viáveis de boa qualidade (sem garantia de qualidade)
  - métodos construtivos
  - busca local
  - metaheurísticas
- Avanços no estudo e desenvolvimento de heurísticas:
  - resolver problemas maiores
  - resolver problemas em tempos menores
  - obter melhores soluções



# Métodos construtivos

- Problema de otimização combinatória:

Dado um conjunto finito

$$E = \{1, 2, \dots, n\}$$

e uma função de custo

$$c: 2^E \rightarrow \mathbb{R}$$

encontrar

$$S^* \in F \text{ tal que } c(S^*) \leq c(S) \quad \forall S \in F$$

onde  $F \in 2^E$  é o conjunto de soluções viáveis do problema

- Conjunto discreto de soluções com um número finito de elementos
- Construção de uma solução:  
selecionar seqüencialmente elementos de  $E$ , eventualmente descartando alguns já selecionados, de tal forma que ao final se obtenha uma solução viável, i.e. pertencente a  $F$ .

# Métodos construtivos

- Problema do caixeiro viajante (PCV)

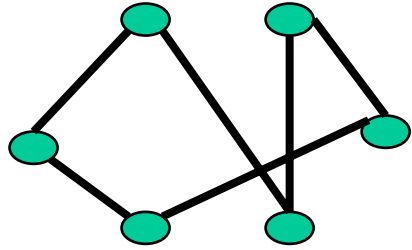
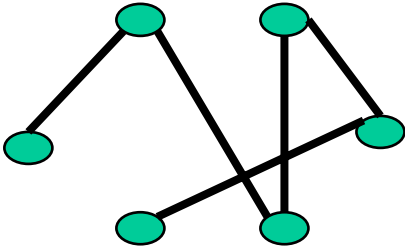
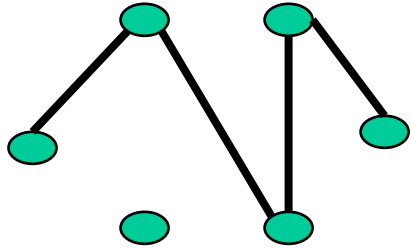
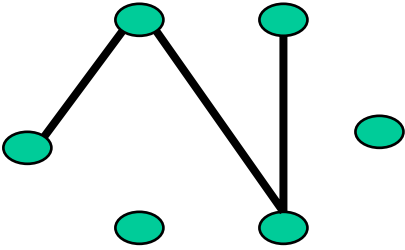
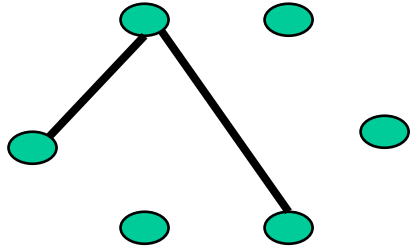
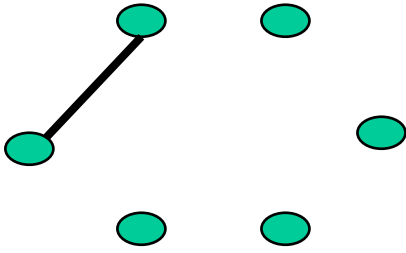
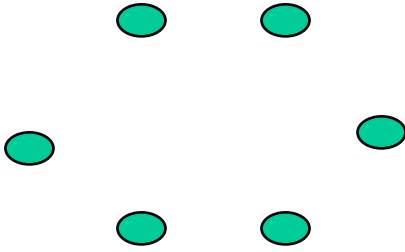
E: conjunto de arestas

F: subconjuntos de E que formam um ciclo hamiltoniano (CH)

$$c(S) = \sum_{e \in S} c_e$$

$c_e$ : custo da aresta e.





# Caixeiro viajante

- Grafo completo  $G=[N,E]$
- $n$  nós
- Distância entre cada par de nós  $(i,j)$ :  $c_{ij}$
- Exemplo com  $n=5$ :

$C =$

-	1	2	7	5
1	-	3	4	3
2	3	-	5	2
7	4	5	-	3
5	3	2	3	-

# Caixeiro viajante

- Algoritmo do vizinho mais próximo para construção de uma solução para o PCV

## Passo 0:

$H \leftarrow \{1\}, L \leftarrow 0, N \leftarrow N - \{1\}, i \leftarrow 1$

## Passo 1:

Se  $N = \emptyset$ , fazer  $L \leftarrow L + c_{i1}$  e terminar.

## Passo 2:

Obter  $j \in N: c_{ij} = \min_{k \in N} \{c_{ik}\}$ .

## Passo 3:

$H \leftarrow H \cup \{j\}$

$L \leftarrow L + c_{ij}$

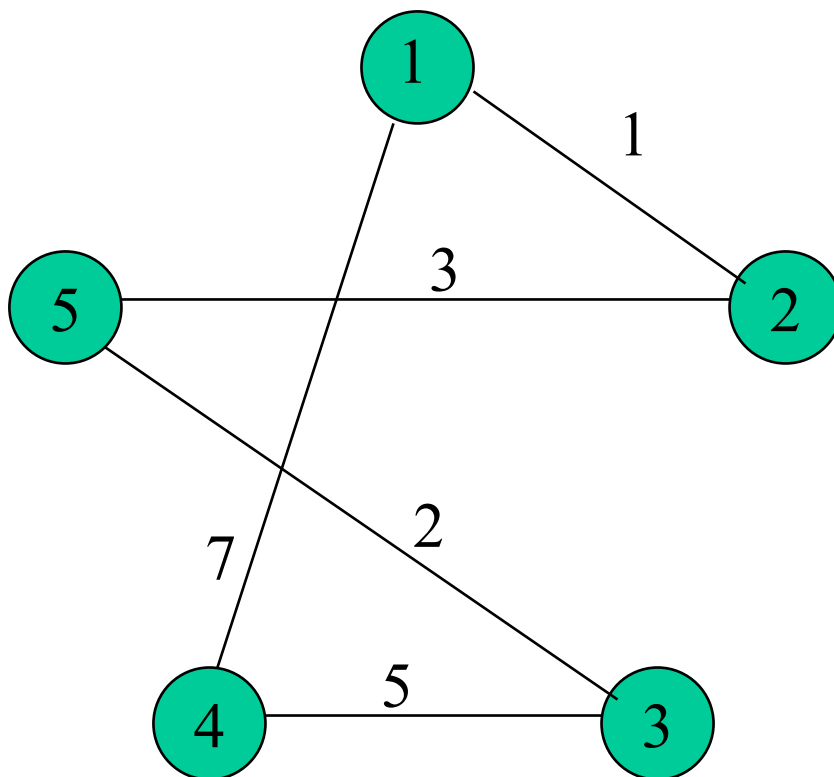
$N \leftarrow N - \{j\}$

$i \leftarrow j$

Retornar ao passo 1.

- Complexidade:  $O(n^2)$
- Influência do nó inicial
- Aplicar a partir de cada nó:  $O(n^3)$

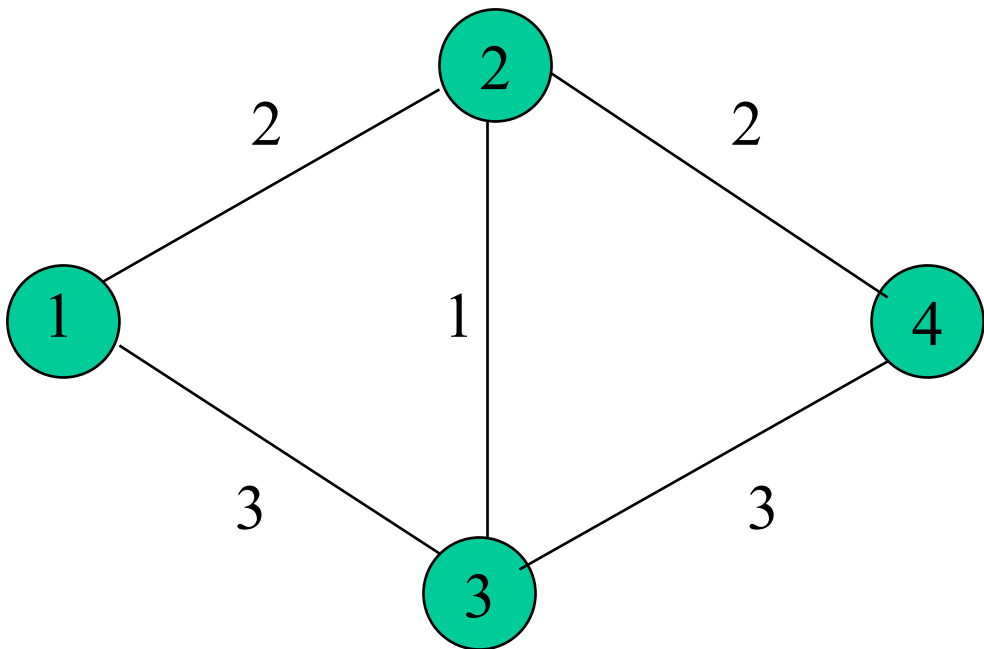
# Caixeiro viajante



$$L=18$$

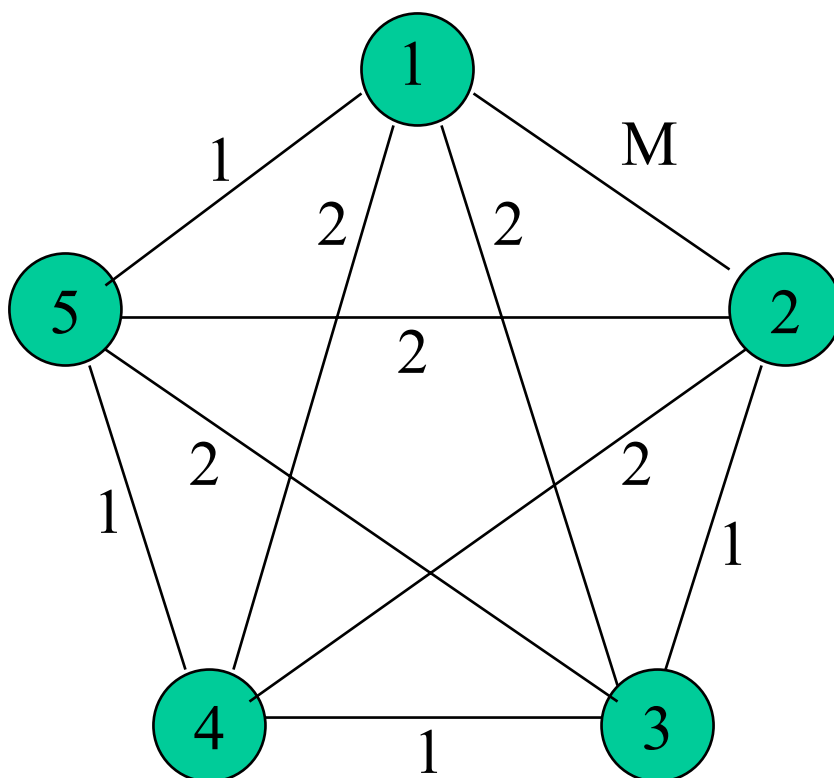
# Caixeiro viajante

- Possibilidade de não encontrar uma solução viável:



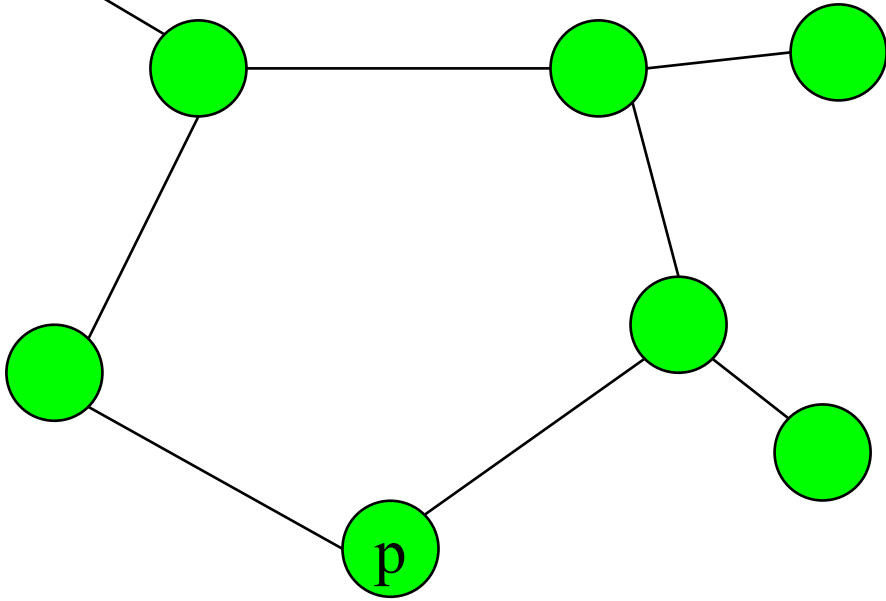
# Caixeiro viajante

- Possibilidade de obter soluções arbitrariamente ruins:

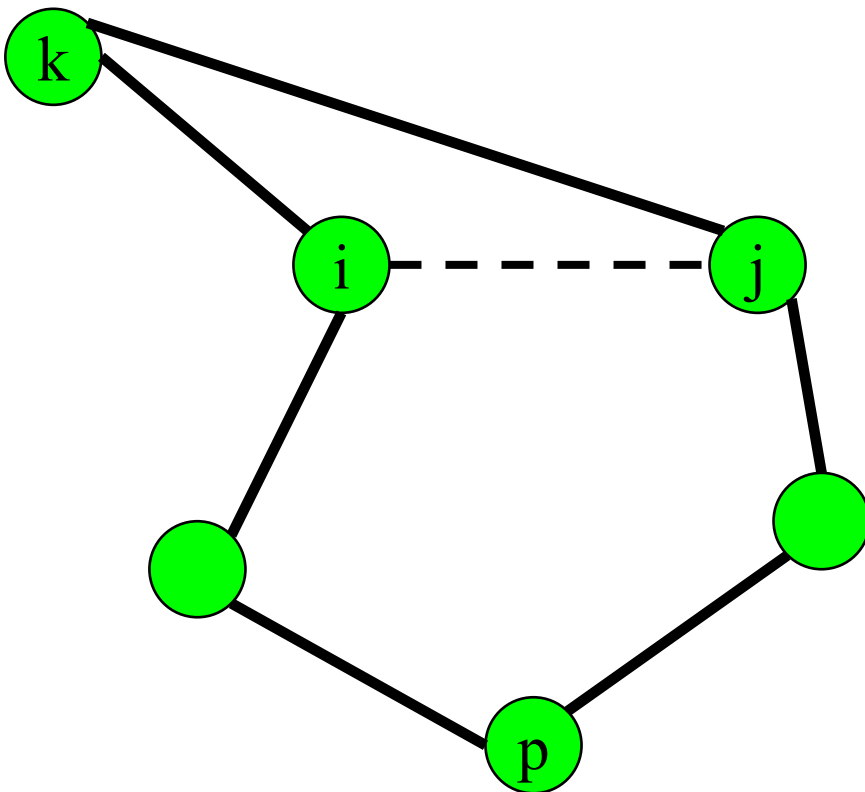


k

### Escolha do vértice



### Escolha das arestas



# Caixeiro viajante

- Algoritmo de inserção mais próxima para construção de uma solução para o PCV

## Passo 0:

Inicializar o ciclo com apenas um vértice.

## Passo 1:

Encontrar o vértice  $k$  fora do ciclo corrente cuja aresta de menor comprimento que o liga a este ciclo é mínima.

## Passo 2:

Encontrar o par de arestas  $(i,k)$  e  $(k,j)$  que ligam o vértice  $k$  ao ciclo minimizando

$$c_{ik} + c_{kj} - c_{ij}$$

Inserir as arestas  $(i,k)$  e  $(k,j)$  e retirar a aresta  $(i,j)$ .

## Passo 3:

Retornar ao passo 1.

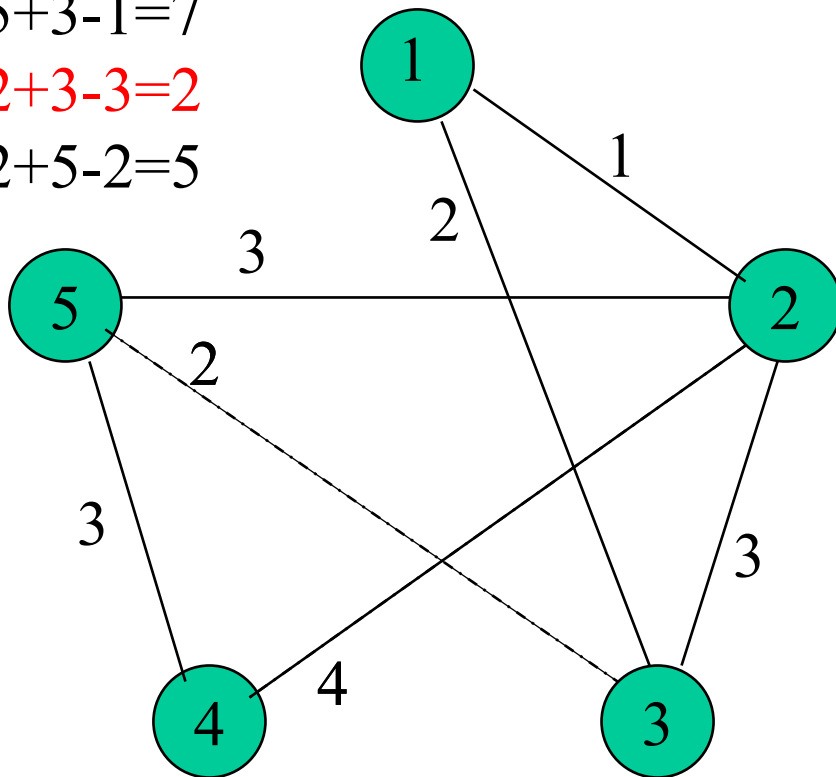


# Caixeiro viajante

$$\Delta_{12}(5) = 5 + 3 - 1 = 7$$

$$\Delta_{23}(5) = 2 + 3 - 3 = 2$$

$$\Delta_{13}(5) = 2 + 5 - 2 = 5$$



$$\Delta_{12}(4) = 7 + 4 - 1 = 10$$

$$\Delta_{13}(4) = 7 + 5 - 2 = 10$$

$$\Delta_{35}(4) = 5 + 3 - 2 = 6$$

$$\Delta_{25}(4) = 4 + 3 - 3 = 4$$

$$L = 12$$

# Caixeiro viajante

- Algoritmo de inserção mais distante para construção de uma solução para o PCV

## Passo 0:

Inicializar o ciclo com apenas um vértice.

## Passo 1:

Encontrar o vértice  $k$  fora do ciclo corrente cuja aresta de menor comprimento que o liga a este ciclo é máxima.

## Passo 2:

Encontrar o par de arestas  $(i,k)$  e  $(k,j)$  que ligam o vértice  $k$  ao ciclo minimizando

$$c_{ik} + c_{kj} - c_{ij}$$

Inserir as arestas  $(i,k)$  e  $(k,j)$  e retirar a aresta  $(i,j)$ .

## Passo 3:

Retornar ao passo 1.

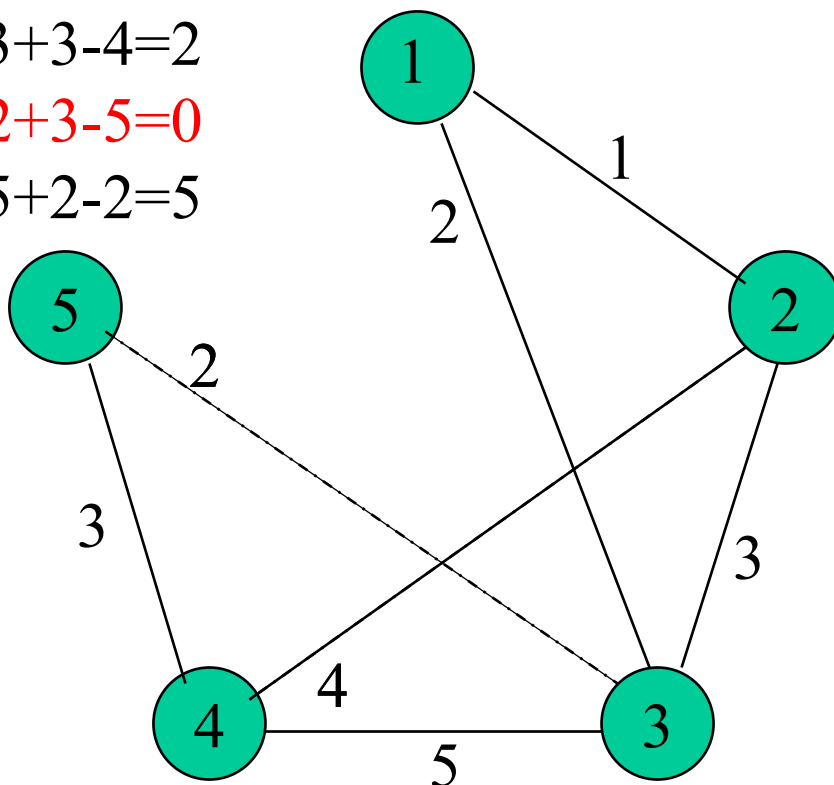
# Caixeiro viajante

$$\Delta_{12}(5) = 5 + 3 - 1 = 7$$

$$\Delta_{24}(5) = 3 + 3 - 4 = 2$$

$$\Delta_{34}(5) = 2 + 3 - 5 = 0$$

$$\Delta_{13}(5) = 5 + 2 - 2 = 5$$



$$\Delta_{12}(4) = 7 + 4 - 1 = 10$$

$$\Delta_{23}(4) = 4 + 5 - 3 = 6$$

$$\Delta_{13}(4) = 7 + 5 - 2 = 10$$

L=12

# Caixeiro viajante

- Algoritmo de inserção mais barata para construção de uma solução para o PCV

## Passo 0:

Inicializar o ciclo com apenas um vértice.

## Passo 1:

Encontrar o vértice  $k$  fora do ciclo corrente e o par de arestas  $(i,k)$  e  $(k,j)$  que ligam o vértice  $k$  ao ciclo minimizando

$$c_{ik} + c_{kj} - c_{ij}$$

## Passo 2:

Inserir as arestas  $(i,k)$  e  $(k,j)$  e retirar a aresta  $(i,j)$ .

## Passo 3:

Retornar ao passo 1.

# Caixeiro viajante

- Algoritmo de inserção pelo maior ângulo para construção de uma solução para o PCV

## Passo 0:

Inicializar o ciclo com a envoltória convexa dos nós do grafo.

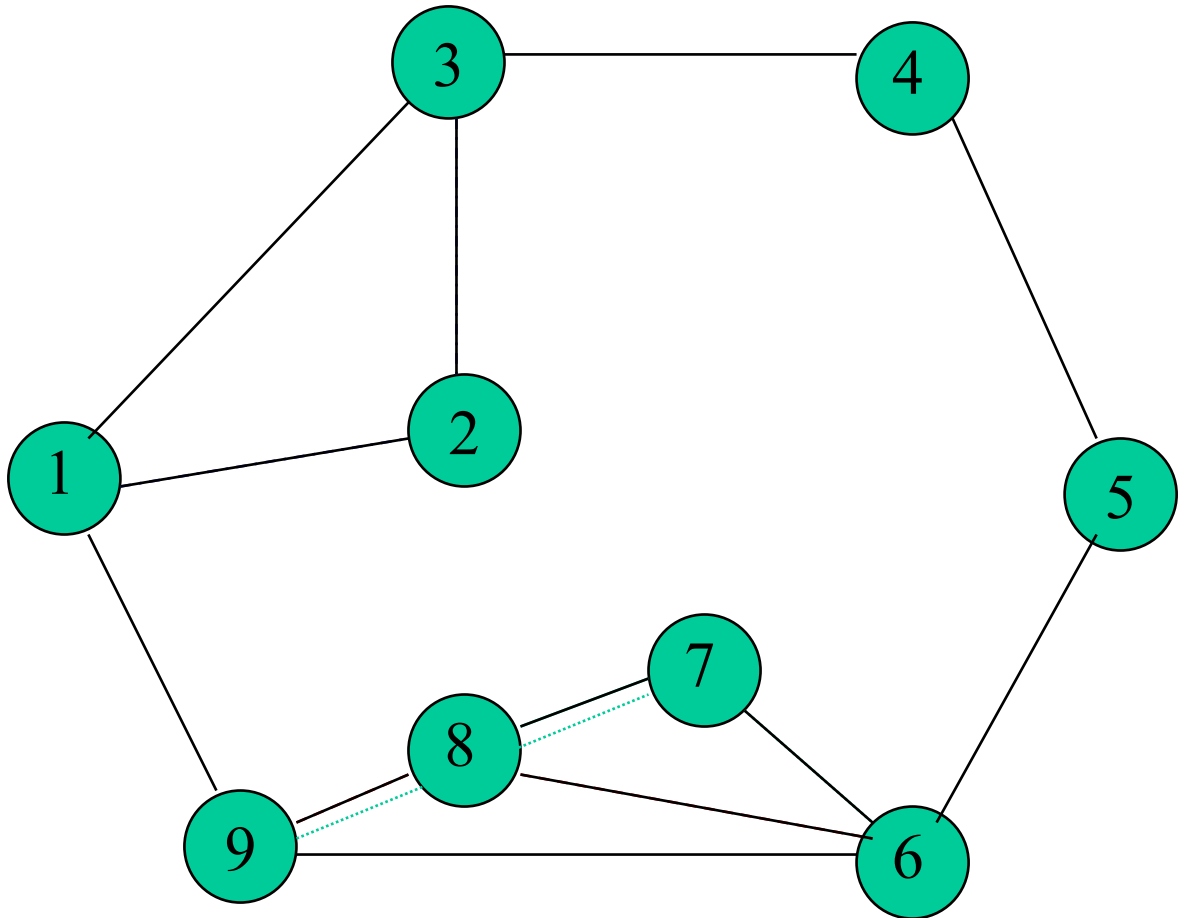
## Passo 1:

Inserir cada um dos nós não pertencentes à envoltória convexa aplicando uma **heurística de inserção**: encontrar o vértice  $k$  fora do ciclo corrente e o par de arestas  $(i,k)$  e  $(k,j)$  que ligam o vértice  $k$  ao ciclo tal que o ângulo formado pelas arestas  $(i,k)$  e  $(k,j)$  seja máximo.

## Passo 2:

Retornar ao passo 1.

# Caixeiro viajante



# Caixeiro viajante

- Algoritmo de economias para construção de uma solução para o PCV

## Passo 0:

Escolher um vértice base inicial (p.ex., nó 1).

## Passo 1:

Construir subciclos de comprimento 2 pelo vértice base e por cada um dos  $n-1$  nós restantes.

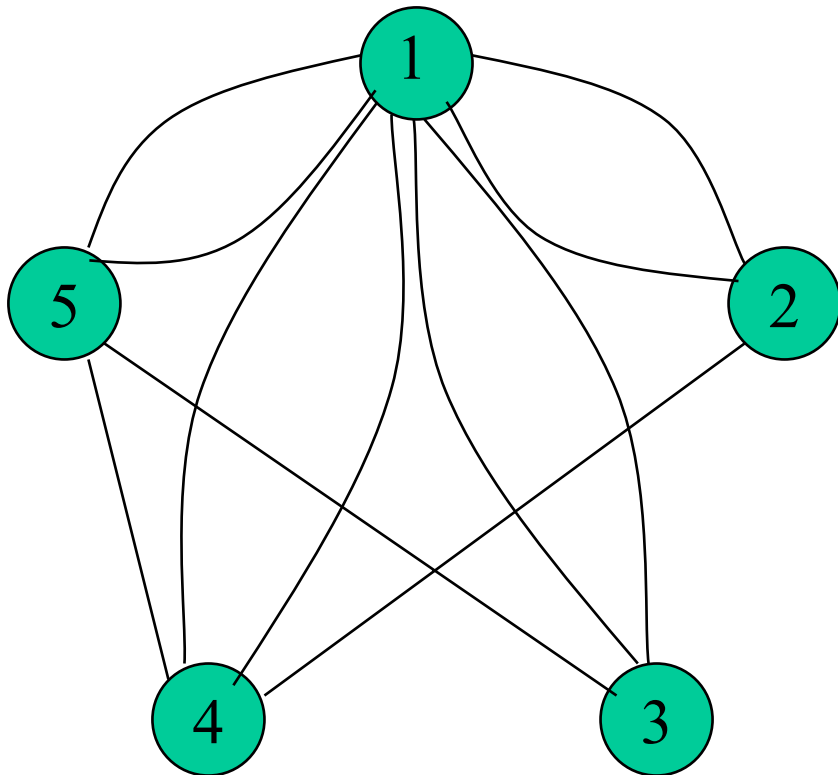
## Passo 2:

Calcular as economias  $s_{ij} = c_{1i} + c_{j1} - c_{ij}$  e ordená-las em ordem decrescente.

## Passo 3:

A cada iteração, maximizar a distância economizada sobre a solução anterior, combinando-se dois subciclos (um chegando e outro saindo do nó 1) e substituindo-os por uma nova aresta: percorrer a lista de economias e realizar as trocas que mantêm uma rota iniciando no vértice 1 e passando pelos demais nós, até obter um ciclo hamiltoniano.

# Caixeiro viajante



$$s_{45} = 9$$

$$s_{35} = 5$$

$$s_{34} = 4$$

$$s_{24} = 4$$

$$s_{25} = 3$$

$$s_{23} = 0$$

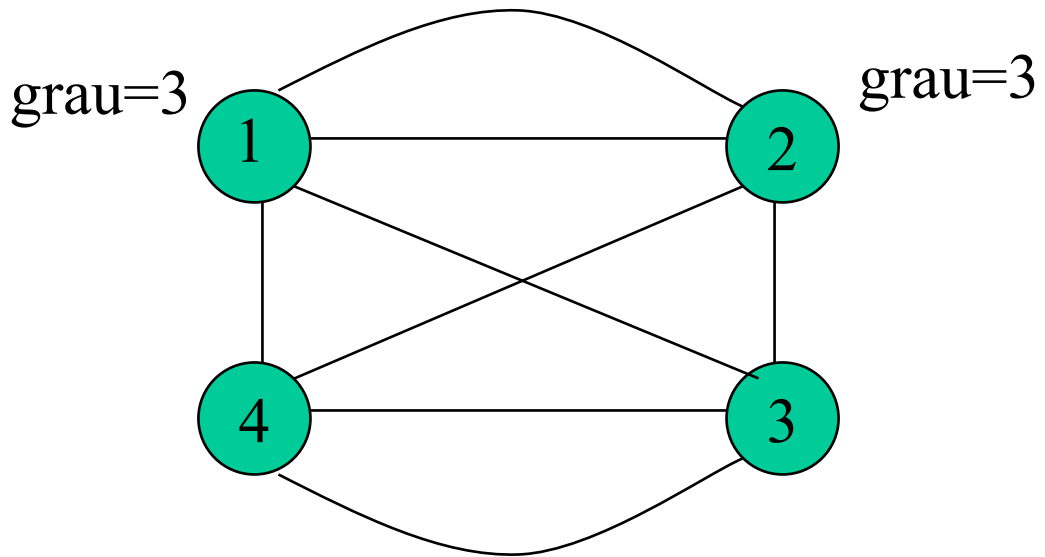
$$L=12$$



# Caixeiro viajante

- Heurísticas baseadas na árvore geradora de peso mínimo para construir uma solução para o PCV
- ✓ Árvore geradora de um grafo com  $n$  nós: grafo conexo sem ciclos com  $n$  nós e  $n-1$  arestas.
- ✓ Comprimento de um ciclo hamiltoniano de peso mínimo:  $H^*$
- ✓ Eliminando-se uma aresta de um ciclo hamiltoniano obtém-se uma árvore geradora:  
 $T^* < H^*$
- ✓ “Twice-around minimal spanning tree procedure”
- ✓ Circuito euleriano: começa e termina no mesmo nó e percorre cada aresta exatamente uma vez.
- ✓ Condição necessária e suficiente para existência de um circuito euleriano em um grafo: todos os nós têm grau par.
- ✓  $H \leq E = 2.T^*$  e  $T^* \leq H^* \rightarrow H \leq 2.H^*$
- ✓ Atalhos: desigualdade triangular válida
- ✓ Heurística de Christofides:  $H \leq 1,5 H^*$

# Caixeiro viajante



todos os nós tem grau 4

# Caixeiro viajante

## **Passo 0:**

Obter a árvore geradora de peso mínimo.

## **Passo 1:**

Duplicar os arcos da árvore geradora de peso mínimo.

## **Passo 2:**

Obter um circuito euleriano (usando busca em profundidade).

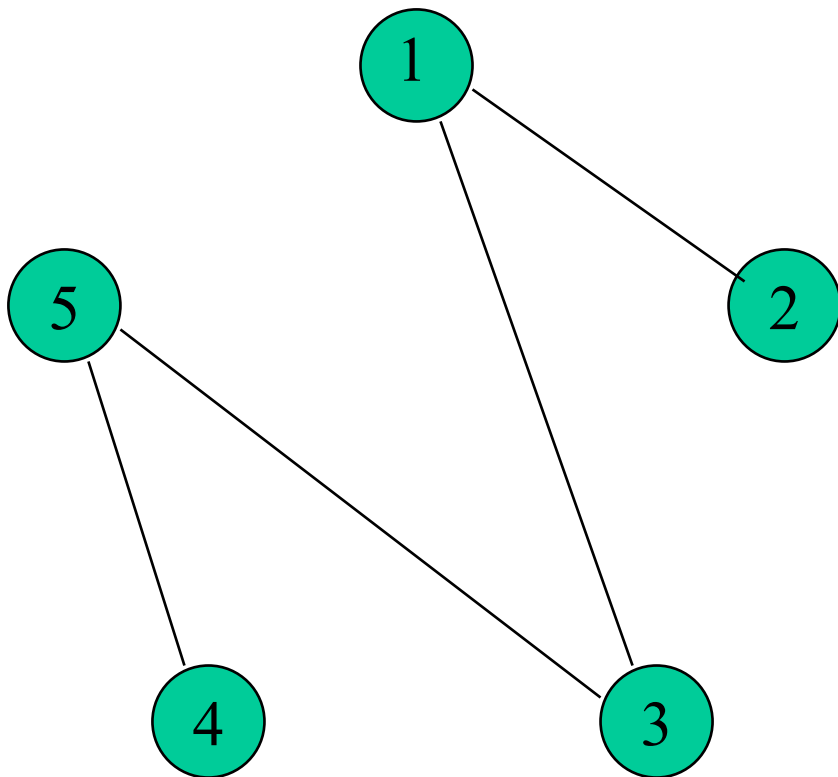
## **Passo 3:**

Escolher um nó inicial.

## **Passo 4:**

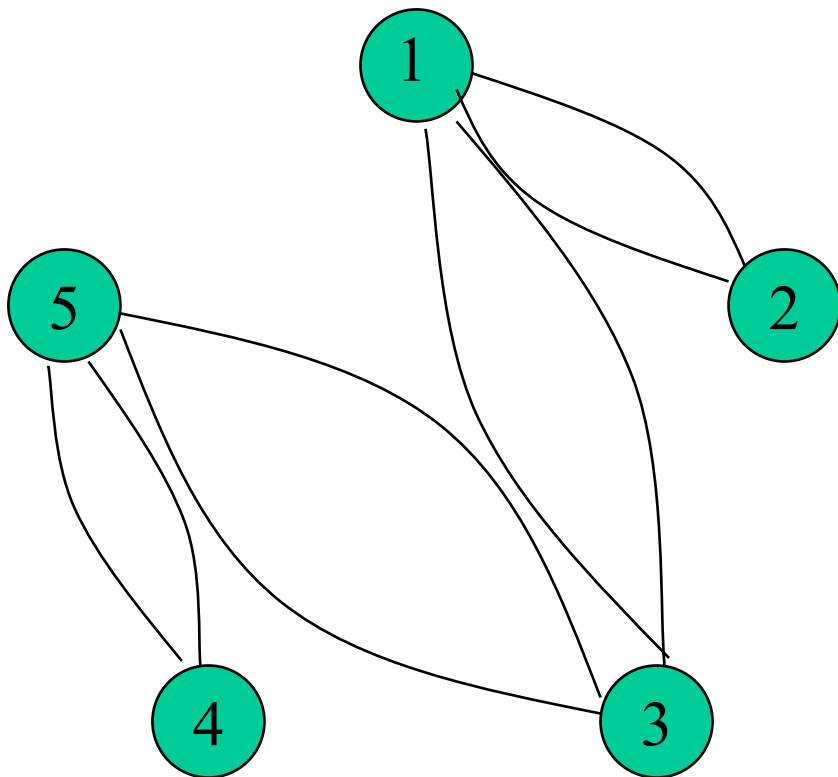
Montar um ciclo hamiltoniano sobre o circuito euleriano, usando atalhos quando necessário para não repetir nós.

# Caixeiro viajante



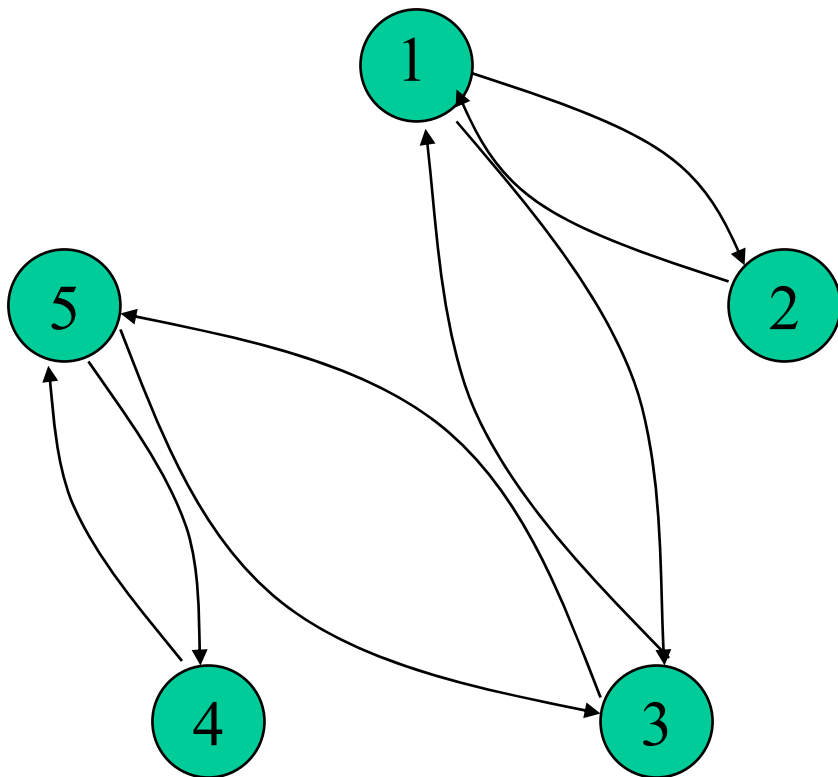
AGPM

# Caixeiro viajante



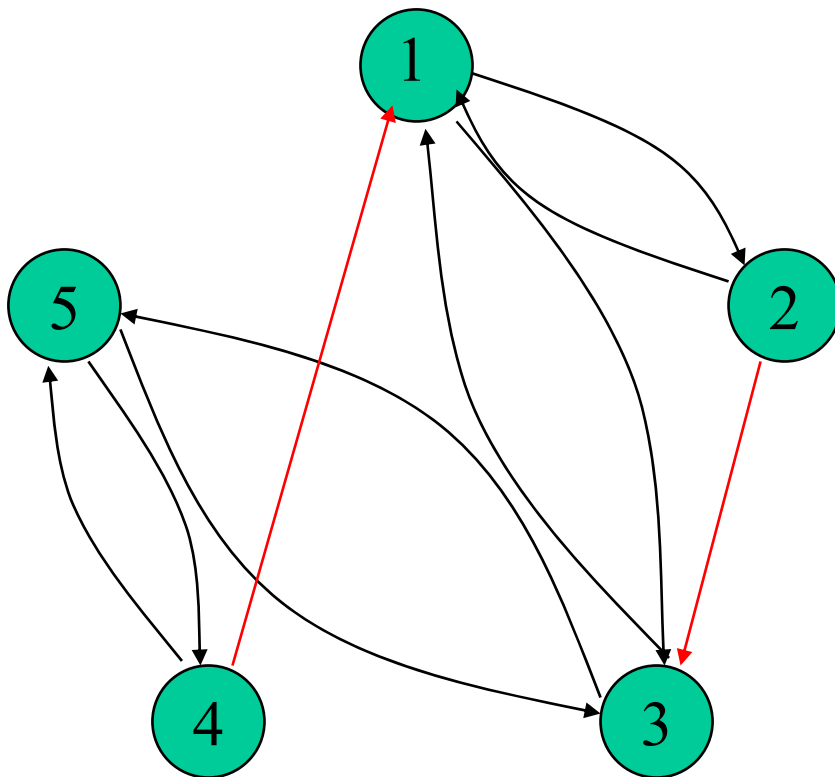
duplicação da AGPM

# Caixeiro viajante



circuito euleriano

# Caixeiro viajante



ciclo hamiltoniano

$L=16$



# Métodos construtivos

- Problema da mochila

E: conjunto de itens

F: subconjuntos de E que satisfazem à  
restrição  $\sum_{e \in S} a_e \leq b$

$$c(S) = \sum_{e \in S} c_e$$

$c_e$ : lucro do item e

$a_e$ : peso do item e

b: capacidade da mochila



$\frac{5}{3}$ 1	$\frac{4}{5}$ 2	$\frac{6}{7}$ 3	$\frac{3}{4}$ 4
$\frac{5}{2}$ 5	$\frac{8}{9}$ 6	$\frac{5}{5}$ 7	$\frac{9}{8}$ 8

$$\frac{c_e}{a_e}$$

i
---

Capacidade da mochila: 15

Item	Peso	Lucro
3	7	6
3, 5	9	11
3, 5, 7	14	16

# Métodos construtivos

- Algoritmos gulosos:  
a construção de uma solução gulosa consiste em selecionar seqüencialmente o elemento de E que minimiza o incremento no custo da solução parcial, eventualmente descartando alguns já selecionados, de tal forma que ao final se obtenha uma solução viável.
- O incremento no custo da solução parcial é chamado de função gulosa.
- Por escolher a cada passo considerando apenas a próxima decisão, chama-se também de algoritmo míope, pois enxerga somente o que está mais próximo.

# Métodos construtivos

- Problema da árvore geradora mínima:

Dado um grafo conexo  $G=(V,E)$  e pesos  $c_e$  associados às arestas  $e \in E$ , determinar uma árvore geradora  $T \subseteq E$  cujo peso  $c(T) = \sum_{e \in T} c_e$  seja mínimo.

E: conjunto de arestas

F: subconjuntos de E que formam árvores geradoras.

# Métodos construtivos

- Algoritmo guloso para construção de uma árvore de custo mínimo (Kruskal)

## Passo 0:

Ordenar as arestas de  $E$  de modo que

$$c_1 \leq c_2 \leq \dots \leq c_n$$

$$T \leftarrow \emptyset$$

## Passo 1:

**Para  $i$  de 1 a  $n$  faça**

**Se  $T \cup \{e_i\} \in F$**

**então  $T \leftarrow T \cup \{e_i\}$**

Observação:

$T$  é uma solução ótima

Função gulosa:  $c_e$  (peso da aresta)



# Métodos construtivos

- Algoritmo guloso para construção de uma solução para o problema da mochila

## **Passo 0:**

Ordenar os elementos de  $E$  de modo que

$$c_1/a_1 \geq c_2/a_2 \geq \dots \geq c_n/a_n$$

$$S \leftarrow \emptyset$$

## **Passo 1:**

**Para**  $i$  **de** 1 **a  $n$  **faça****

**Se**  $S \cup \{e_i\} \in F$

**então**  $S \leftarrow S \cup \{e_i\}$

## Observação:

$S$  não é necessariamente uma solução ótima

Função gulosa:  $c_e/a_e$  (“densidade” do elemento)



# Métodos construtivos

- Algoritmo guloso aleatorizado
  - Algoritmo guloso encontra sempre a mesma solução para um dado problema, exceto por eventuais empates
  - Aleatorização permite alcançar diversidade nas soluções encontradas
  - Criar uma lista de candidatos L e forçar uma escolha aleatória a cada iteração
- A qualidade da solução obtida depende da qualidade dos elementos na lista de candidatos L
- A diversidade das soluções encontradas depende da cardinalidade da lista L
- Casos extremos:
  - algoritmo guloso puro
  - solução gerada aleatoriamente



# Métodos construtivos

- Algoritmo guloso aleatorizado:  
(minimização)

## **Passo 0:**

Ordenar os elementos de  $E$  de modo que

$$c_1 \leq c_2 \leq \dots \leq c_n$$

$$S \leftarrow \emptyset$$

## **Passo 1:**

### **Para $i$ de 1 a $n$ faça**

Criar uma lista  $L \subseteq \{1, 2, \dots, n\} \setminus S$   
tal que  $S \cup \{e\} \in F, \forall e \in L$

Selecionar aleatoriamente um elemento  
 $e \in L$

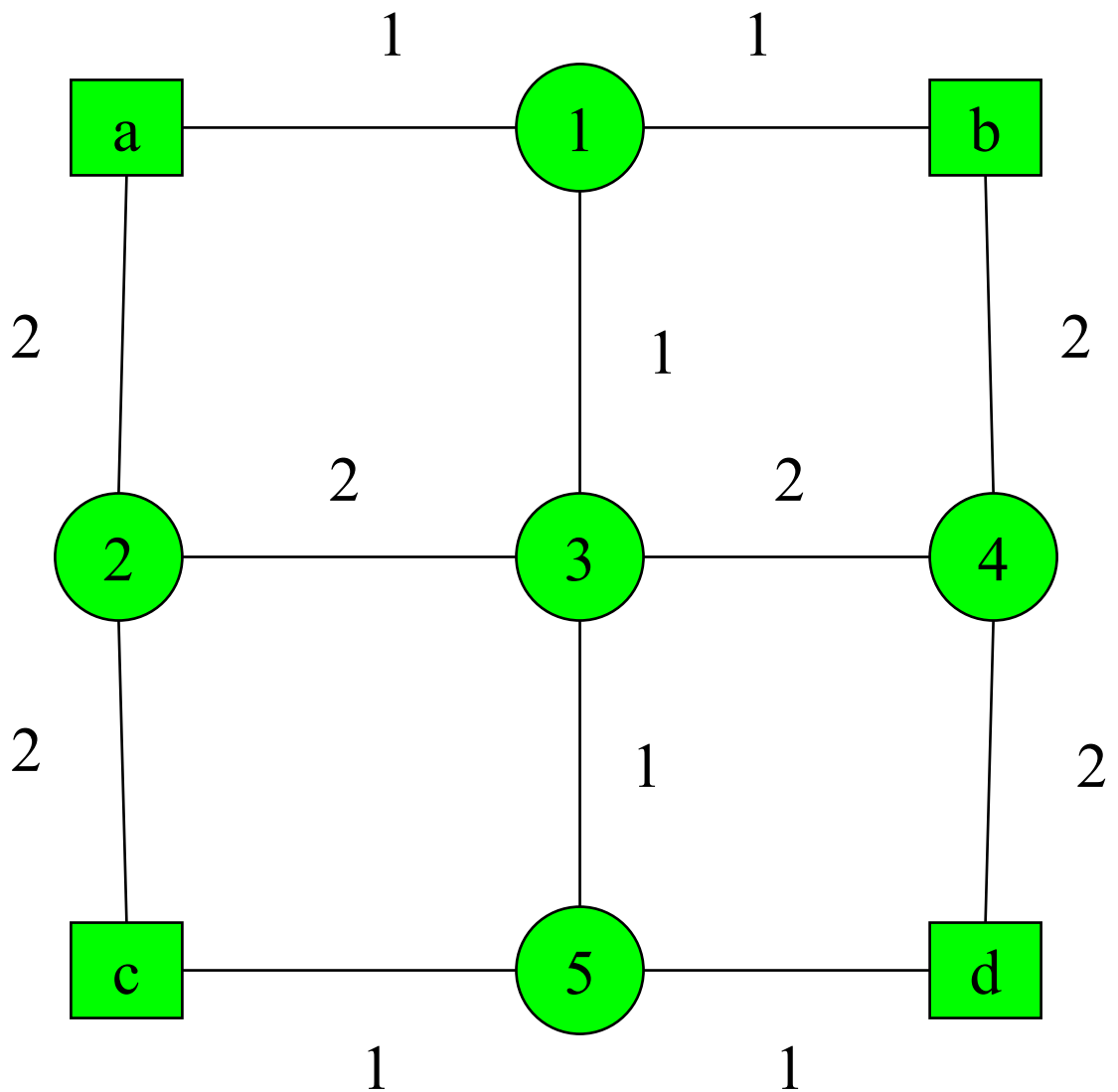
$$S \leftarrow S \cup \{e\}$$

# Problema de Steiner em grafos

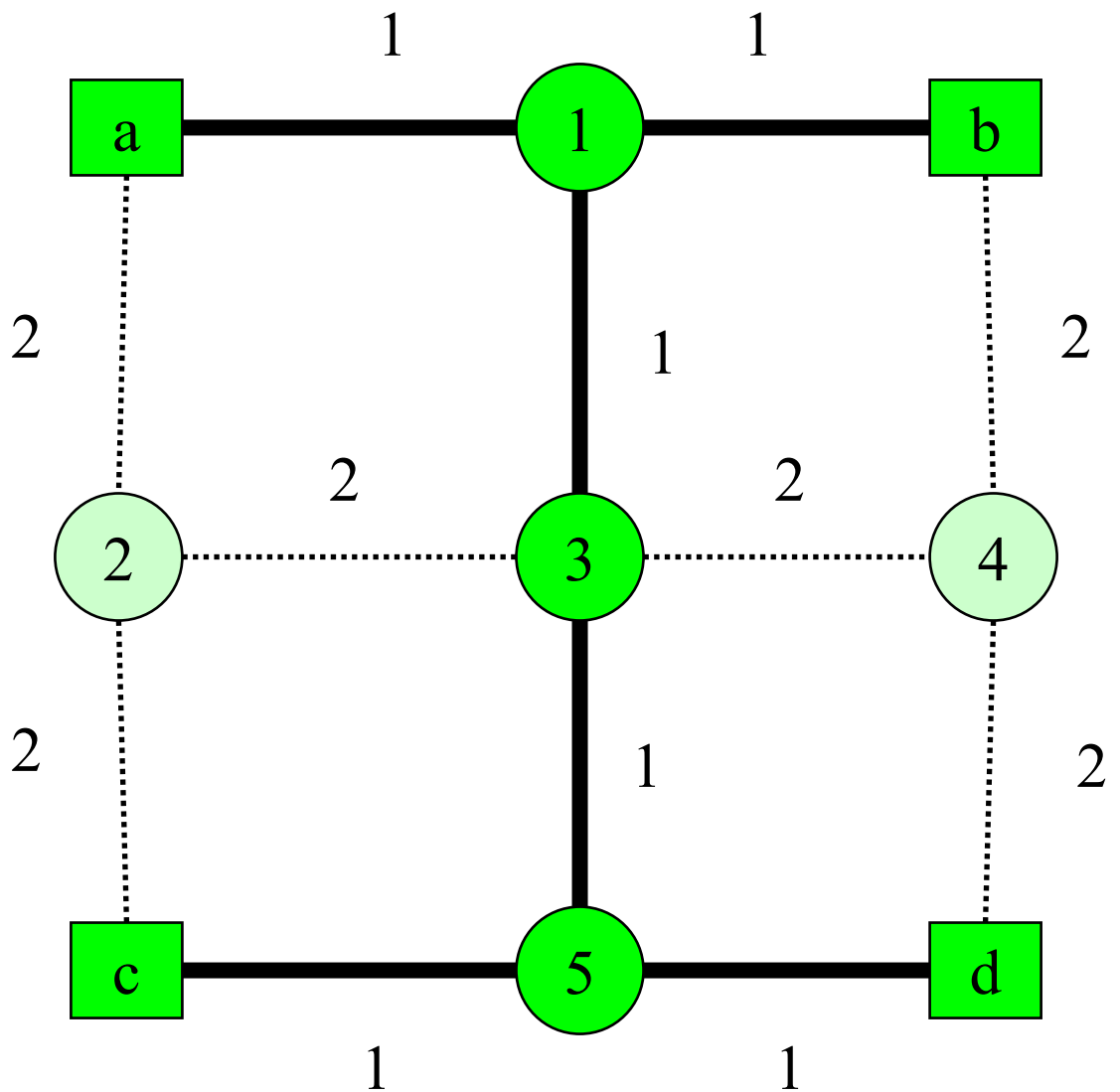
- Problema de Steiner em grafos:  
grafo não-orientado  $G=(V,E)$   
 $V$ : vértices  
 $E$ : arestas  
 $T$ : vértices terminais (obrigatórios)  
 $c_e$ : peso da aresta  $e \in E$
- Determinar uma árvore geradora dos vértices terminais com peso mínimo  
(caso particular: se  $T = V$ , problema da árvore geradora de peso mínimo)
- Vértices de Steiner: vértices opcionais que fazem parte da solução ótima
- Aplicações: projeto de redes de computadores, redes de telecomunicações, problema da filogenia em biologia



# Problema de Steiner em grafos



# Problema de Steiner em grafos



# Problema de Steiner em grafos

- Heurística de caminhos mais curtos

Iteração  $k$ :

$S_k = \{\text{terminais gerados pela árvore SPH}_k\}$

**Passo 0:**

Calcular o caminho mais curto de cada terminal para cada nó do grafo.

**Passo 1:**

Sejam  $s_0 \in T$ ,  $\text{SPH}_0 \leftarrow \{s_0\}$ ,  $S_0 \leftarrow \{s_0\}$ ,  $k \leftarrow 0$

**Passo 2:**

Se  $T = S_k$ , fim. Senão, fazer  $k \leftarrow k+1$ .

**Passo 3:**

Obter o terminal  $s_k$  mais próximo de  $\text{SPH}_{k-1}$  e o caminho correspondente  $C_k$ .

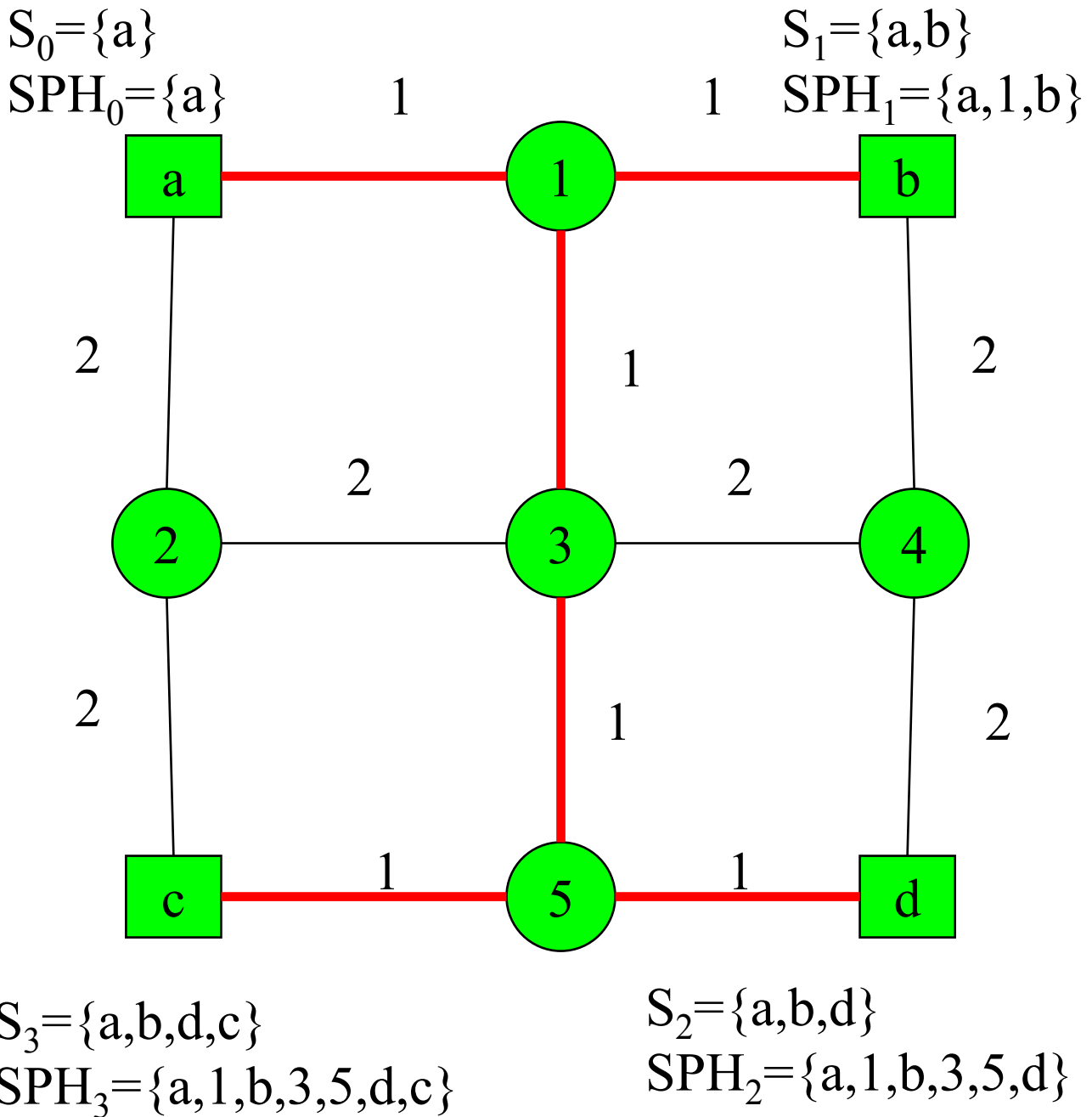
**Passo 4:**

Fazer  $S_k \leftarrow S_{k-1} \cup \{s_k\}$  e  $\text{SPH}_k \leftarrow \text{SPH}_{k-1} \cup C_k$ .

**Passo 5:**

Atualizar as distâncias e retornar ao passo 2.

# Problema de Steiner em grafos



# Problema de Steiner em grafos

- Heurística da rede de distâncias

Rede de distâncias  $D_G=(T,E)$ : para cada  $(i,j) \in T \times T$ :  $w_{ij}$  = comprimento do caminho mais curto de  $i$  a  $j$  em  $G$  em relação aos pesos  $c_{ij}$ .

## **Passo 0:**

Calcular a rede de distâncias  $D_G=(T,E)$ , isto é, os caminhos mais curtos entre cada par de terminais do grafo.

## **Passo 1:**

Obter uma árvore geradora de peso mínimo  $T^*$  da rede de distâncias  $D_G=(T,E)$ .

## **Passo 2:**

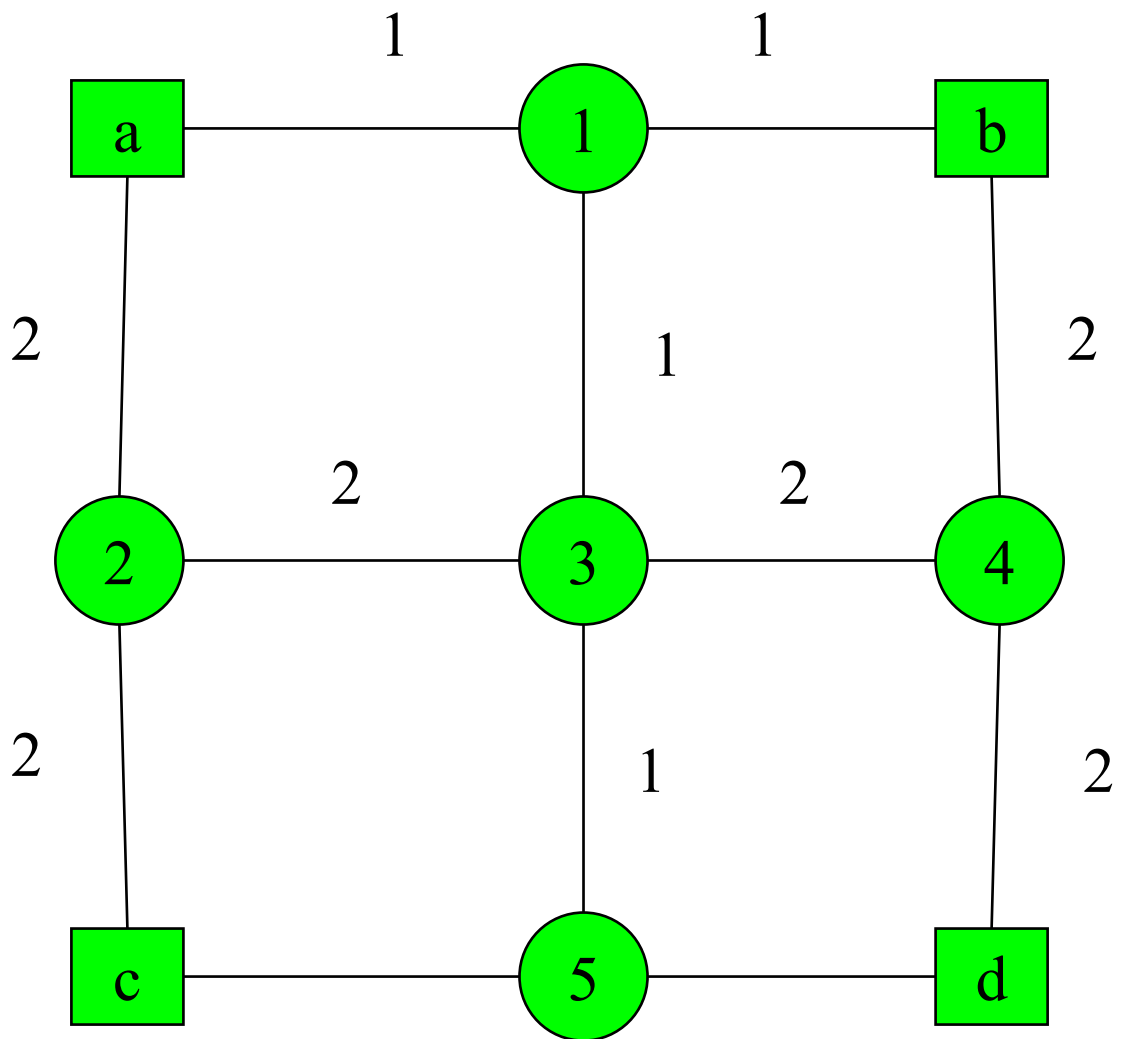
Expandir as arestas de  $T^*$ .

## **Passo 3:**

Eliminar folhas que não sejam terminais.

# Problema de Steiner em grafos

Grafo  $G=(V,E)$



$C_{ab}: a,1,b$  (2)

1

1

$C_{ac}: a,2,c$  (4)

$C_{ad}: a,1,3,5,d$  (4)

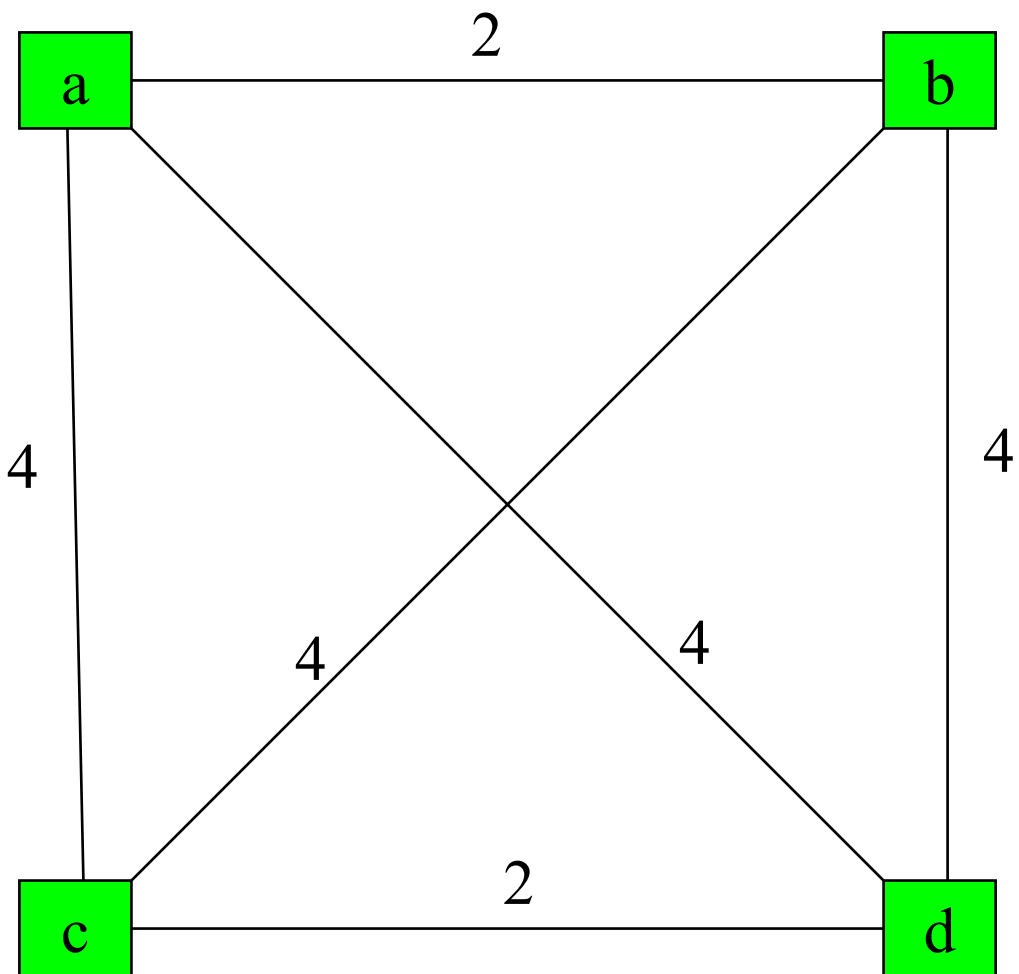
$C_{bc}: b,1,3,5,c$  (4)

$C_{bd}: b,4,d$  (4)

$C_{cd}: c,5,d$  (2)

# Problema de Steiner em grafos

Rede de distâncias  $D_G=(T,E)$



$C_{ab}: a,1,b$  (2)

$C_{ac}: a,2,c$  (4)

$C_{ad}: a,1,3,5,d$  (4)

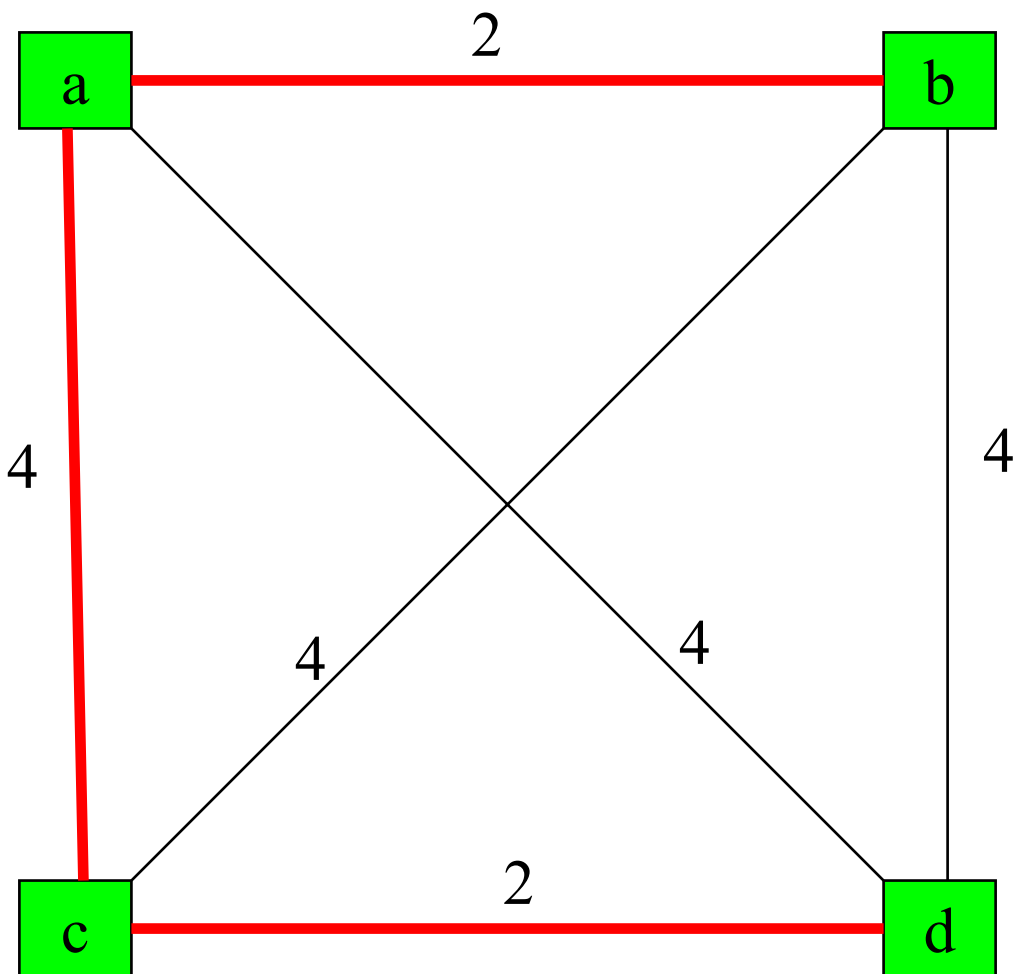
$C_{bc}: b,1,3,5,c$  (4)

$C_{bd}: b,4,d$  (4)

$C_{cd}: c,5,d$  (2)

# Problema de Steiner em grafos

Árvore geradora de peso mínimo da rede de distâncias  $D_G=(T,E)$



$C_{ab}: a,1,b$  (2)

$C_{ac}: a,2,c$  (4)

$C_{ad}: a,1,3,5,d$  (4)

$C_{bc}: b,1,3,5,c$  (4)

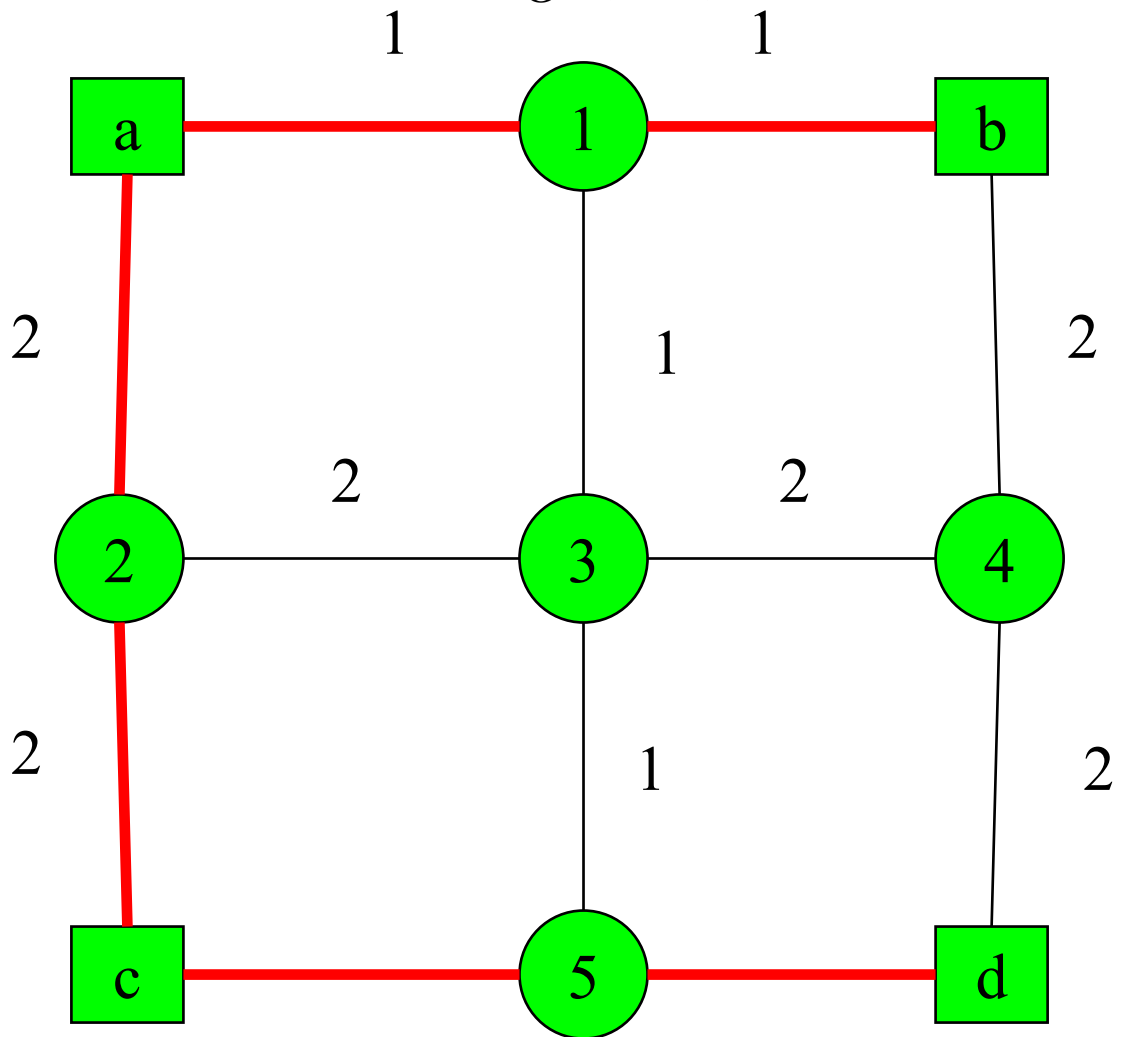
$C_{bd}: b,4,d$  (4)

$C_{cd}: c,5,d$  (2)



# Problema de Steiner em grafos

Expansão da árvore geradora de peso mínimo da rede de distâncias  $D_G=(T,E)$



$C_{ab}: a, 1, b$  (2)

$C_{ac}: a, 2, c$  (4)


$C_{ad}: a, 1, 3, 5, d$  (4)

$C_{bc}: b, 1, 3, 5, c$  (4)

$C_{bd}: b, 4, d$  (4)

$C_{cd}: c, 5, d$  (2)

# Representação de soluções

- **Busca local**: técnica de exploração sistemática do espaço de soluções, à procura de melhores soluções.
- Conjunto  $F$  de soluções (viáveis) definido por subconjuntos de um conjunto  $E$  (suporte/base) de elementos que satisfazem determinadas condições.
- Representação de uma solução: indicar quais elementos de  $E$  estão presentes e quais não estão. 
- Problema da mochila:  $n$  itens, vetor 0-1 com  $n$  posições,  $x_j = 1$  se o item  $j$  é selecionado,  $x_j = 0$  caso contrário

# Representação de soluções

- Problema do caixeiro viajante

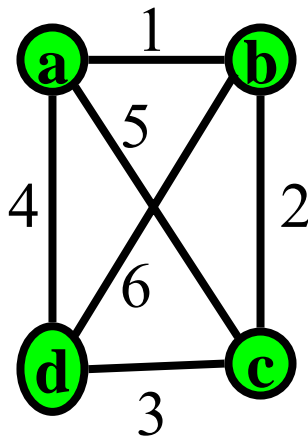
E: conjunto de arestas

F: subconjuntos de E que formam um circuito Hamiltoniano

Solução é um vetor de  $n = |E|$  posições

$v_e = 1$ , se a aresta  $e$  pertence ao CH

$v_e = 0$ , caso contrário.



Soluções viáveis (64 possibilidades):

$(1,1,1,1,0,0)$ ,  $(1,0,1,0,1,1)$ ,  $(0,1,0,1,1,1)$

# Representação de soluções

- Outra representação para as soluções do PCV: representar cada solução pela ordem em que os vértices são visitados, isto é, como uma permutação circular dos  $n$  vértices (já que o primeiro vértice é arbitrário)
  - (a)bcd
  - (a)bdc
  - (a)cbd
  - (a)cdb
  - (a)dbc
  - (a)dc b

# Representação de soluções

- Indicadores 0-1 de pertinência:
  - Problema da mochila
  - Problema de Steiner em grafos
  - Problemas de recobrimento e de particionamento
- Indicadores gerais de pertinência:
  - Particionamento de grafos
  - Coloração de grafos
  - *Clustering*
- Permutações:
  - Problemas de escalonamento
    - *Job/Flow/Open Shop Scheduling*
  - Problema do caixeiro viajante

# Vizinhanças

- Problema combinatório:

$f(s^*) = \text{mínimo } \{f(s) : s \in S\}$  S  
é um conjunto discreto de soluções

- Vizinhança: elemento que introduz a noção de proximidade entre as soluções em S.

- Uma vizinhança é um mapeamento

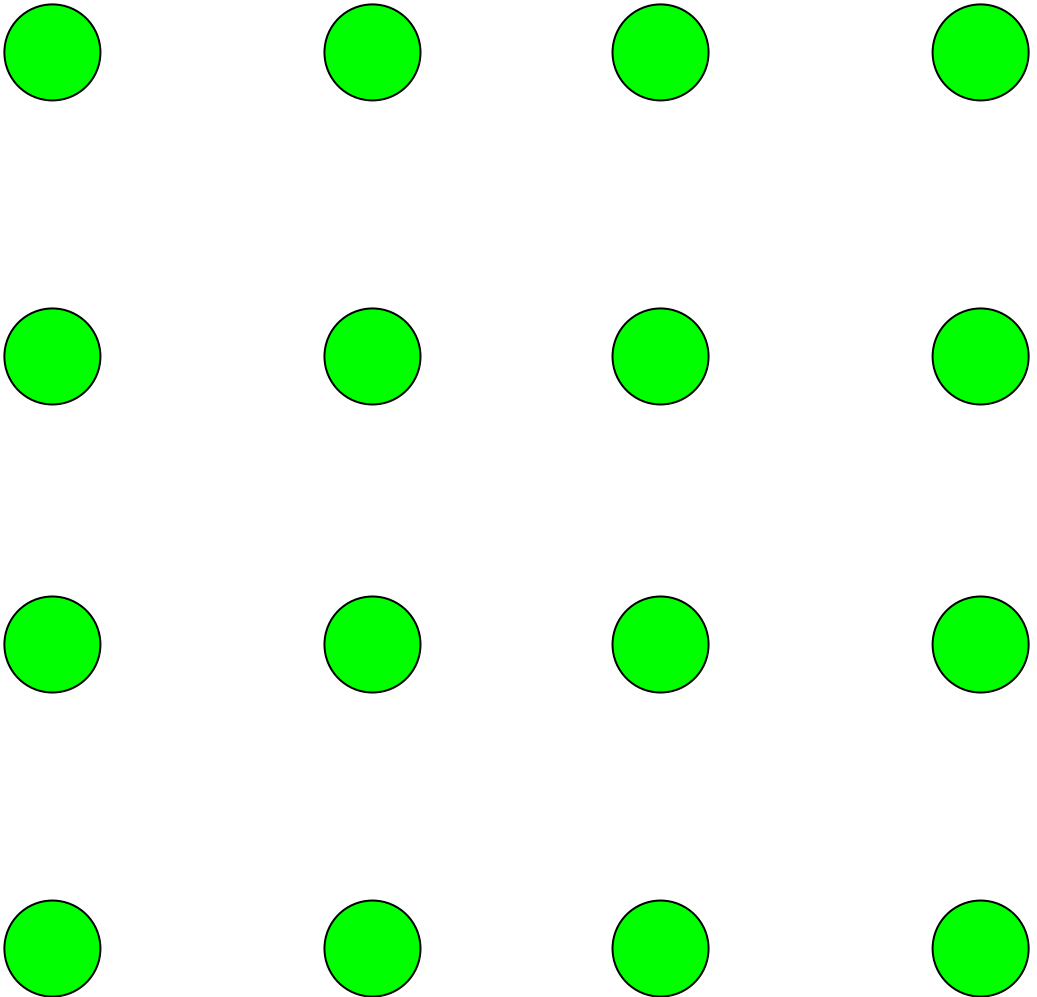
$$N: S \rightarrow 2^S$$

que leva as soluções de S em um subconjunto deste mesmo conjunto de soluções.

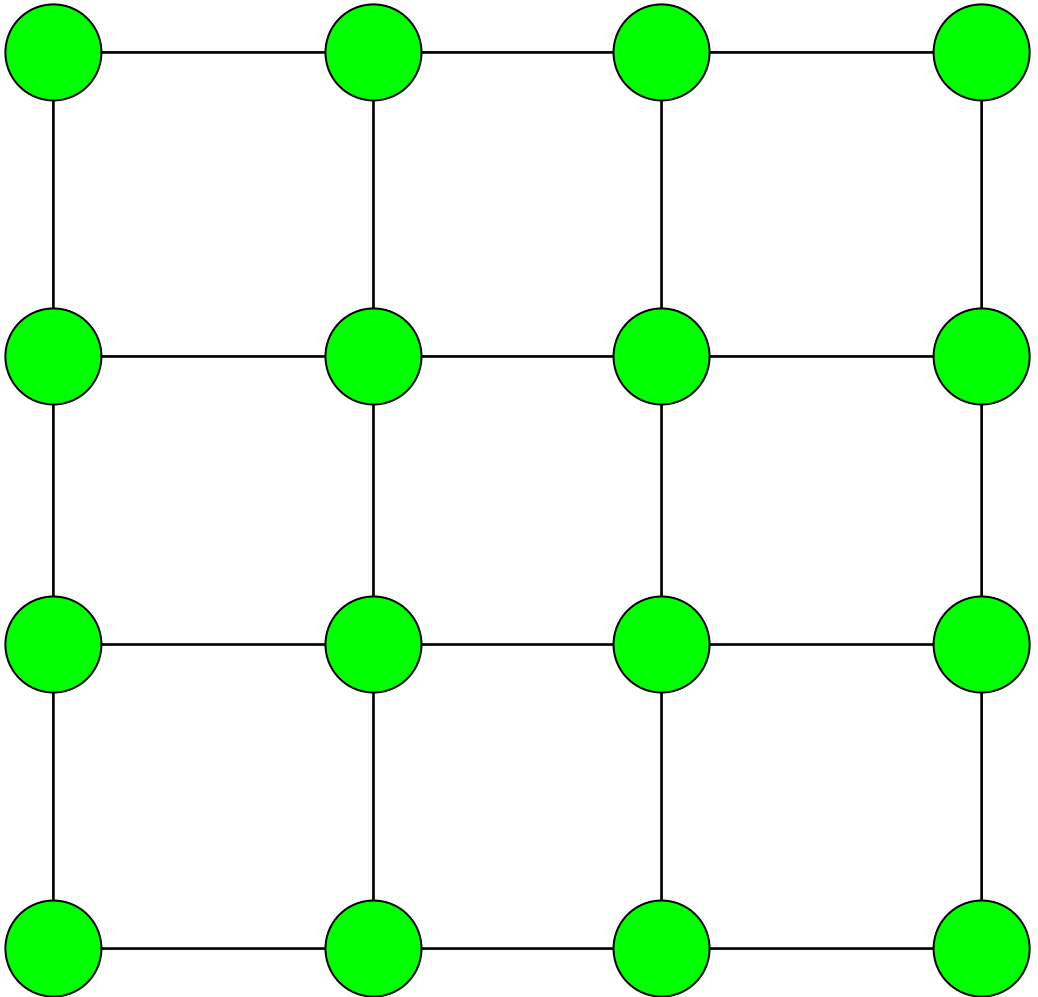
$$N(s) = \{s_1, s_2, \dots, s_k\} \text{ soluções vizinhas de } s$$

- Boas vizinhanças permitem representar de forma compacta/eficiente o conjunto de soluções vizinhas a qualquer solução s

# Vizinhanças

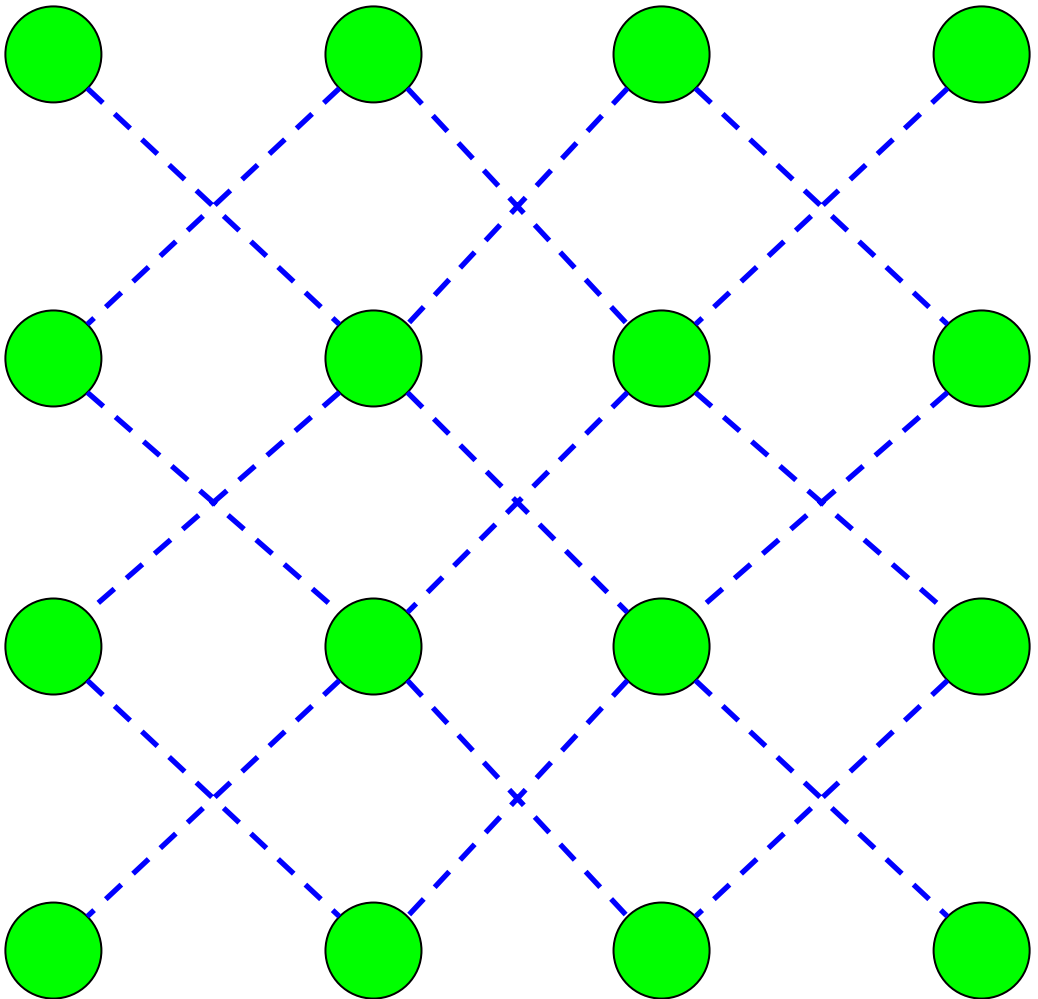


# Vizinhanças

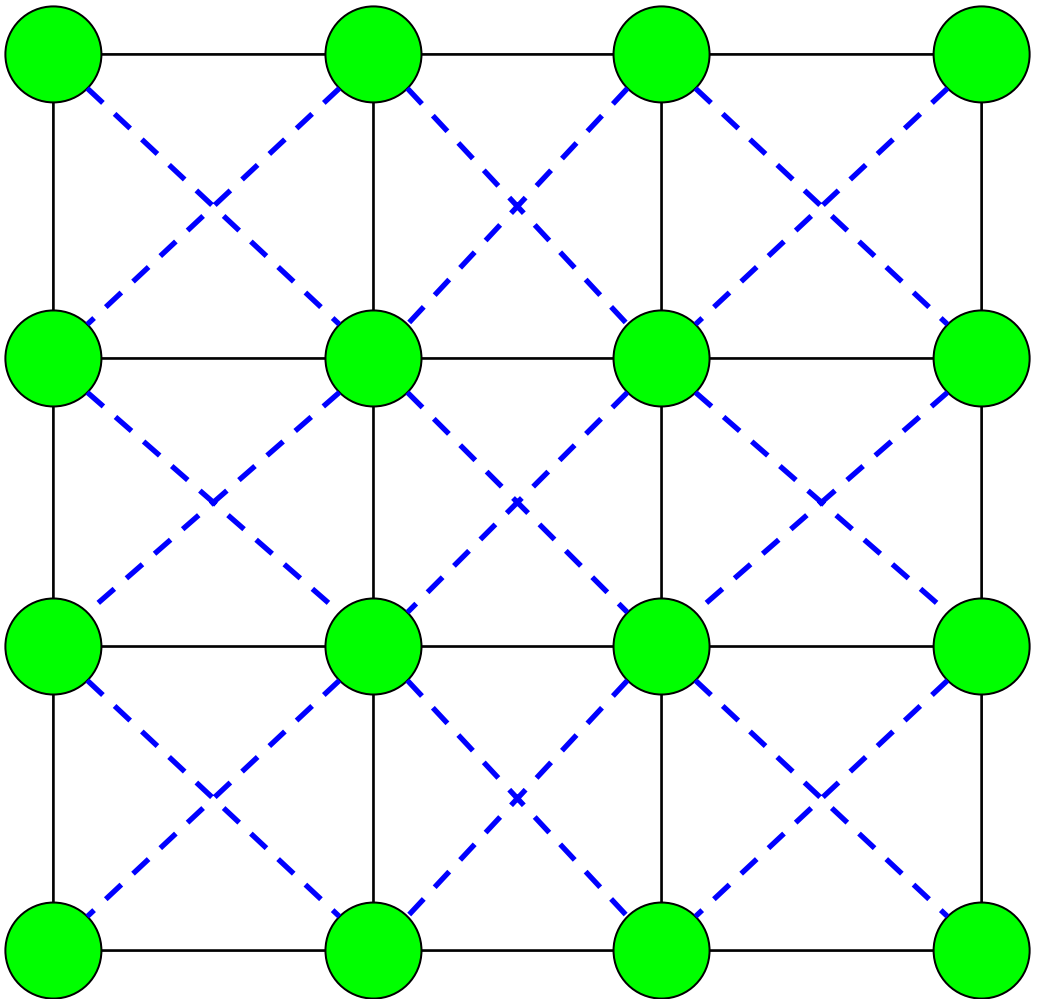




# Vizinhanças



# Vizinhanças



# Vizinhanças

- Vizinhanças no espaço de permutações:
- Solução  $\pi = (\pi_1, \dots, \pi_{i-1}, \pi_i, \pi_{i+1}, \dots, \pi_j, \dots, \pi_n)$
- $N1(\pi) = \{(\pi_1, \dots, \pi_{i+1}, \pi_i, \dots, \pi_n) : i=1, \dots, n-1\}$   
Vizinhos de  $(1, 2, 3, 4) = \{(2, 1, 3, 4), (1, 3, 2, 4), (1, 2, 4, 3)\}$
- $N2(\pi) = \{(\pi_1, \dots, \pi_j, \dots, \pi_i, \dots, \pi_n) : i=1, \dots, n-1; j=i+1, \dots, n\}$   
Vizinhos de  $(1, 2, 3, 4) = \{(2, 1, 3, 4), (1, 3, 2, 4), (1, 2, 4, 3), (3, 2, 1, 4), (1, 4, 3, 2), (4, 2, 3, 1)\}$
- $N3(\pi) = \{(\pi_1, \dots, \pi_{i-1}, \pi_{i+1}, \dots, \pi_j, \pi_i, \dots, \pi_n) : i=1, \dots, n-1; j=i+1, \dots, n\}$   
Vizinhos de  $(1, 2, 3, 4) = \{(2, 1, 3, 4), (2, 3, 1, 4), (2, 3, 4, 1), (1, 3, 2, 4), (1, 3, 4, 2), (1, 2, 4, 3)\}$



# Vizinhanças

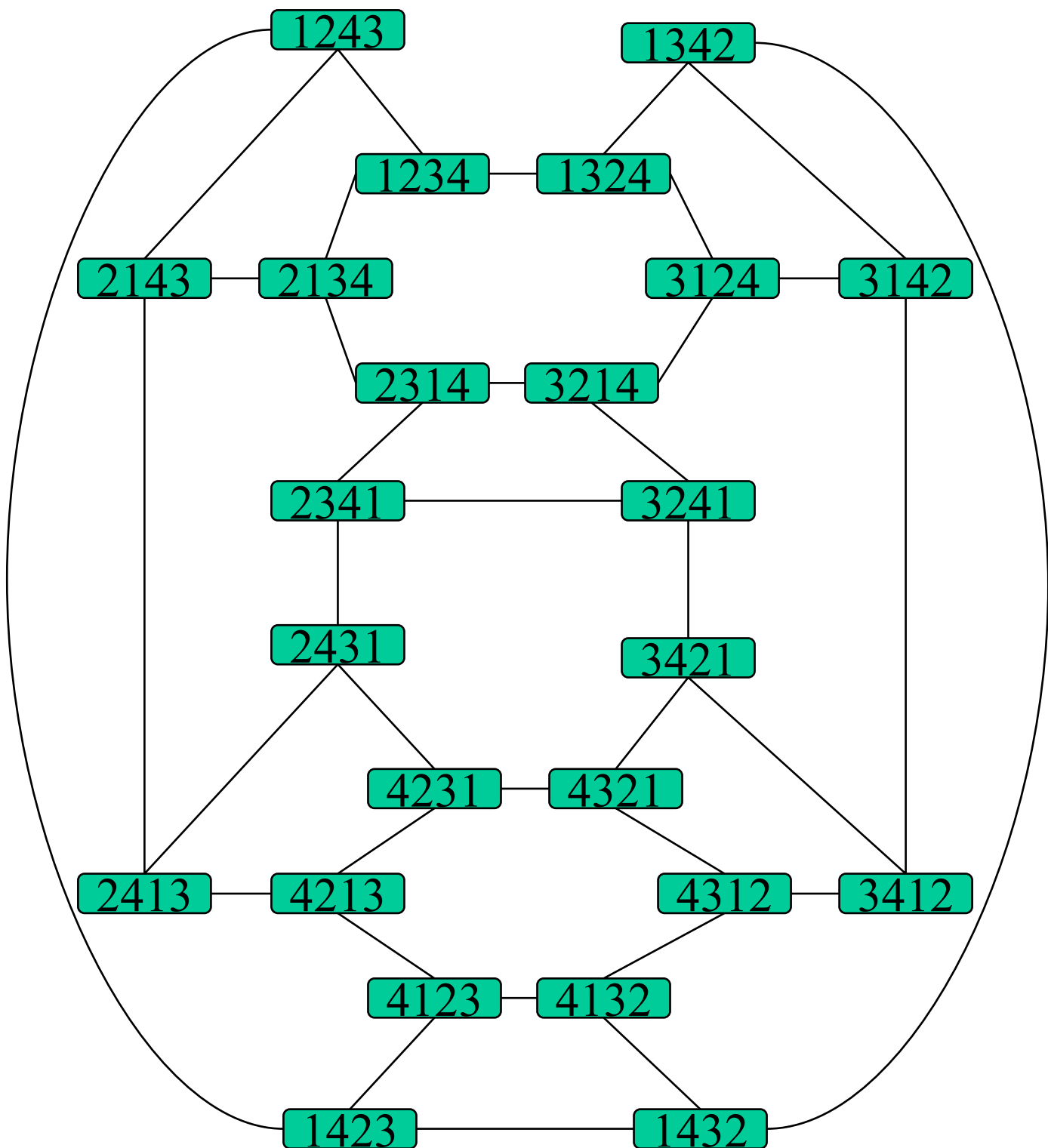
- Espaço de busca: definido pelo conjunto de soluções  $S$  e por uma vizinhança  $N$
- Exemplo 1:  
permutações com a vizinhança  $N1$
- Exemplo 2:  
vetores de pertinência 0-1 com a vizinhança  $N4$ .

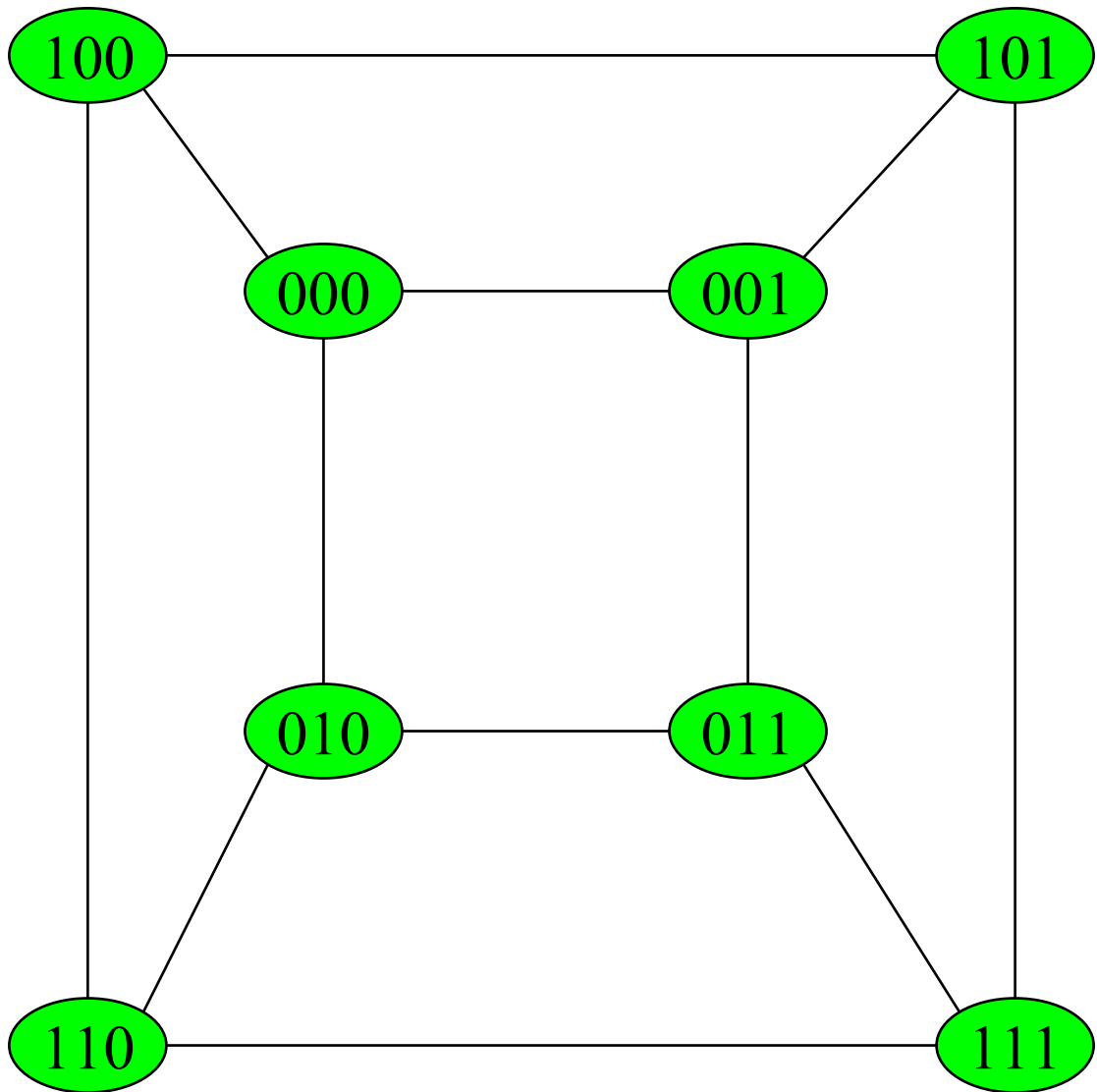
$$v = (v_1, \dots, v_i, \dots, v_n)$$

$$N4(v) = \{(v_1, \dots, 1-v_i, \dots, v_n) : i=1, \dots, n\}$$

Vizinhos de  $(1,0,1,1) =$

$$\{(0,0,1,1), (1,1,1,1), (1,0,0,1), (1,0,1,0)\}$$





# Vizinhanças

- O espaço de busca pode ser visto como um grafo onde os vértices são as soluções e existem arestas entre pares de vértices associados a soluções vizinhas.
- Este espaço pode ser visto como uma superfície com vales e cumes definidos pelo valor e pela proximidade (vizinhança) das soluções.
- Um caminho no espaço de busca consiste numa seqüência de soluções, onde duas soluções consecutivas quaisquer são vizinhas.

# Vizinhanças

- A noção de proximidade induz o conceito de distância entre soluções, que define um espaço topológico.
- Ótimo local: solução tão boa ou melhor do que qualquer das soluções vizinhas
- Problema de minimização:

$s^+$  é um ótimo local

$\uparrow\downarrow$

$$f(s^+) \leq f(s), \quad \forall s \in N(s^+)$$

- Ótimo global ou solução ótima  $s^*$ :

$$f(s^*) \leq f(s), \quad \forall s \in S$$



# Busca local

- Algoritmos de busca local são construídos como uma forma de exploração do espaço de busca.
- Partida: solução inicial obtida através de um método construtivo
- Iteração: melhoria sucessiva da solução corrente através de uma busca na sua vizinhança
- Parada: primeiro ótimo local encontrado (não existe solução vizinha aprimorante)
- Heurística subordinada utilizada para obter uma solução aprimorante na vizinhança

# Busca local

- Questões fundamentais:
  - Definição da vizinhança
  - Estratégia de busca na vizinhança
  - Complexidade de cada iteração:
    - Proporcional ao tamanho da vizinhança
    - Eficiência depende da forma como é calculada a variação da função objetivo para cada solução vizinha: algoritmos eficientes são capazes de recalcular as variações de modo a atualizá-las quando a solução corrente se modifica, evitando cálculos repetitivos e desnecessários da função objetivo.

# Busca local

- Melhoria iterativa: a cada iteração, selecionar qualquer (eventualmente a primeira) solução aprimorante na vizinhança

```
procedure Melhoria-Iterativa( $s_0$ )  
   $s \leftarrow s_0$ ; melhoria  $\leftarrow$  .verdadeiro.  
  while melhoria do  
    melhoria  $\leftarrow$  .falso.  
    for-all  $s' \in N(s)$  e melhoria =.falso. do  
      if  $f(s') < f(s)$  then  
         $s \leftarrow s'$ ; melhoria  $\leftarrow$  .verdadeiro.  
      end-if  
    end-for-all  
  end-while  
  return  $s$   
end Melhoria-Iterativa
```



# Busca local

- Descida mais rápida: selecionar a melhor solução aprimorante na vizinhança

**procedure** Descida-Mais-Rápida( $s_0$ )

$s \leftarrow s_0$ ; melhoria  $\leftarrow$  .verdadeiro.

**while** melhoria **do**

    melhoria  $\leftarrow$  .falso.;  $f_{\min} \leftarrow +\infty$

**for-all**  $s' \in N(s)$  **do**

**if**  $f(s') < f_{\min}$  **then**

$s_{\min} \leftarrow s'$ ;  $f_{\min} \leftarrow f(s')$

**end-if**

**end-for-all**

**if**  $f_{\min} < f(s)$  **then**

$s \leftarrow s_{\min}$ ; melhoria  $\leftarrow$  .verdadeiro.

**end-if**

**end-while**

**return**  $s$

**end** Descida-Mais-Rápida



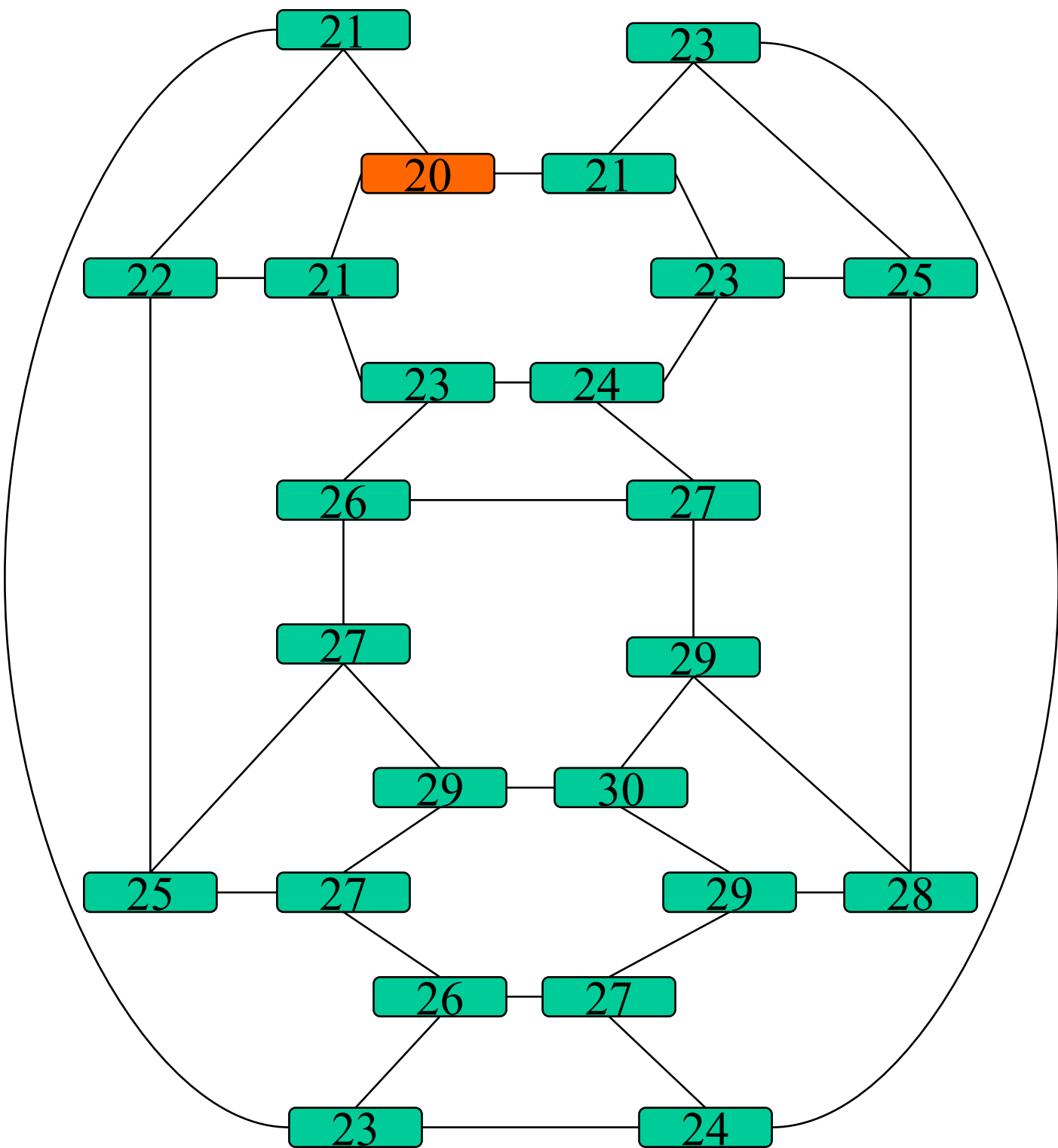
# Busca local

- Exemplo: algoritmo de descida mais rápida aplicado ao problema de ordenação
- Espaço de busca: permutações de  $n$  elementos
- Solução  $\pi = (\pi_1, \dots, \pi_{i-1}, \pi_i, \pi_{i+1}, \dots, \pi_j, \dots, \pi_n)$
- Vizinhaça:  
 $N1(\pi) = \{(\pi_1, \dots, \pi_{i+1}, \pi_i, \dots, \pi_n) : i=1, \dots, n-1\}$
- Custo de uma permutação:  
 $f(\pi) = \sum_{i=1, \dots, n} i \cdot \pi_i$

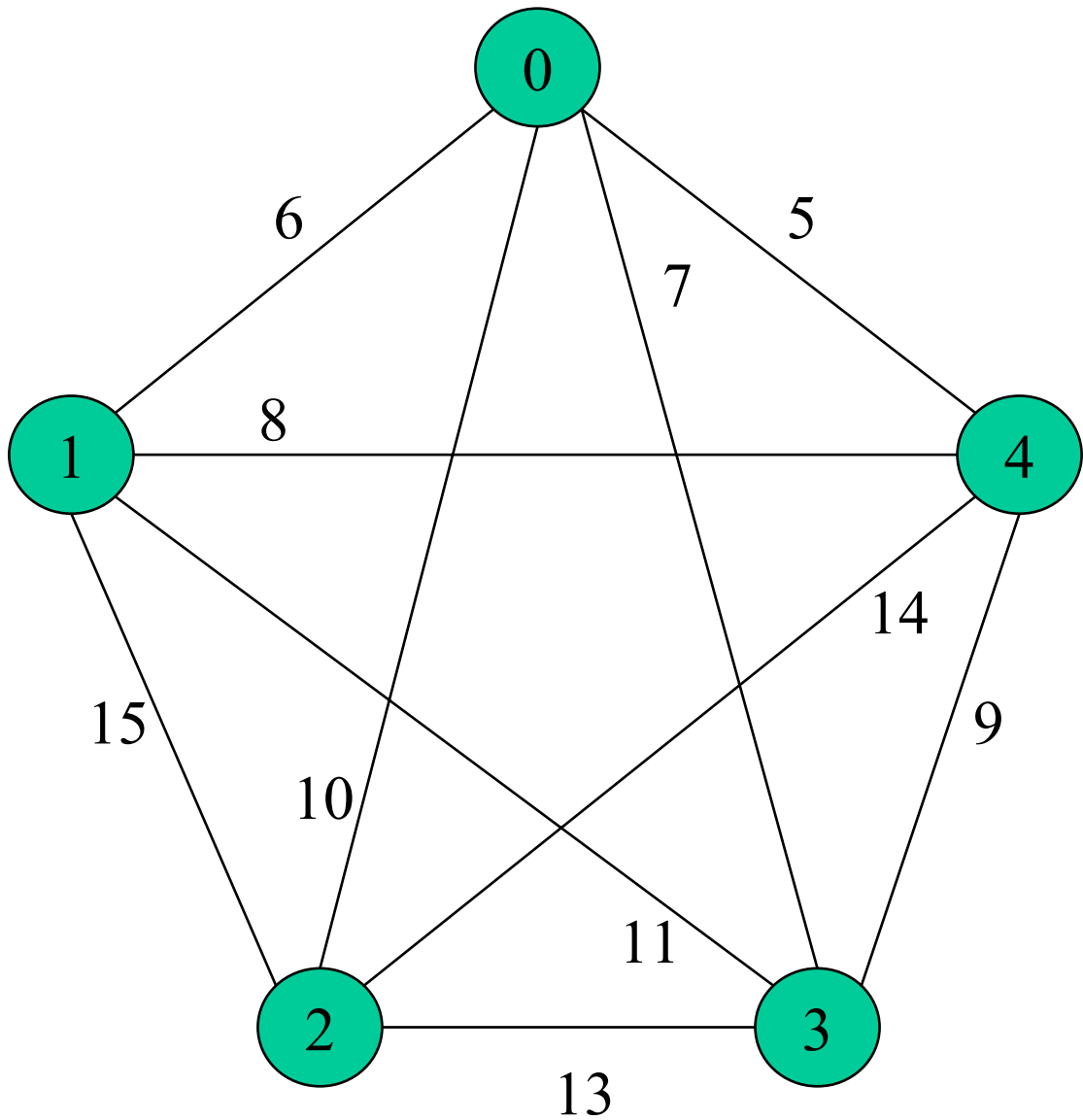


# Busca local

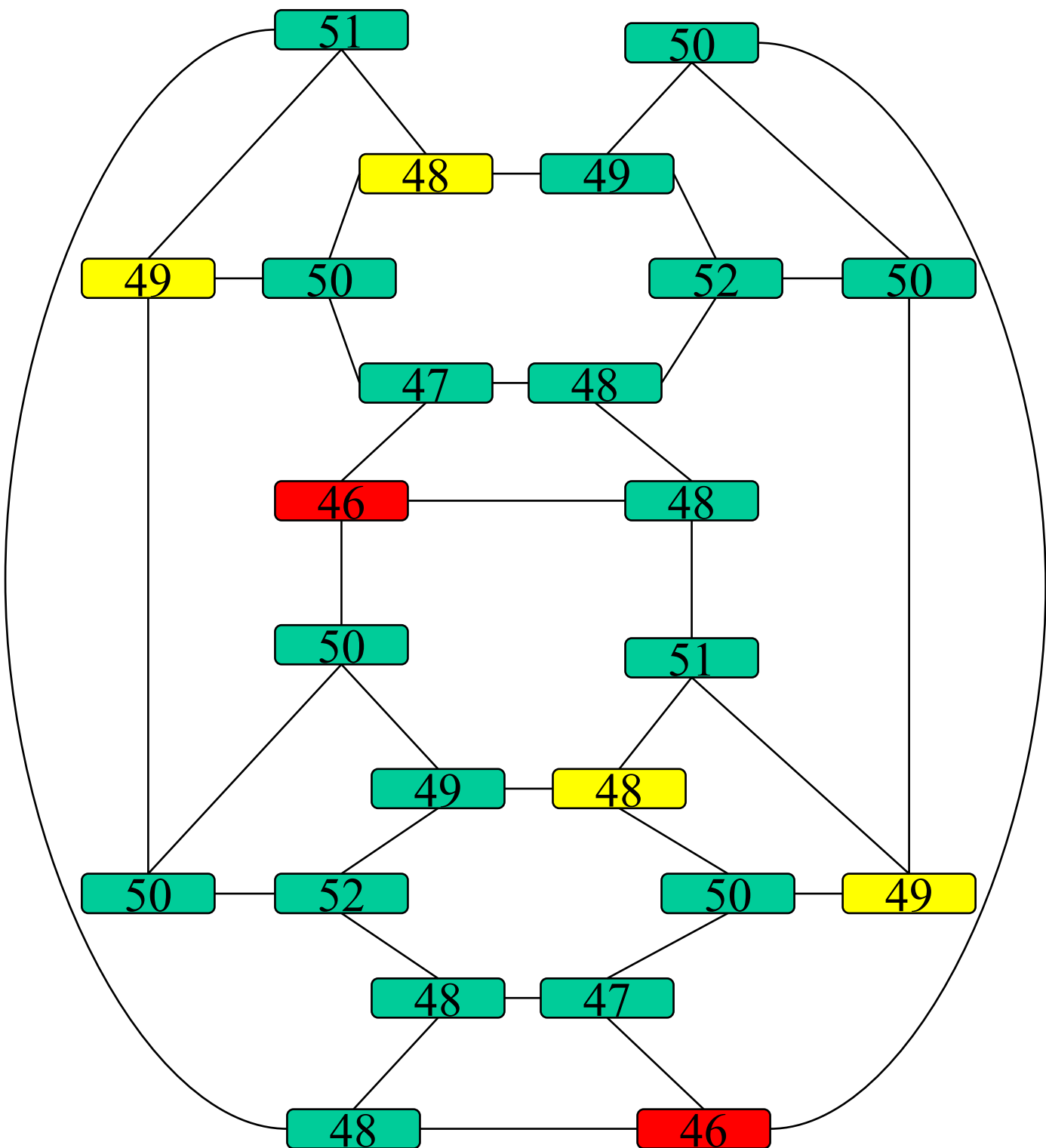
```
procedure BL-Perm-N1( $\pi_0$ )  
   $\pi \leftarrow \pi_0$ ; melhora  $\leftarrow$  .verdadeiro.  
  while melhora do  
    melhora  $\leftarrow$  .falso.;  $f_{\min} \leftarrow +\infty$   
    for  $i = 1$  to  $n-1$  do  
       $\pi' \leftarrow \pi$ ;  $\pi'_i \leftarrow \pi_{i+1}$ ;  $\pi'_{i+1} \leftarrow \pi_i$ ;  
      if  $f(\pi') < f_{\min}$  then  
         $\pi_{\min} \leftarrow \pi'$  ;  $f_{\min} \leftarrow f(\pi')$   
      end-if  
    end-for  
    if  $f_{\min} < f(\pi)$  then  
       $\pi \leftarrow \pi_{\min}$  ; melhora  $\leftarrow$  .verdadeiro.  
    end-if  
  end-while  
   $\pi^+ \leftarrow \pi$   
  return  $\pi^+$   
end BL-Perm-N1
```



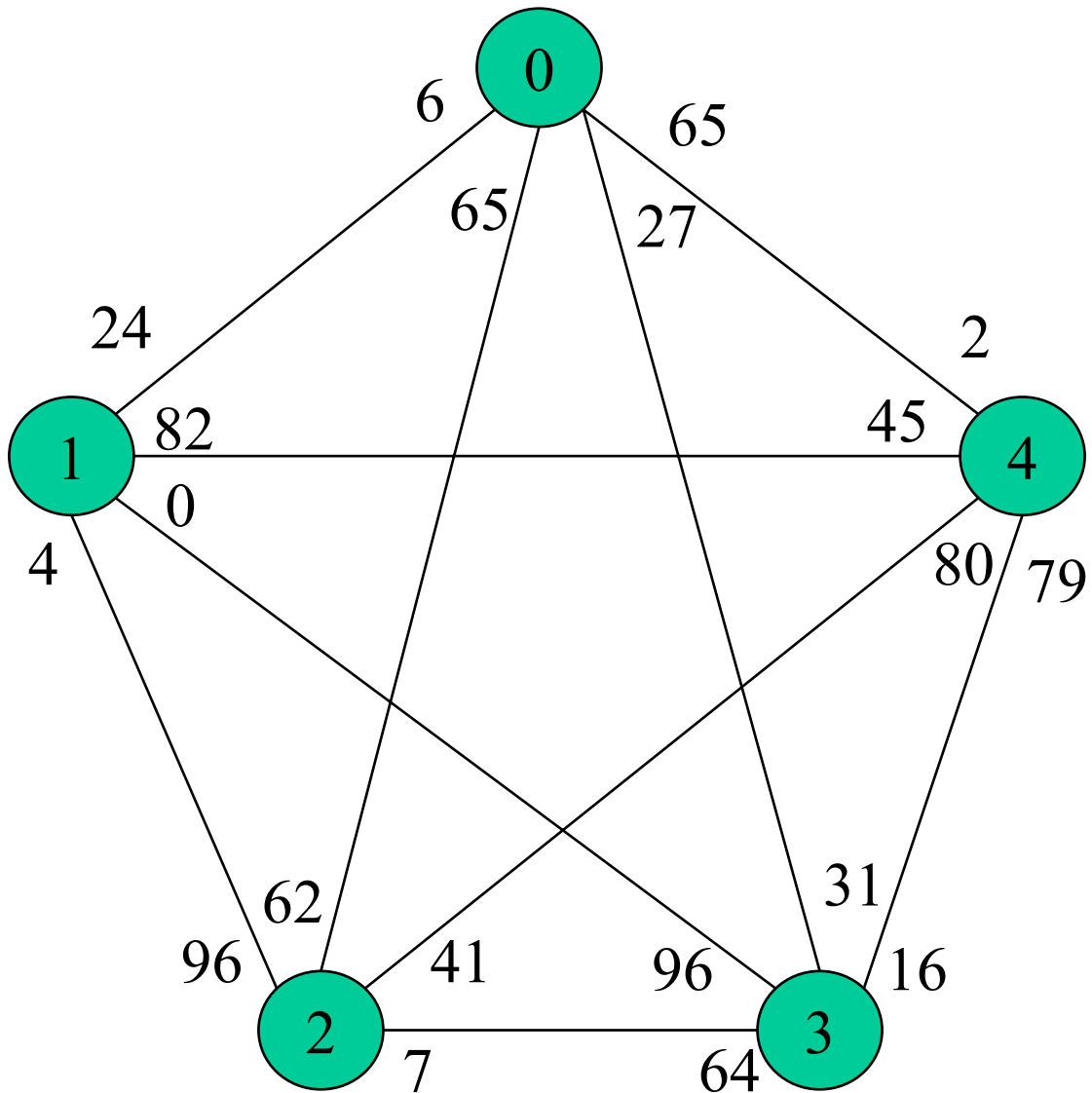
# Caixeiro viajante simétrico

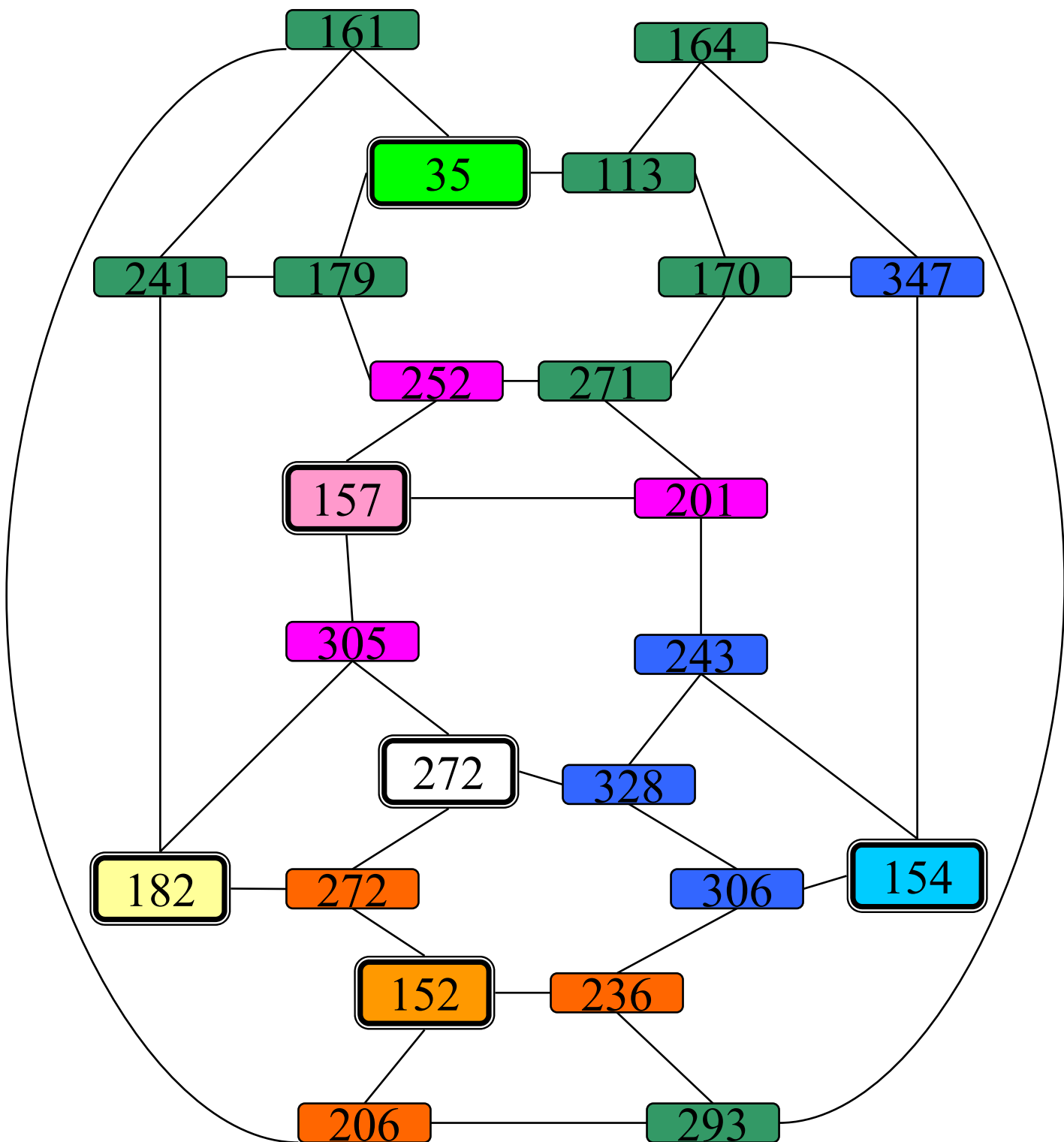






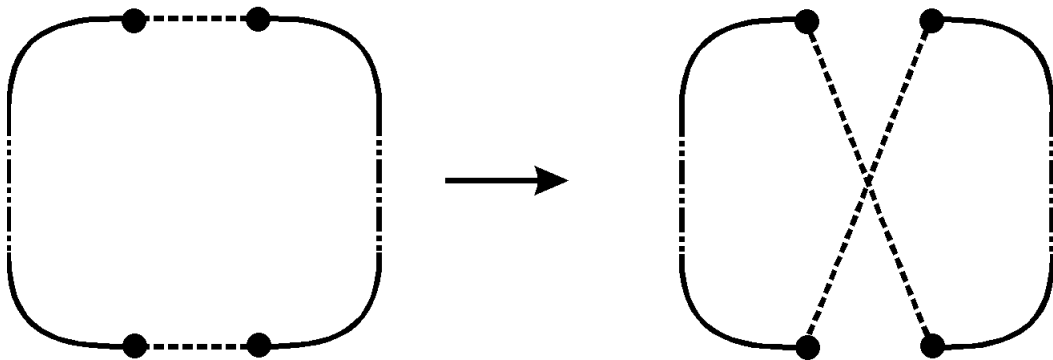
# Caixeiro viajante assimétrico





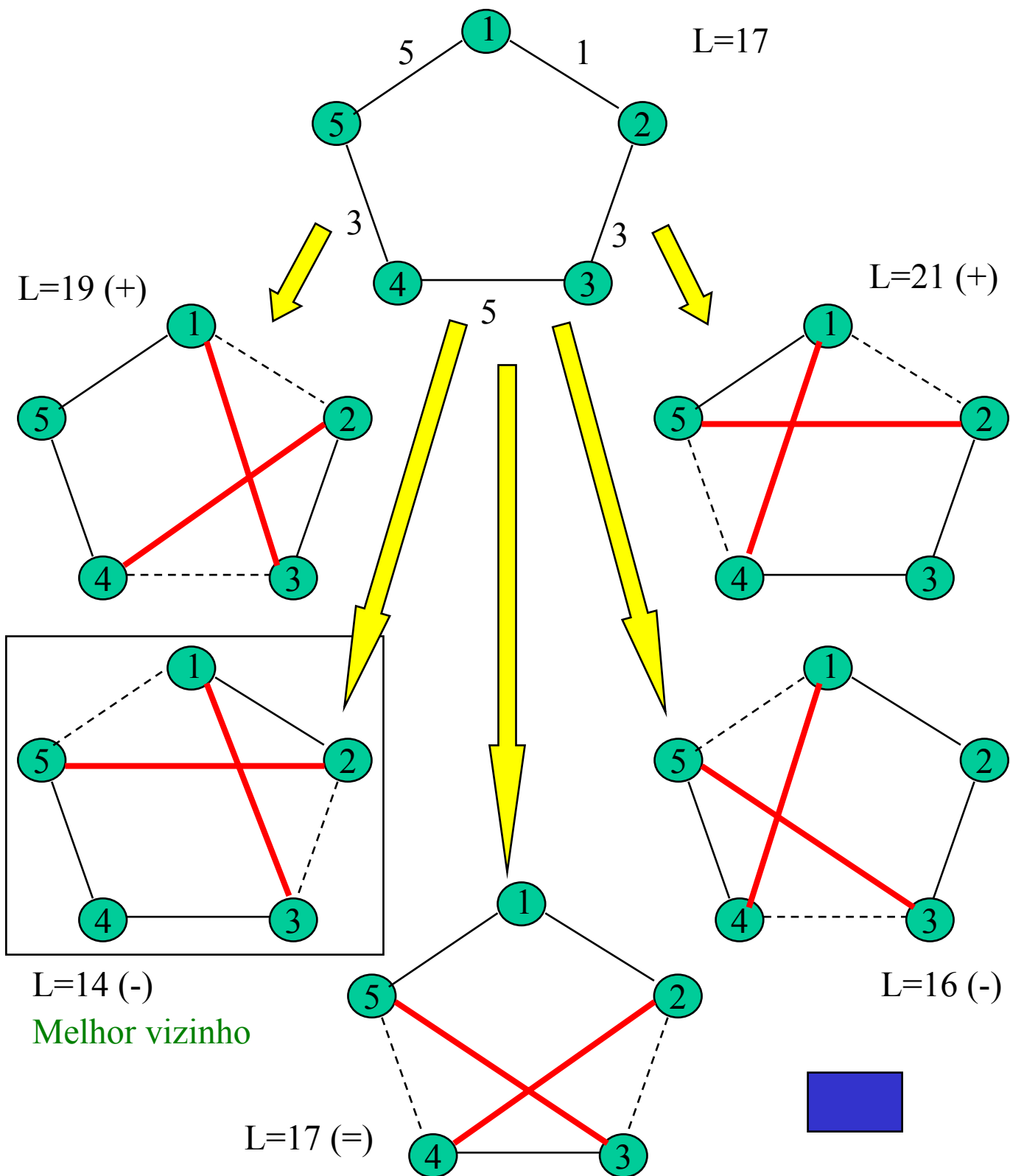
# Busca local para o PCV

- Vizinhaça 2-opt para o problema do caixeiro viajante:



- Há um vizinho para cada par válido de arestas, logo o número de vizinhos é  $O(n^2)$
- Como o custo de cada vizinho pode ser avaliado em  $O(1)$ , a complexidade de cada iteração da busca local é  $O(n^2)$

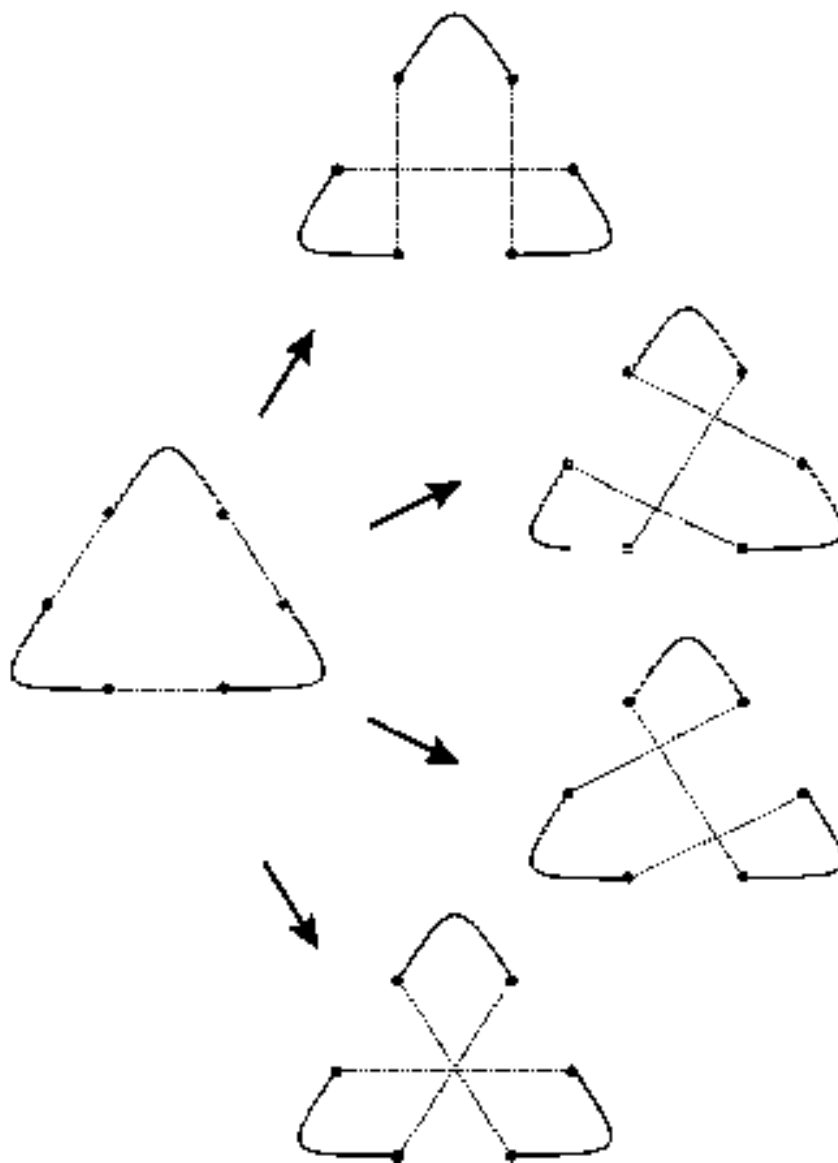
# Busca local para o PCV



Melhor vizinho

# Busca local para o PCV

- Vizinhança 3-opt para o problema do caixeiro viajante:



# Busca local para o PCV

- Há um vizinho para cada tripla válida de arestas, logo o número de vizinhos é  $O(n^3)$
- Como o custo de cada vizinho pode ser avaliado em  $O(1)$ , a complexidade de cada iteração da busca local é  $O(n^3)$
- A vizinhança  $(k+1)$ -opt inclui as soluções da vizinhança  $k$ -opt
- Este processo pode ser levado até  $n$ -opt, que corresponde a uma busca exaustiva do espaço de soluções, mas...
- À medida em que  $k$  aumenta, aumentam também a cardinalidade da vizinhança e a complexidade de cada iteração, enquanto o ganho possível diminui progressivamente.

# Busca local

- Diferentes aspectos do espaço de busca influenciam o desempenho de algoritmos de busca local
- Conexidade: deve existir um caminho entre qualquer par de soluções no espaço de busca
- Distância entre duas soluções: número de soluções visitadas ao longo de um caminho mais curto entre elas
- Diâmetro: distância entre duas das soluções mais afastadas (diâmetros reduzidos)
- Bacia de atração de um ótimo local: conjunto de soluções iniciais a partir das quais o algoritmo de **descida mais rápida** leva a este ótimo local.



# Busca local

- Dificuldades:
  - Término prematuro no primeiro ótimo local encontrado
  - Sensível à solução de partida
  - Sensível à vizinhança escolhida
  - Sensível à estratégia de busca
  - Pode exigir um número exponencial de iterações!

# Busca local

- Extensões para contornar algumas dificuldades da busca local
- Redução da vizinhança: investigar um subconjunto da vizinhança da solução corrente (e.g. por aleatorização)
- Multi-partida: repetir a busca local a partir de diferentes soluções
- Multi-vizinhança: considera mais de uma vizinhança. Ao atingir um ótimo local com relação a uma vizinhança, inicia uma outra busca local empregando outra vizinhança. O algoritmo termina quando a solução corrente é um ótimo local em relação a todas as vizinhanças empregadas.
- Segmentação da vizinhança: utilizada para aumentar a eficiência quando vizinhanças muito grandes são utilizadas, pode ser vista como uma estratégia multi-vizinhança.

$$N(s) = N_1(s) \cup N_2(s) \cup \dots \cup N_p(s)$$

# Busca local

**procedure** BL-Segmentada( $s_0$ )

$s \leftarrow s_0$ ; melhoria  $\leftarrow$  .verdadeiro.

**while** melhoria **do**

    melhoria  $\leftarrow$  .falso.;  $f_{\min} \leftarrow +\infty$ ;  $j \leftarrow 1$

**while**  $f_{\min} \geq f(s)$  **e**  $j \leq p$  **do**

**for-all**  $s' \in N_j(s)$  **do**

**if**  $f(s') < f_{\min}$  **then**

$s_{\min} \leftarrow s'$ ;  $f_{\min} \leftarrow f(s')$

**end-if**

**end-for-all**

**if**  $f_{\min} < f(s)$  **then**

$s \leftarrow s_{\min}$ ; melhoria  $\leftarrow$  .verdadeiro.

**end-if**

$j \leftarrow j + 1$

**end-while**

**end-while**

**return**  $s$

**end** BL-Segmentada

# Metaheurísticas

- *Simulated annealing*
- GRASP
- VNS (*Variable Neighborhood Search*)
- VND (*Variable Neighborhood Descent*)
- Busca tabu
- Algoritmos genéticos
- Colônias de formigas

# Simulated annealing

- Princípio: analogia entre um processo de mecânica estatística e a solução de um problema de otimização combinatória
- O termo *annealing* refere-se a um processo térmico que começa pela liquidificação de um cristal a uma alta temperatura, seguido pela lenta e gradativa diminuição de sua temperatura, até que o ponto de solidificação seja atingido, quando o sistema atinge um estado de energia mínima.
- função objetivo  $f(s) \Leftrightarrow$  nível de energia  
solução viável  $\Leftrightarrow$  estado do sistema  
solução vizinha  $\Leftrightarrow$  mudança de estado  
parâmetro de controle  $\Leftrightarrow$  temperatura  
melhor solução  $\Leftrightarrow$  estado de solidificação

# Simulated annealing

- Algoritmo básico:

$s \leftarrow s_0; T \leftarrow T_0$

**while** temperatura elevada **do**

**for** iterações para equilíbrio **do**

        Gerar uma solução  $s'$  de  $N(s)$

        Avaliar a variação de energia

$$\Delta E = f(s') - f(s)$$

**if**  $\Delta E < 0$  **then**  $s \leftarrow s'$

**else**

            Gerar  $u \in \text{Unif}[0,1]$

**if**  $u < \exp(-\Delta E/K_B.T)$

**then**  $s \leftarrow s'$

**end-if**

**end-for**

    Reduzir a temperatura  $T$

**end-while**

# Simulated annealing

- Questão básica: escalonamento da redução da temperatura  $T$
- Controla a convergência do algoritmo.
- $T_k$ : temperatura no ciclo  $k$  de iterações.
  - Executar  $h(T_k)$  iterações para se atingir o equilíbrio nesta temperatura.
  - Atualização geométrica de  $T$ :  
$$T_{k+1} \leftarrow \beta \cdot T_k \text{ onde } 0 < \beta < 1$$

# Simulated annealing

- Observações:
  - T elevada:  $\exp(-\Delta E/K_B.T)$  próximo de um e, conseqüentemente, quase todas as soluções vizinhas são aceitas (algoritmo permanece “vagando “ no espaço de busca)
  - T próxima de zero: somente são aceitas soluções que melhoram a solução corrente (comportamento análogo ao algoritmo de **melhoria iterativa**)
- Controle da deterioração da solução corrente: no início, uma solução vizinha pior que a corrente é aceita com alta probabilidade (randomização). Quando a temperatura diminui, a probabilidade de aceitação vai sendo gradativamente reduzida até zero (estabilização).
- Capaz de escapar de ótimos locais



# Simulated annealing

- Existe prova de convergência para a solução ótima: entretanto, a velocidade de redução de temperatura exigida implica em visitar um número exponencial de soluções.
- Implementação simples: como só visita uma solução a cada iteração, calcular o valor da função objetivo da solução vizinha gerada não degrada demasiadamente a eficiência do algoritmo.
- Processo lento de redução da temperatura é necessário para se obter soluções competitivas, implicando em tempos de processamento elevados
- Utiliza como informação do problema somente a variação do valor da função objetivo, ou seja é pouco “inteligente”

# Simulated annealing

- Método essencialmente não-determinístico: depende fundamentalmente de geradores de números pseudo-aleatórios
- Parâmetros a serem ajustados:
  - Solução inicial
  - Temperatura inicial
  - Número de iterações no mesmo patamar de temperatura
  - Fator de redução da temperatura
  - Tolerância (critério de parada)

# Simulated annealing

- Paralelização: deve-se procurar produzir seqüências distintas de soluções visitadas (geração da solução inicial e escolha da solução vizinha)
- Estratégia A
  - Processadores usam trechos diferentes da seqüência de números pseudo-aleatórios, gerando diferentes cadeias de soluções visitadas, até o ponto de redução da temperatura.
  - Todos os processadores alteram a temperatura ao mesmo tempo e recomeçam a partir da melhor solução.
  - Temperaturas elevadas: cadeias de soluções diferentes, pois a probabilidade de aceitação de soluções não-aprimorantes é grande (randomização).
  - Temperaturas reduzidas: processadores obtêm soluções iguais ou próximas

# Simulated annealing

- Estratégia B
  - Cada processador gera aleatoriamente uma solução vizinha diferente e testa sua aceitação independentemente dos demais.
  - Cada vez que uma solução vizinha é aceita por um processador, ela é disseminada para os demais.
  - Todos processadores passam a utilizar esta mesma solução como solução corrente, examinando sua vizinhança.
  - Temperaturas altas: não é eficiente, pois quase todos os vizinhos são aceitos.
  - Temperaturas reduzidas: permite acelerar consideravelmente a busca, identificando mais rapidamente um vizinho aceito numa fase do algoritmo em que normalmente a maioria das soluções seriam rejeitadas.

# Simulated annealing

- Estratégia C
  - Iniciar com a estratégia A (mais eficiente em temperaturas elevadas) até que a taxa de soluções rejeitadas exceda um certo limiar, passando-se então para a estratégia B (mais eficiente a temperaturas reduzidas).
- Estratégia D
  - Manter sempre a melhor solução corrente na memória central
  - Todos os processadores trabalham independentemente, cada um visitando uma cadeia de soluções diferente.

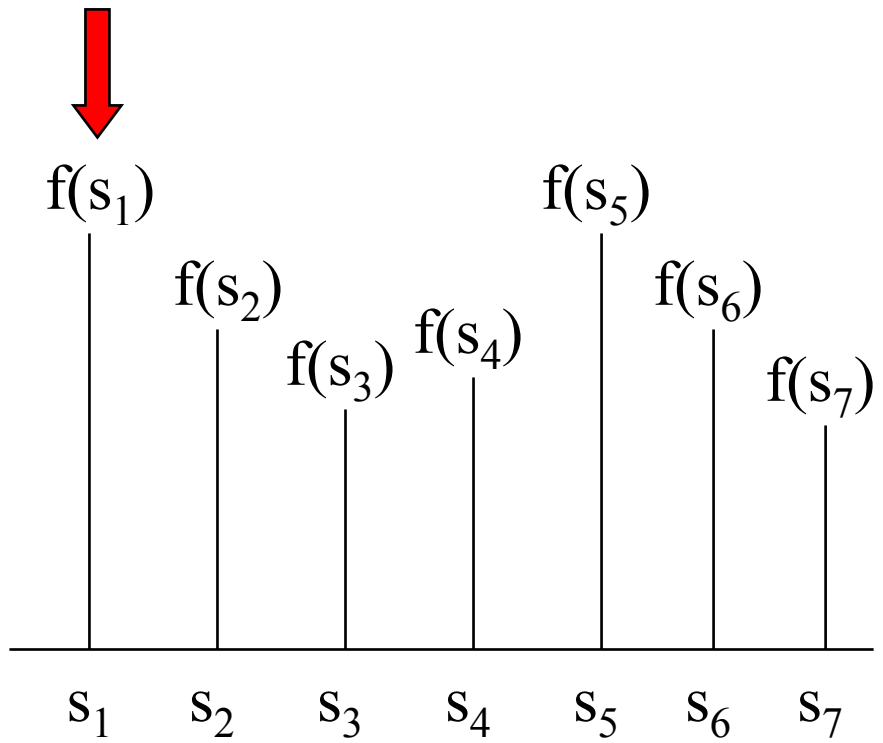
# Busca tabu

- Busca tabu é um procedimento adaptativo que guia um algoritmo de busca local na exploração contínua do espaço de busca, sem:
  - ser confundido pela ausência de vizinhos aprimorantes
  - retornar a um ótimo local previamente visitado (condição desejada, mas não necessária)
- Utiliza estruturas flexíveis de memória para armazenar conhecimento sobre o espaço de busca, contrariamente a algoritmos que:
  - não utilizam memória (e.g. *simulated annealing*)
  - utilizam estruturas rígidas de memória (e.g. *branch-and-bound*)

# Busca tabu

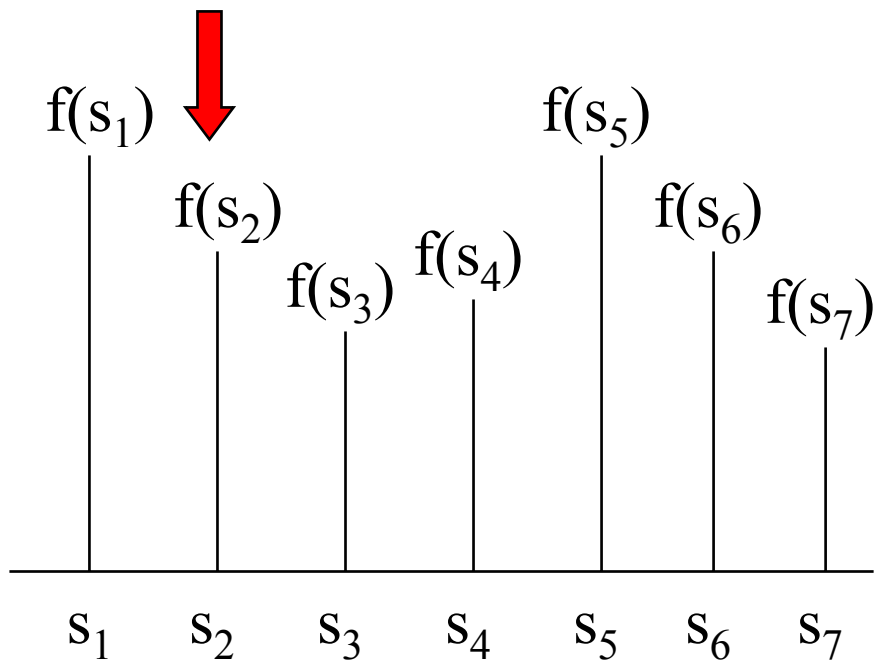
- Parte de uma solução inicial e, a cada iteração, desloca-se para a melhor solução na vizinhança:
  - Proibição de movimentos que levam a soluções já visitadas (lista tabu)
  - Solução tabu: se a melhor solução na vizinhança não é melhor do que a solução corrente (deterioração do valor da função de custo)
- Lista tabu: estrutura de memória básica, formada pelas soluções proibidas (tabu)
- Determinada por informações históricas da busca
- Soluções proibidas por um certo número de iterações (prazo tabu ou *tabu-tenure*)

# Busca tabu

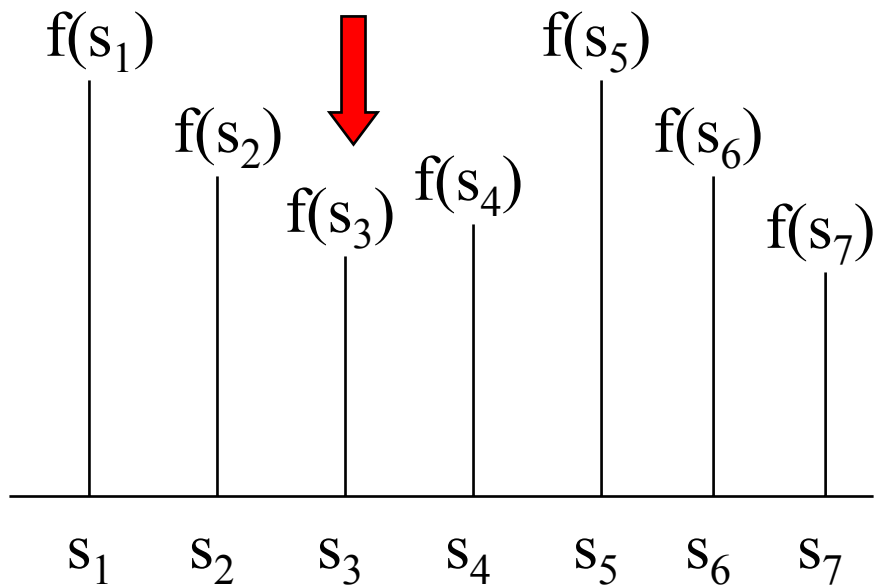




# Busca tabu



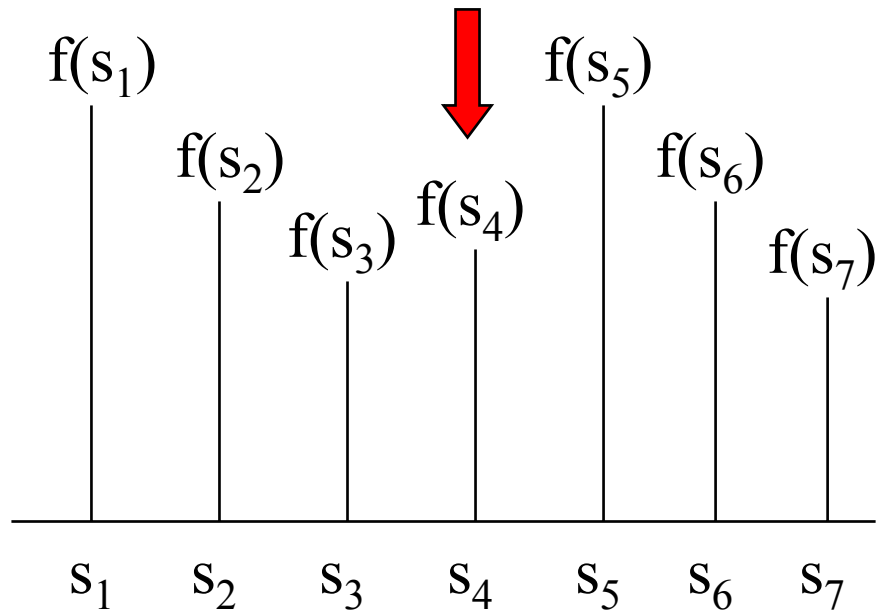
# Busca tabu



não há movimento aprimorante:  
 $s_4$  é a melhor solução vizinha

$s_3$  torna-se uma solução proibida

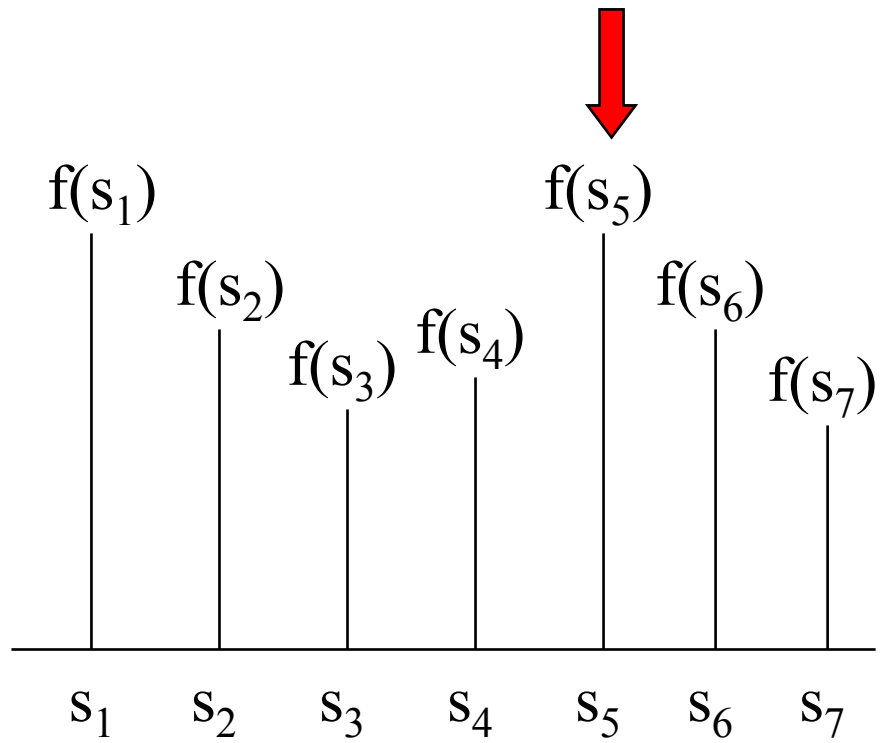
# Busca tabu



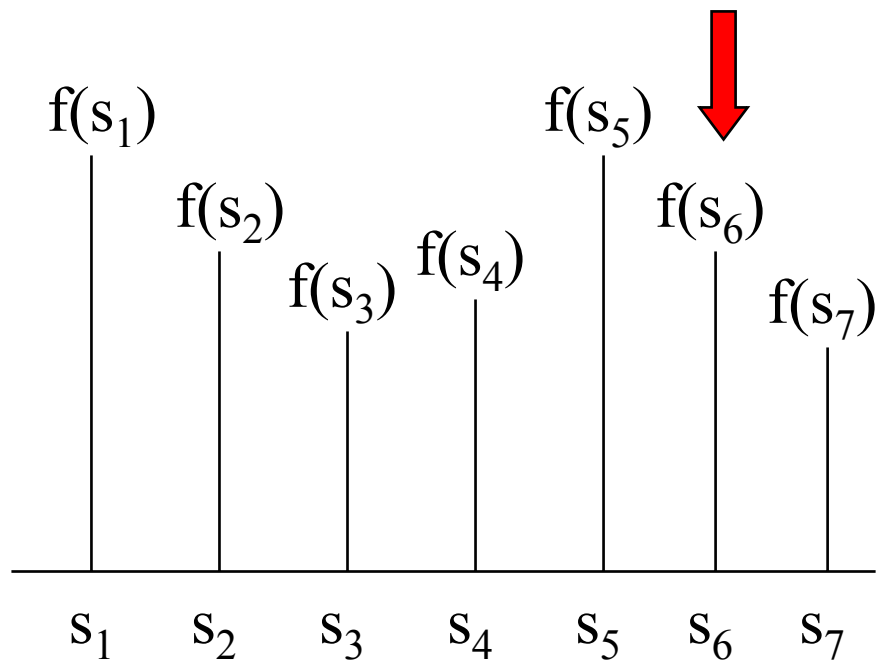
como  $s_3$  é uma solução proibida,  
a solução vizinha escolhida é  $s_5$

como  $s_5$  é pior do que a solução  
corrente  $s_4$ , esta torna-se uma  
solução proibida

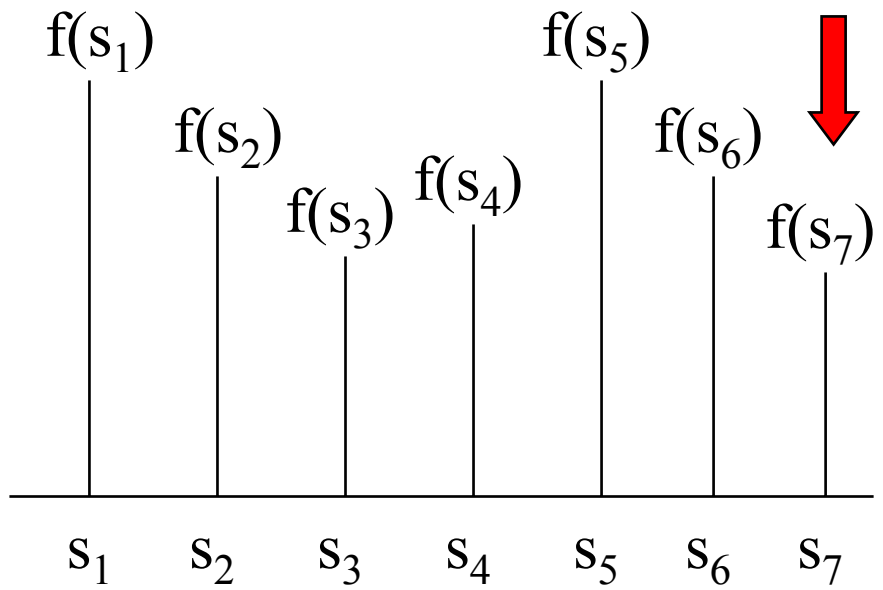
# Busca tabu



# Busca tabu



# Busca tabu



# Busca tabu

- Algoritmo básico:

$s \leftarrow s_0$ ;  $s^* \leftarrow s_0$ ;  $LT \leftarrow \emptyset$

**while** critério-de-parada **do**

    Obter a melhor solução  $s' \in N(s) \setminus LT$

**if**  $f(s') < f(s^*)$

**then**  $s^* \leftarrow s'$

**else**

**if**  $f(s') \geq f(s)$

**then**  $LT \leftarrow LT \cup \{s\}$

**end-if**

$s \leftarrow s'$

    Atualizar lista-tabu LT (prazos)

**end-while**

# Busca tabu

- Aspectos de implementação do algoritmo básico:
  - manter as soluções proibidas demanda muito espaço e tempo para armazenar e, mais ainda, para testar se uma dada solução está proibida.
  - armazenar movimentos proibidos permite superar os problemas acima. Entretanto, pode proibir soluções que ainda não foram visitadas.

- Exemplo: problema de otimização 0-1 com representação por vetor de pertinência

$$v = (v_1, \dots, v_i, \dots, v_n)$$

$$N(v) = \{(v_1, \dots, 1-v_i, \dots, v_n), i=1, \dots, n\}$$

$$\text{Lista tabu: vetor } t = (t_1, \dots, t_i, \dots, t_n)$$

$t_i$ : índice da iteração a partir da qual a direção  $i$  pode ser utilizada (i.e., a variável  $i$  pode ser novamente complementada).



# Busca tabu

- Solução corrente:  $s = (0,1,0)$

Melhor vizinho:  $s' = (0,1,1)$

Movimento:  $x_3 = 0 \rightarrow x_3 = 1$

Movimento reverso proibido:  $x_3 = 1 \rightarrow x_3 = 0$

Soluções proibidas em consequência:

$(0,0,0), (0,1,0), (1,0,0), (1,1,0)$

# Busca tabu

**procedure** Tabu-Perm-N1( $\pi_0$ )

$\pi^*, \pi \leftarrow \pi_0$ ; iteração  $\leftarrow 0$ ;  $t_i \leftarrow 0$ ,  $i=1, \dots, n-1$

**while** critério-de-parada **do**

iteração  $\leftarrow$  iteração + 1;  $f_{\min} \leftarrow +\infty$

**for**  $i = 1$  **to**  $n-1$  **do**

**if**  $t_i \leq$  iteração

**then**  $\pi' \leftarrow \pi$ ;  $\pi'_i \leftarrow \pi_{i+1}$ ;  $\pi'_{i+1} \leftarrow \pi_i$

**if**  $f(\pi') < f_{\min}$

**then**  $\pi_{\min} \leftarrow \pi'$ ;  $f_{\min} \leftarrow f(\pi')$ ;  $i_{\min} \leftarrow i$

**end-for**

**if**  $f_{\min} < f(\pi^*)$  **then**  $\pi^* \leftarrow \pi_{\min}$

**else**

**if**  $f_{\min} \geq f(\pi)$

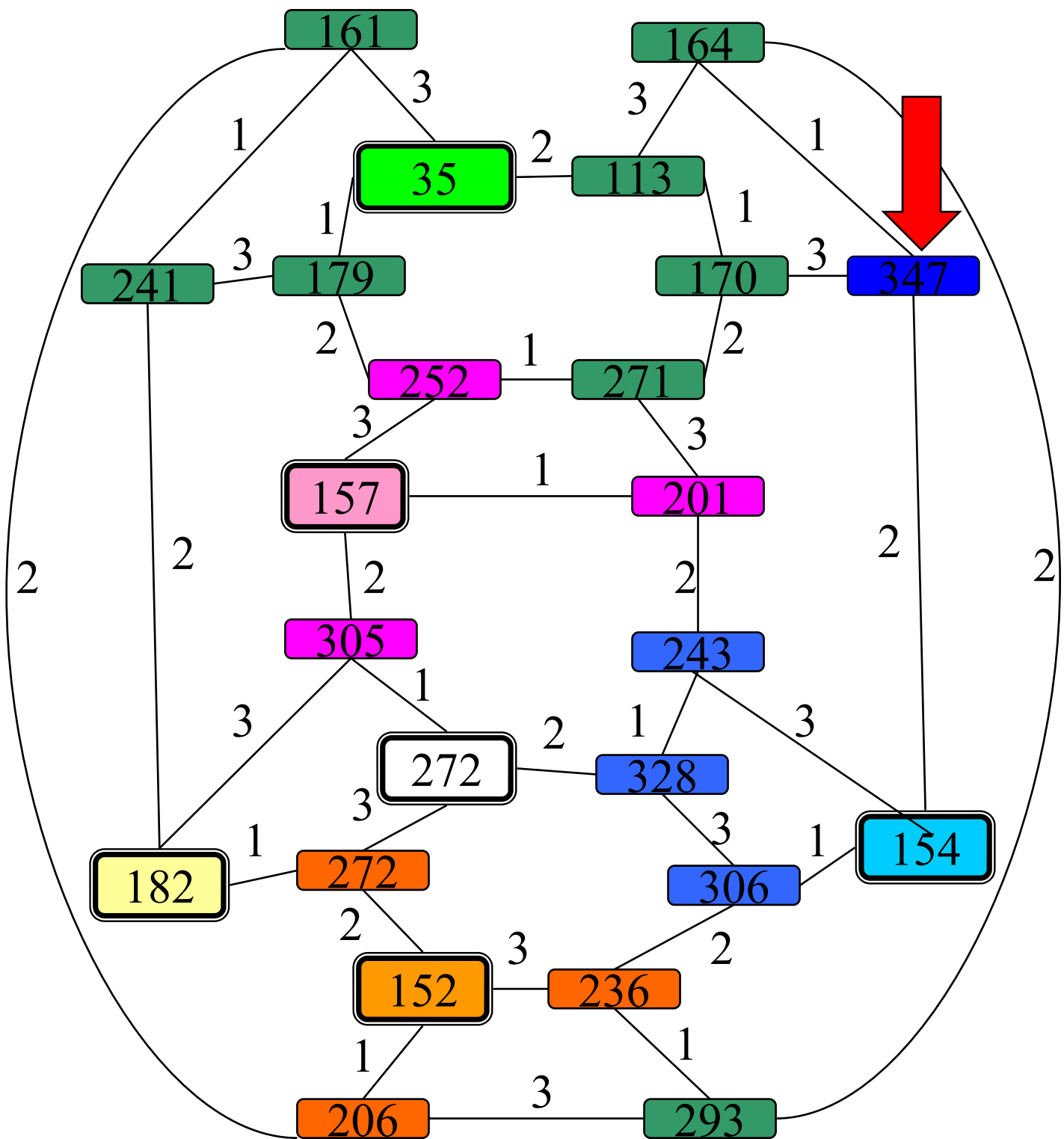
**then**  $t_{i_{\min}} \leftarrow$  iteração + prazo-tabu

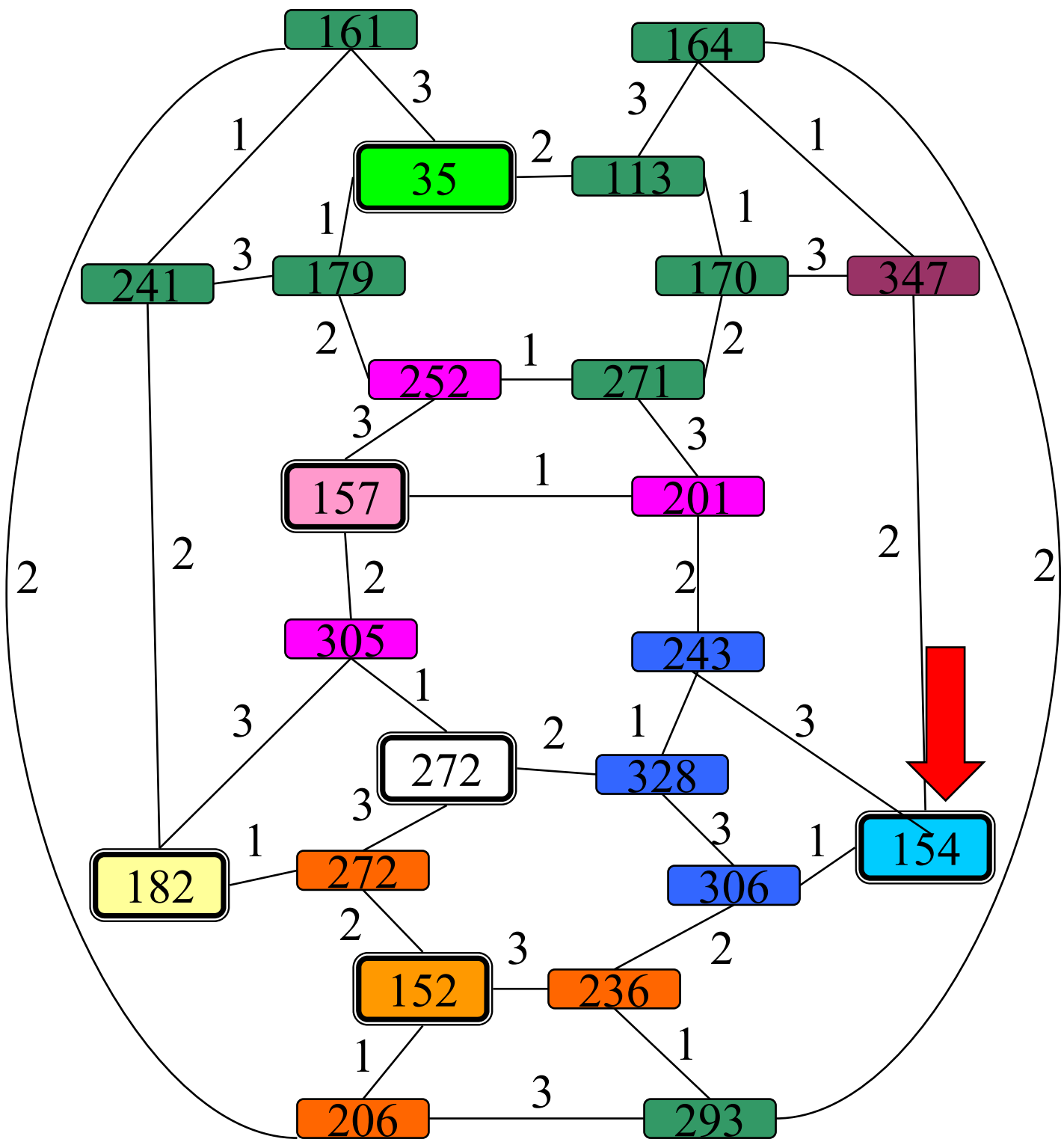
$\pi \leftarrow \pi_{\min}$

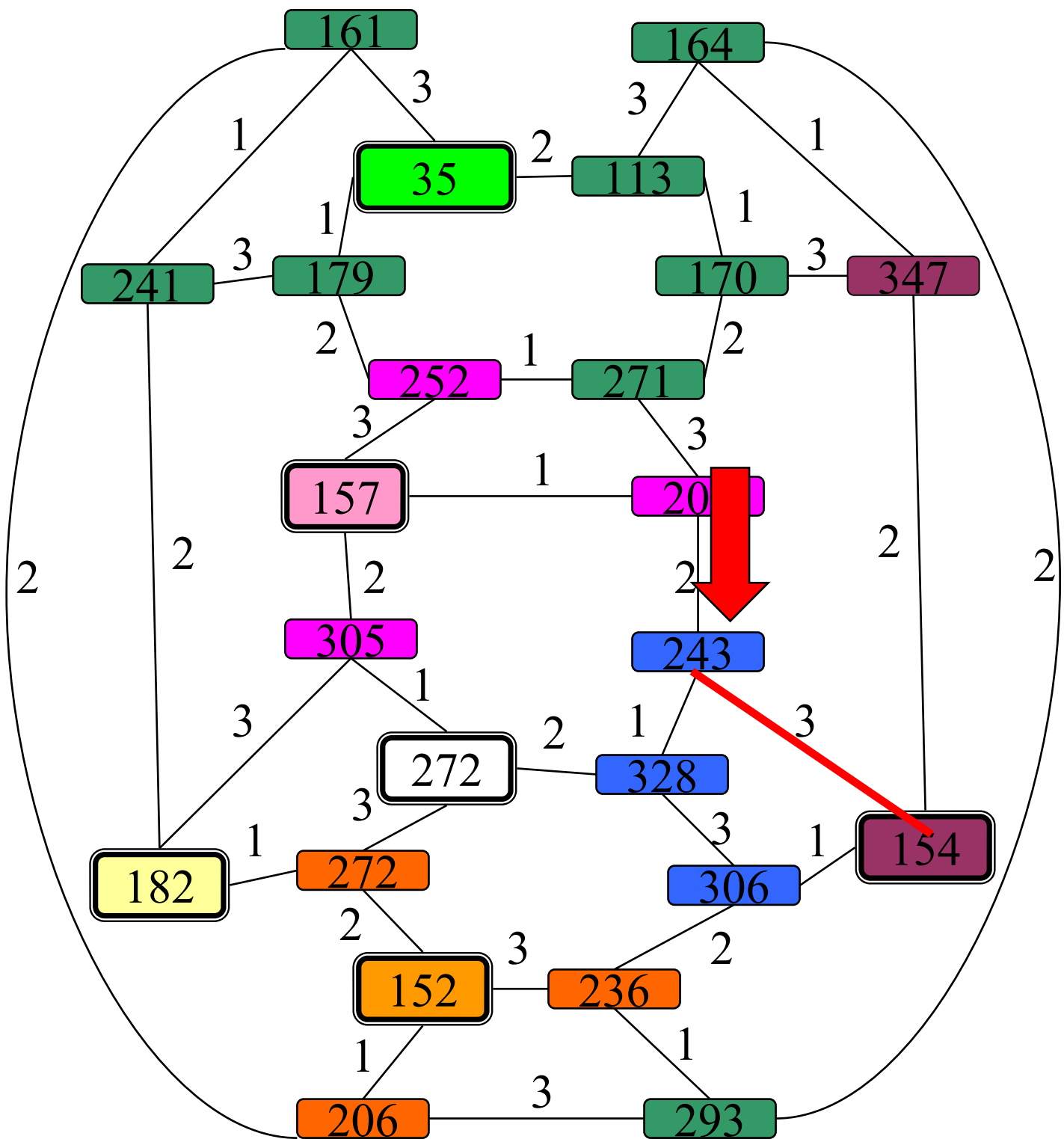
**end-while**

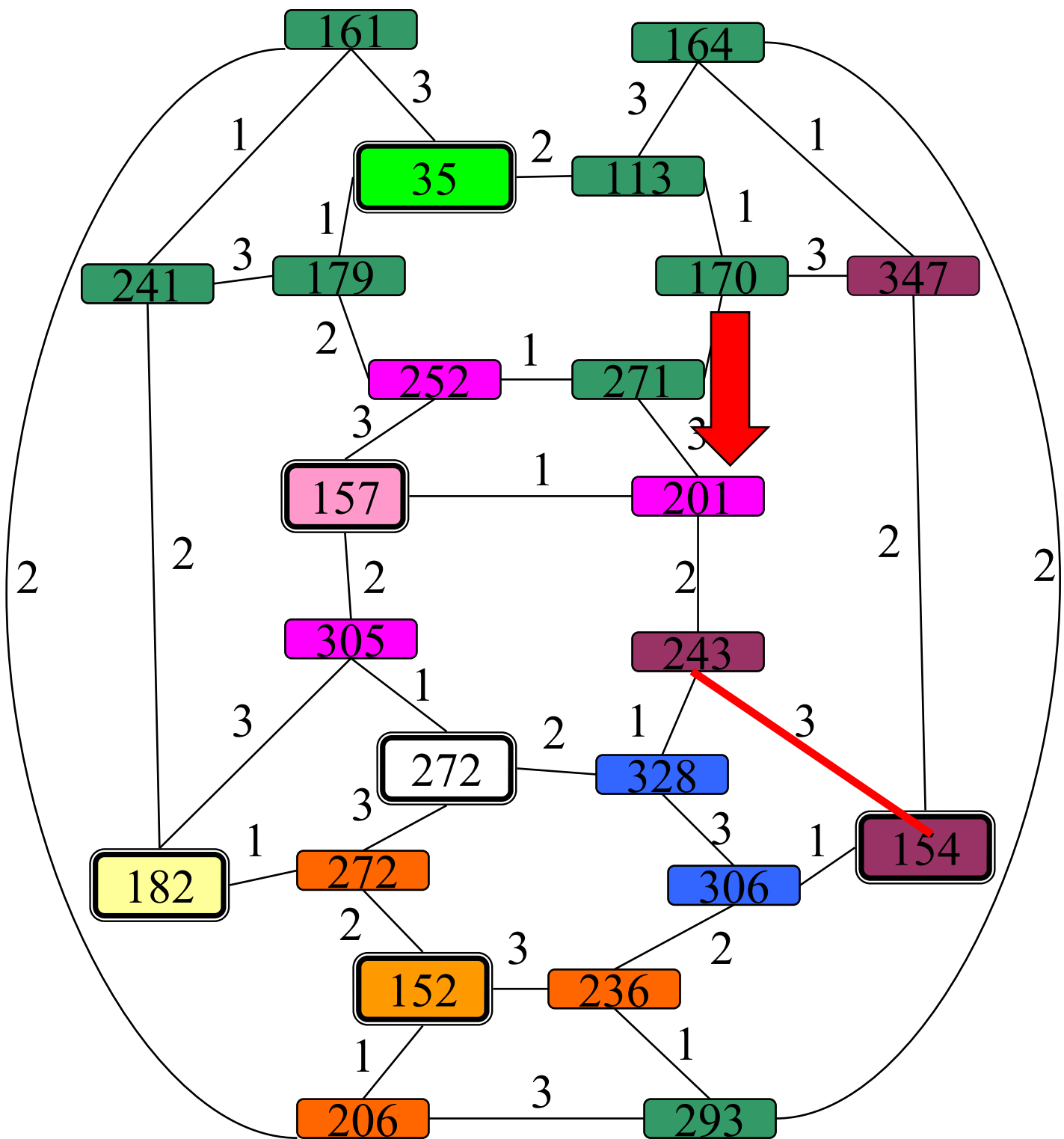
**return**  $\pi^*$

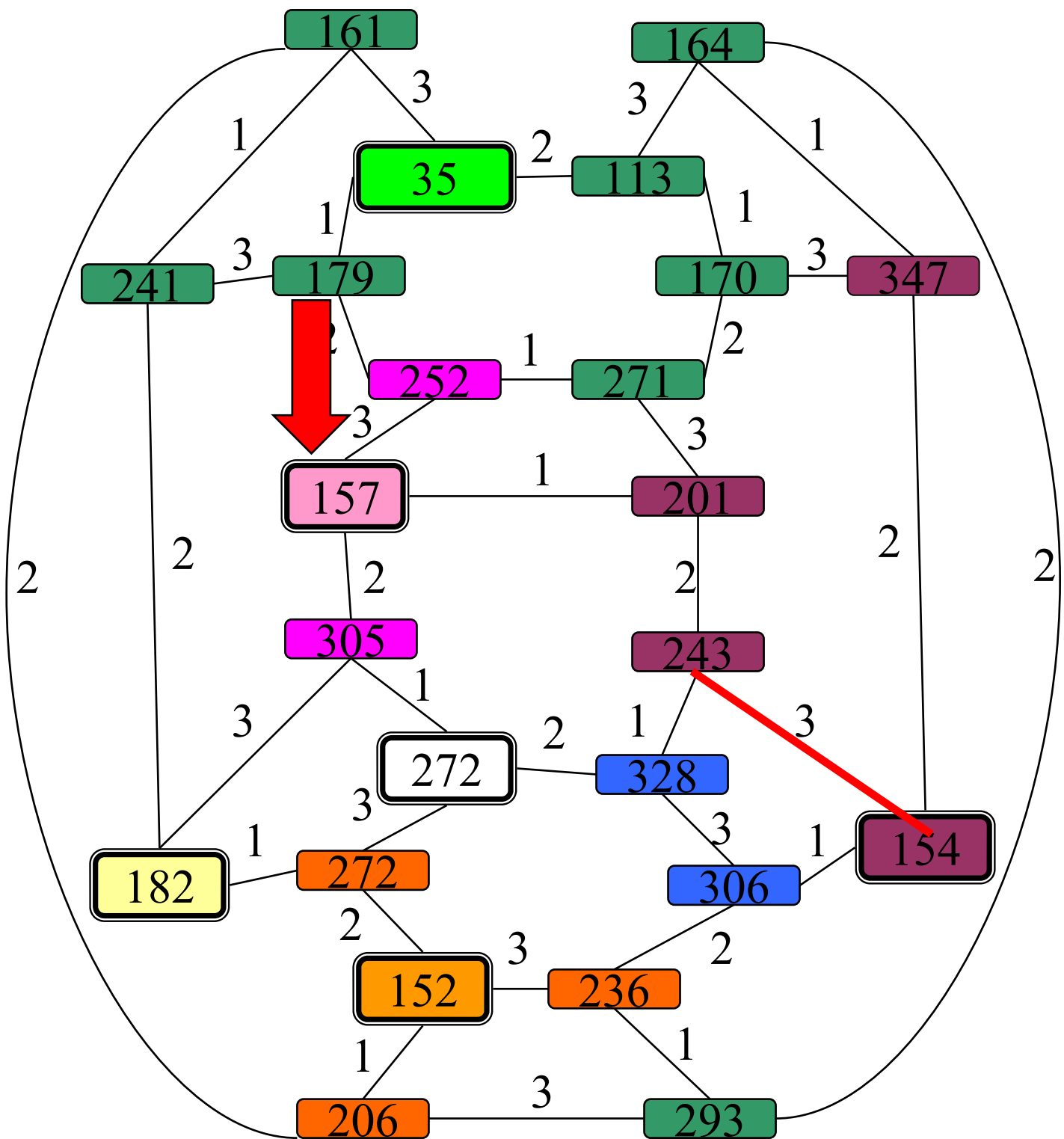
**end** Tabu-Perm-N1

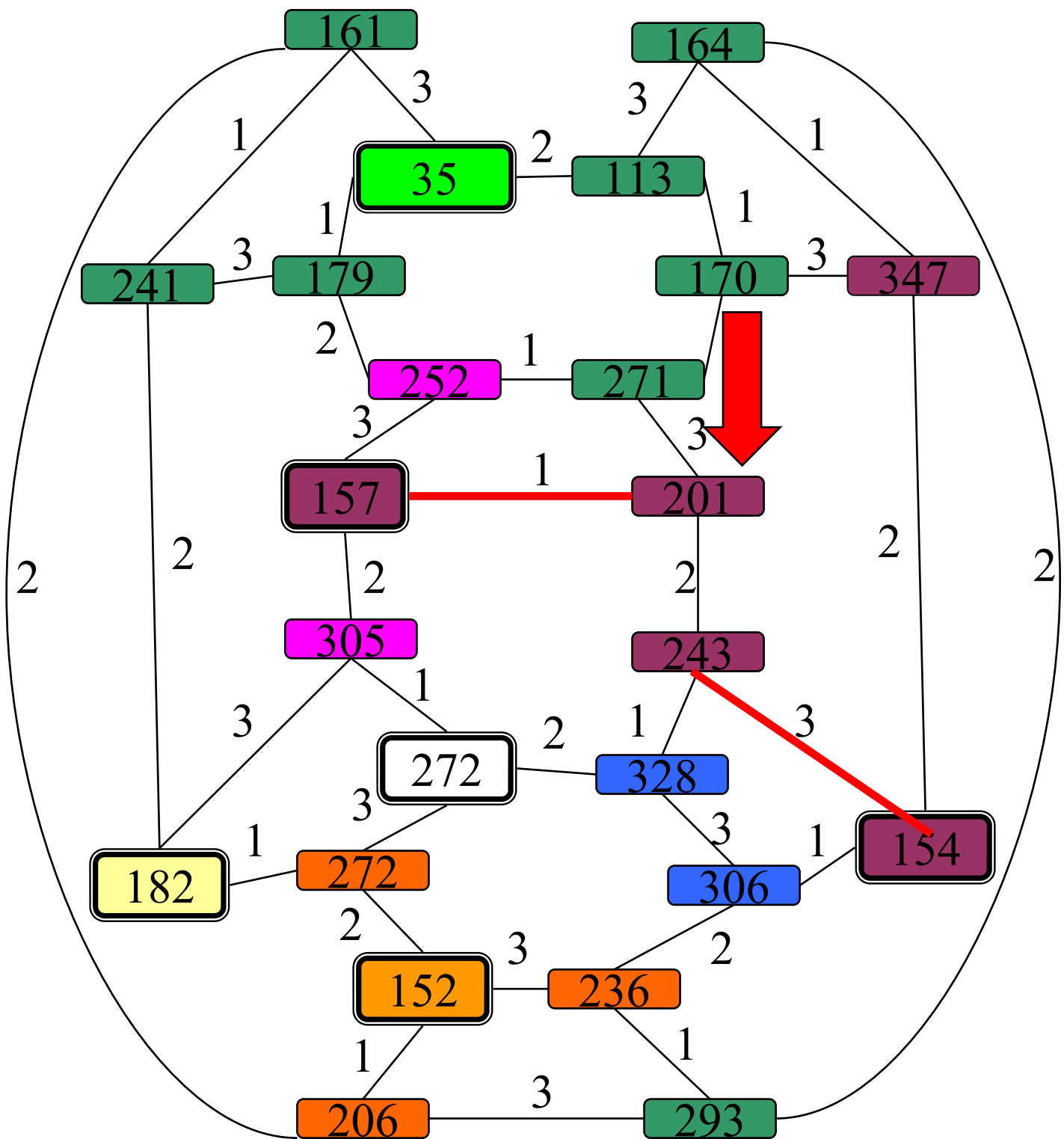














# Busca tabu

- Critério de aspiração: condição em que um movimento torna-se permitido apesar de estar na lista tabu. Exemplos:
  - quando a solução tabu passa a ser a melhor conhecida até o momento na busca
  - quando a solução tabu reduz significativamente o valor da solução corrente (buscar soluções num patamar mais promissor)
- Necessidade do critério de aspiração: vem da proibição de soluções ainda não visitadas, decorrentes de uma estrutura mais eficiente para o tratamento da lista tabu.
- Implementação de um critério de aspiração exige maior esforço computacional a cada iteração.

# Busca tabu

- Intensificação:

Concentrar a busca em regiões promissoras do espaço, em torno de boas soluções (soluções de elite), modificando as regras de escolha da próxima solução de modo a combinar bons movimentos com a estrutura destas soluções.

- Diversificação:

Fazer com que a busca visite regiões ainda inexploradas. Pode ser feito penalizando-se movimentos que voltam a utilizar elementos que estiveram freqüentemente presentes em soluções visitadas anteriormente e/ou incentivando a entrada de elementos pouco utilizados no passado.

# Busca tabu

- Estruturas de memória

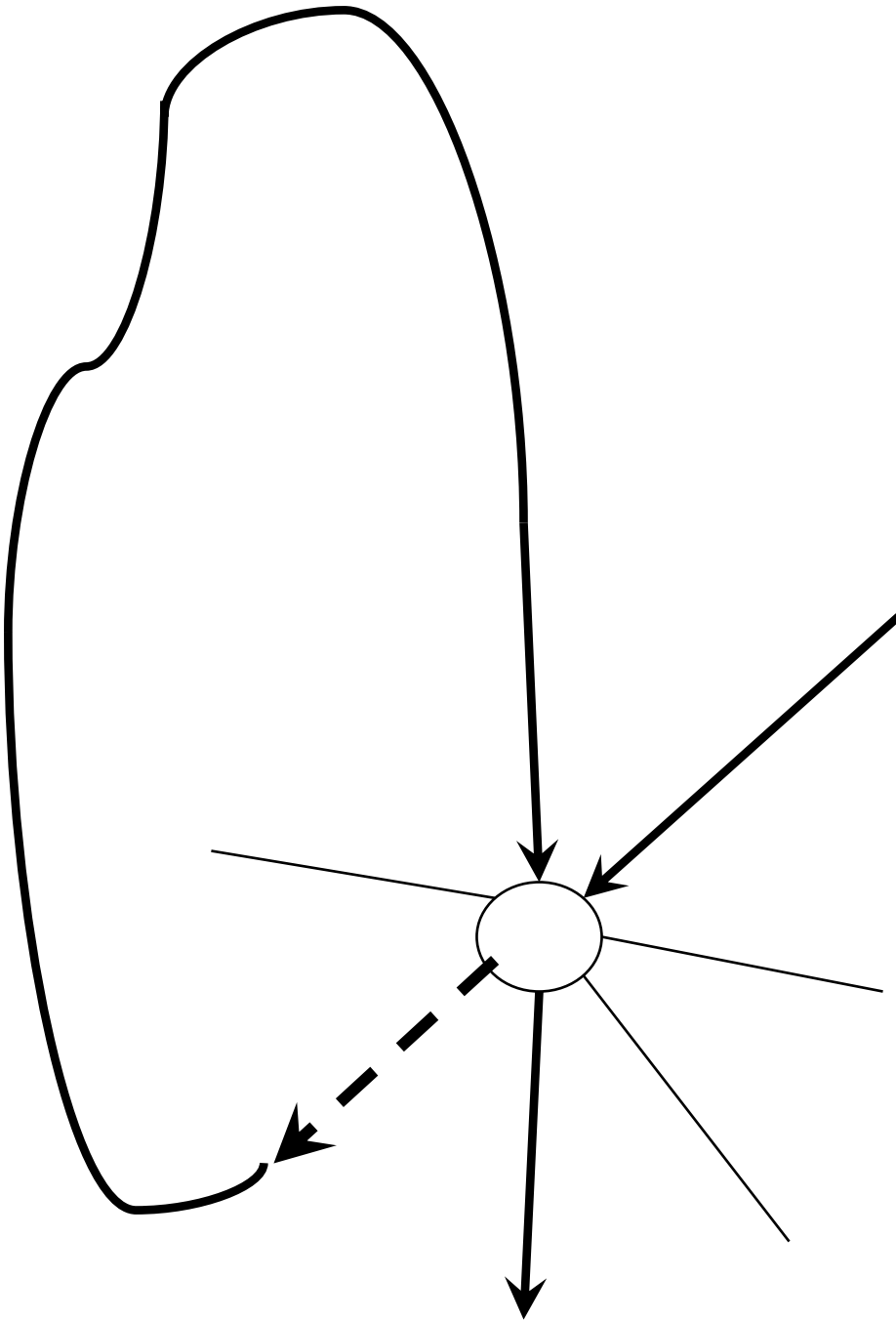
Memória de curto prazo:

- essencialmente, a lista de movimentos tabu
- objetivo básico: evitar ciclagem na busca tabu (retornar a uma mesma solução com lista tabu diferentes não é ciclar)

Memória de longo prazo:

- lista de soluções de elite
- frequência ou número de vezes em que um elemento da solução foi utilizado em soluções já visitadas anteriormente

# Busca tabu



# Busca tabu

- Intensificação:
  - reduzir o número de iterações em que um movimento é tabu (prazo-tabu): maior chance de ciclagem.
  - forçar a entrada de elementos freqüentes nas soluções de elite
  - considerar restrições sobre o espaço de busca, seja retirando as soluções que não as satisfazem, seja penalizando-as.
- Diversificação:
  - aumentar o prazo-tabu: tende a degradar demasiadamente as soluções encontradas
  - forçar a entrada de elementos com baixa frequência nas soluções visitadas: é importante também levar em conta a contribuição destes elementos na função de custo
  - eventualmente, retirar restrições da estrutura do problema.

# Busca tabu

- Oscilação estratégica: alternar entre intensificação e diversificação
- Executar movimentos até encontrar uma fronteira de viabilidade onde a busca normalmente seria interrompida. Em vez de parar, a definição de vizinhança é estendida ou o critério de seleção de movimentos é modificado, de modo a permitir que a fronteira de viabilidade seja cruzada. A busca prossegue então até uma determinada profundidade além da fronteira e retorna. A partir deste ponto a fronteira é novamente aproximada e cruzada, desta vez pela direção oposta. O processo de repetidamente aproximar e cruzar a fronteira em direções diferentes cria uma forma de oscilação, que dá o nome ao método

# Busca tabu

- Exemplo: problema multidimensional da mochila
  - (a) variáveis são modificadas de 0 para 1 até que seja atingida a fronteira de viabilidade
  - (b) a partir deste ponto, continuar a busca em direção à região inviável usando o mesmo tipo de movimentos, mas com um critério de avaliação diferente
  - (c) após determinado número de passos, reverter a direção de busca, passando a modificar variáveis de 1 para 0

# Busca tabu

- Exemplo: penalização da violação de restrições
  - em vez de considerar as restrições, incorporá-las à função objetivo através da penalização de sua violação
  - penalização definida por um coeficiente de penalidade associado a cada restrição
  - executar uma seqüência de  $K$  iterações com um determinado conjunto de valores dos coeficientes de penalidade
  - para cada tipo de restrição, executar o seguinte procedimento:
    - (a) se todas as soluções obtidas durante esta seqüência forem viáveis, reduzir o coeficiente de penalização (e.g., dividi-lo por 2)
    - (b) se todas as soluções obtidas durante esta seqüência forem inviáveis, aumentar o coeficiente de penalização (e.g., multiplicá-lo por 2)



# Busca tabu

- Utilização de valores diferentes (e.g., selecionados aleatoriamente dentro de uma faixa) para o parâmetro prazo-tabu reduz a possibilidade de ciclagem: lista tabu de tamanho variável.
- Prazos-tabu diferentes para tipos diferentes de movimentos: muitas vezes, as soluções viáveis usam apenas uma pequena fração dos elementos possíveis
  - prazos-tabu maiores para os elementos que saem das soluções, pois há mais opções de inserção do que de eliminação

# Busca tabu

- Exemplo: problema do caixeiro viajante  
solução viável:  $n$  arestas  
número de arestas ( $|E|$ ):  $n(n-1)/2$ 
  - utiliza-se um prazo-tabu maior para as arestas que saíram da solução, do que para aquelas que entraram na solução
- Exemplo: problema de Steiner em grafos  
solução viável: poucos vértices opcionais
  - utiliza-se um prazo-tabu maior para os nós que foram eliminados, do que para aqueles que foram inseridos
- Outra alternativa para lista tabu: proibir qualquer solução (ou qualquer movimento), e não apenas aquelas que deterioraram a solução corrente.

# Busca tabu

- Lista de movimentos candidatos: redução ou segmentação da vizinhança, visando acelerar as iterações e permitindo contemplar incentivos à aproximação de soluções de elite e regular a “agressividade” da busca.
  - amostragem aleatória da vizinhança
  - busca em seqüência de segmentos da vizinhança: executa o movimento aprimorante encontrado no primeiro segmento que contiver um; inclusão na lista tabu somente quando não houver movimento aprimorante em nenhum segmento
  - primeiro movimento aprimorante
  - lista de movimentos promissores que são re-examinados em cada iteração, até que todos tenham sido examinados (quando cada um deles terá sido efetuado ou terá deixado de ser aprimorante)

# Busca tabu

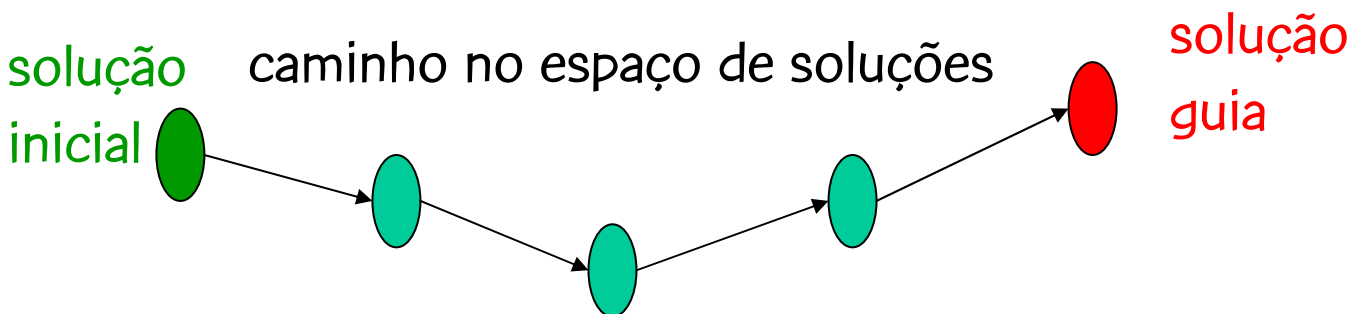
- Características:
  - uso intensivo de estruturas de memória adaptativas
  - requer mais engenharia para a sua implementação:
    - muitos parâmetros
    - muitas estruturas e possibilidades de condução da busca
  - soluções de boa qualidade: as melhores soluções encontradas para muitos problemas
  - depende pouco da solução inicial, pois possui mecanismos eficientes que permitem escapar de ótimos locais
  - robustez

# Busca tabu

- Path relinking: intensificação da busca no caminho que leva a soluções de elite
  - armazenar um conjunto de soluções de elite
  - para cada par de soluções de elite:
    - (a) identificar as diferenças entre elas
    - (b) executar um procedimento de busca local sem parar em nenhum ótimo local, executando sempre o melhor movimento dentre aqueles ainda não executados
    - (c) verificar a existência nesta trajetória de soluções melhores do que as extremidades
- Exemplo: problema de Steiner em grafos
  - vértices de Steiner que pertencem a uma solução mas não à outra (e vice-versa)

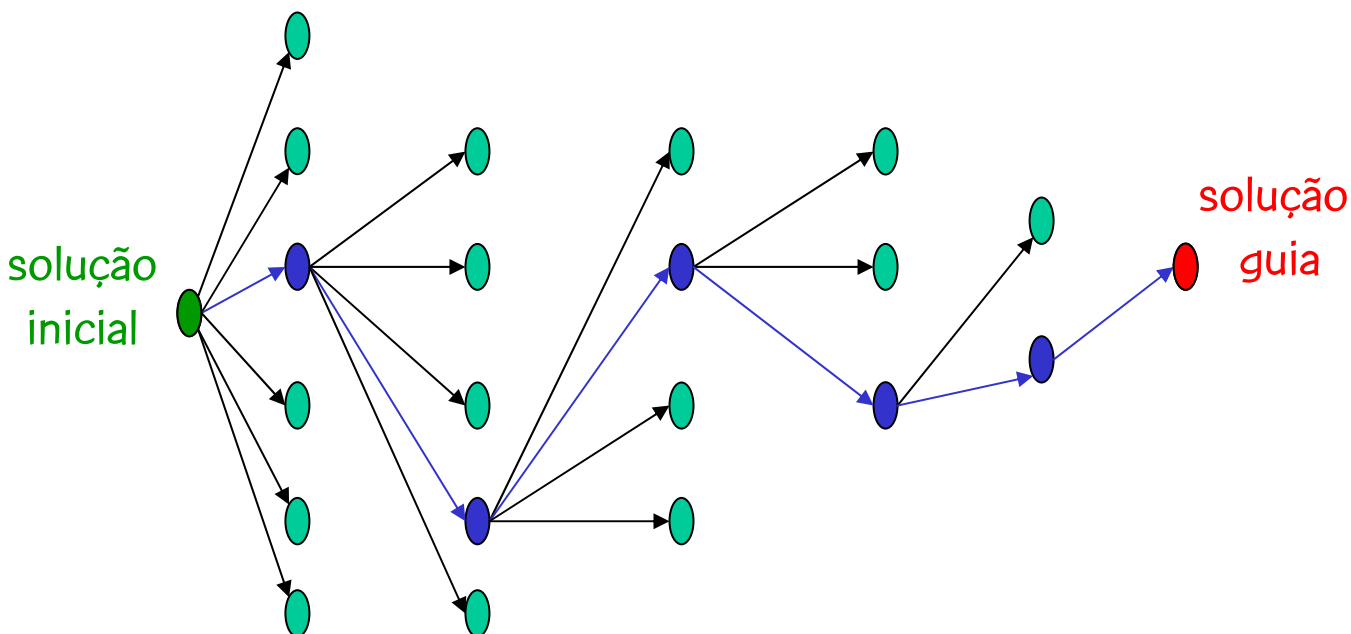
# Busca tabu

- Path-relinking:
  - Estratégia de intensificação para explorar trajetórias conectando soluções de elite
  - Originalmente proposta no contexto de busca tabu
  - Caminhos conectando soluções de elite ao longo do espaço de soluções são explorados na busca de melhores soluções:
    - Seleção de movimentos que introduzem atributos da solução guia na solução corrente



# Busca tabu

- Um caminho é gerado e explorado, por meio da seleção de movimentos que introduzem na **solução inicial** atributos da **solução guia**
- A cada passo, são avaliados todos os movimentos que incorporam atributos da solução guia e o melhor deles é selecionado:

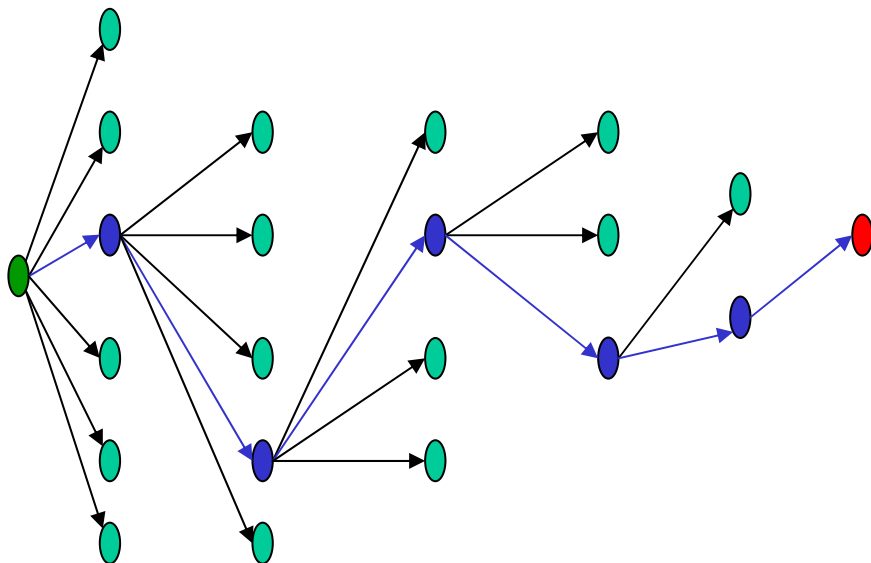


# Busca tabu

Soluções de elite  $x$  e  $y$

$\Delta(x,y)$ : diferença simétrica entre  $x$  e  $y$

```
while (  $|\Delta(x,y)| > 0$  ) {  
    avaliar os movimentos em  $\Delta(x,y)$   
    selecionar e realizar o melhor  
        movimento  
    atualizar  $\Delta(x,y)$   
}
```





# Busca tabu

- Paralelização:
  - Decomposição da vizinhança: permite acelerar cada iteração (estratégia síncrona)
    - particionamento simples
    - particionamento múltiplo (melhoria do balanceamento de carga)
  - Múltiplas buscas independentes (estratégia assíncrona)
  - Múltiplas buscas cooperativas usando memória compartilhada (estratégia assíncrona)
    - combinação de soluções iniciais diferentes
    - combinação de estratégias de busca diferentes
    - troca de informações sobre a busca através da memória

# Busca tabu

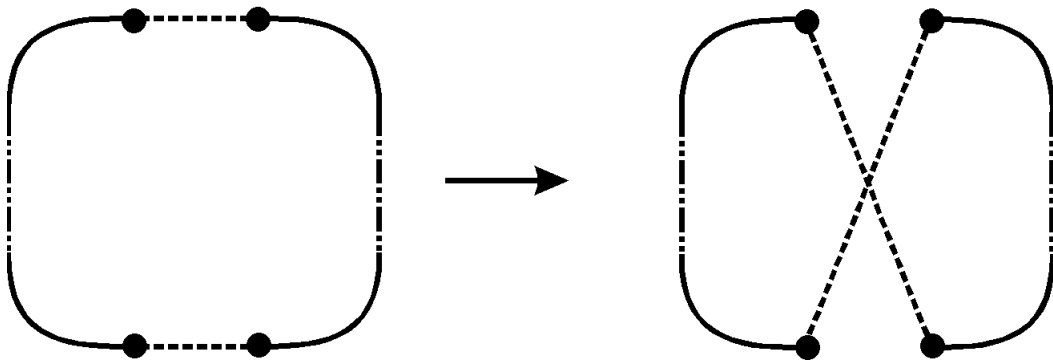
- Parâmetros:
  - Solução inicial
  - Tamanho da lista tabu (prazo tabu)
  - Critério de entrada na lista tabu
  - Critério de aspiração
  - Critério de parada

# Variable Neighborhood Search (VNS)

- Princípio: aplicar busca local amostrando vizinhos da solução corrente. Quando não for possível encontrar uma solução vizinha melhor, utilizar uma vizinhança diferente (“maior”).
- Conjunto de vizinhanças pré-selecionadas  
 $N = \{N_1, N_2, \dots, N_{\max}\}$   
 $N_k(s)$ : vizinhos de  $s$  na  $k$ -ésima vizinhança
- Exemplo: problema do caixeiro viajante  
 $N_1$ : 2-opt,  $N_2$ : 3-opt,  $N_3$ : 4-opt...  
mudar de  $k$ -opt para  $(k+1)$ -opt
- Exemplo: otimização 0-1  
 $N_k$ : complementar  $k$  variáveis da solução corrente

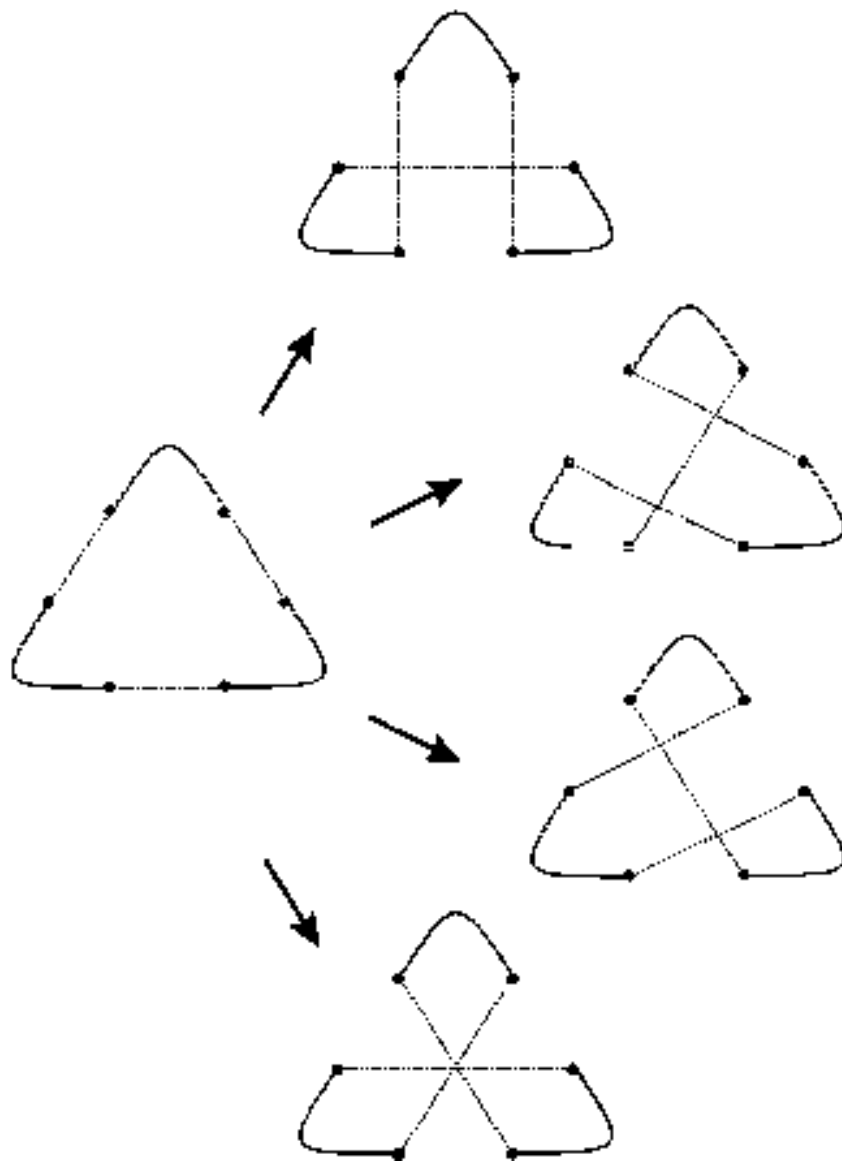
# VNS

- Vizinhaça 2-opt para o problema do caixeiro viajante:



# VNS

- Vizinhaça 3-opt para o problema do caixeiro viajante:



# VNS

- Algoritmo básico:

$s \leftarrow s_0$

**while** (critério de parada) **do**

$k \leftarrow 1$

**while** ( $k \leq \text{max}$ ) **do**

        Gerar aleatoriamente uma solução  
         $s'$  pertencente à vizinhança  
         $N_k(s)$

        Aplicar busca local a partir de  $s'$ ,  
        obtendo a solução  $s''$

**if**  $f(s'') < f(s)$

**then**

$s \leftarrow s''$

$k \leftarrow 1$

**else**  $k \leftarrow k + 1$

**end-if**

**end-while**

**end-while**

# VNS

- Critérios de parada:
  - tempo máximo de processamento
  - número máximo de iterações
  - número máximo de iterações sem melhoria
- Em geral, vizinhanças aninhadas
- GRASP: esforço de aleatorização apenas na construção de soluções
- VNS: aleatorização empregada na geração da solução vizinha (exploração da vizinhança), como forma de escapar de ótimos locais

# Variable Neighborhood Descent (VND)

- Princípio: aplicar a mudança de vizinhanças durante a busca local.

- Conjunto de vizinhanças pré-selecionadas

$$N = \{N_1, N_2, \dots, N_{\max}\}$$

$N_k(s)$ : vizinhos de  $s$  na  $k$ -ésima vizinhança



# VND

- Algoritmo básico:

$s \leftarrow s_0$

melhoria  $\leftarrow$  .verdadeiro.

**while** (melhoria) **do**

$k \leftarrow 1$

    melhoria  $\leftarrow$  .falso.

**while** ( $k \leq \text{max}$ ) **do**

        Aplicar busca local a partir de  $s$   
        utilizando a vizinhança  
         $N_k(s)$ , obtendo a solução  $s'$

**if**  $f(s') < f(s)$

**then**

$s \leftarrow s'$

            melhoria  $\leftarrow$  .verdadeiro.

**else**  $k \leftarrow k + 1$

**end-if**

**end-while**

**end-while**



# Greedy Randomized Adaptive Search Procedures (GRASP)

- Princípio: combinação de um método construtivo com busca local, em um procedimento iterativo com iterações completamente independentes
  - (a) construção de uma solução
  - (b) busca local
- Algoritmo básico:

$f(s^*) \leftarrow +\infty$

**for**  $i = 1, \dots, N$  **do**

    Construir uma solução  $s$  usando um algoritmo guloso aleatorizado

    Aplicar um procedimento de busca local a partir de  $s$ , obtendo a solução  $s'$

**if**  $f(s') < f(s^*)$  **then**  $s^* \leftarrow s'$

**end-for**

# GRASP

- Fase de construção: construir uma solução viável, um elemento por vez
- Cada iteração da fase de construção:
  - usando uma função gulosa, avaliar o benefício de cada elemento
  - criar uma lista restrita de candidatos, formada pelos elementos de melhor avaliação
  - selecionar aleatoriamente um elemento da lista restrita de candidatos
  - adaptar a função gulosa com base na decisão do elemento que foi incluído
- A escolha do elemento a ser incluído na próxima etapa do método construtivo é feita colocando-se os elementos ainda não escolhidos em uma lista ordenada de acordo com a função gulosa.

# GRASP

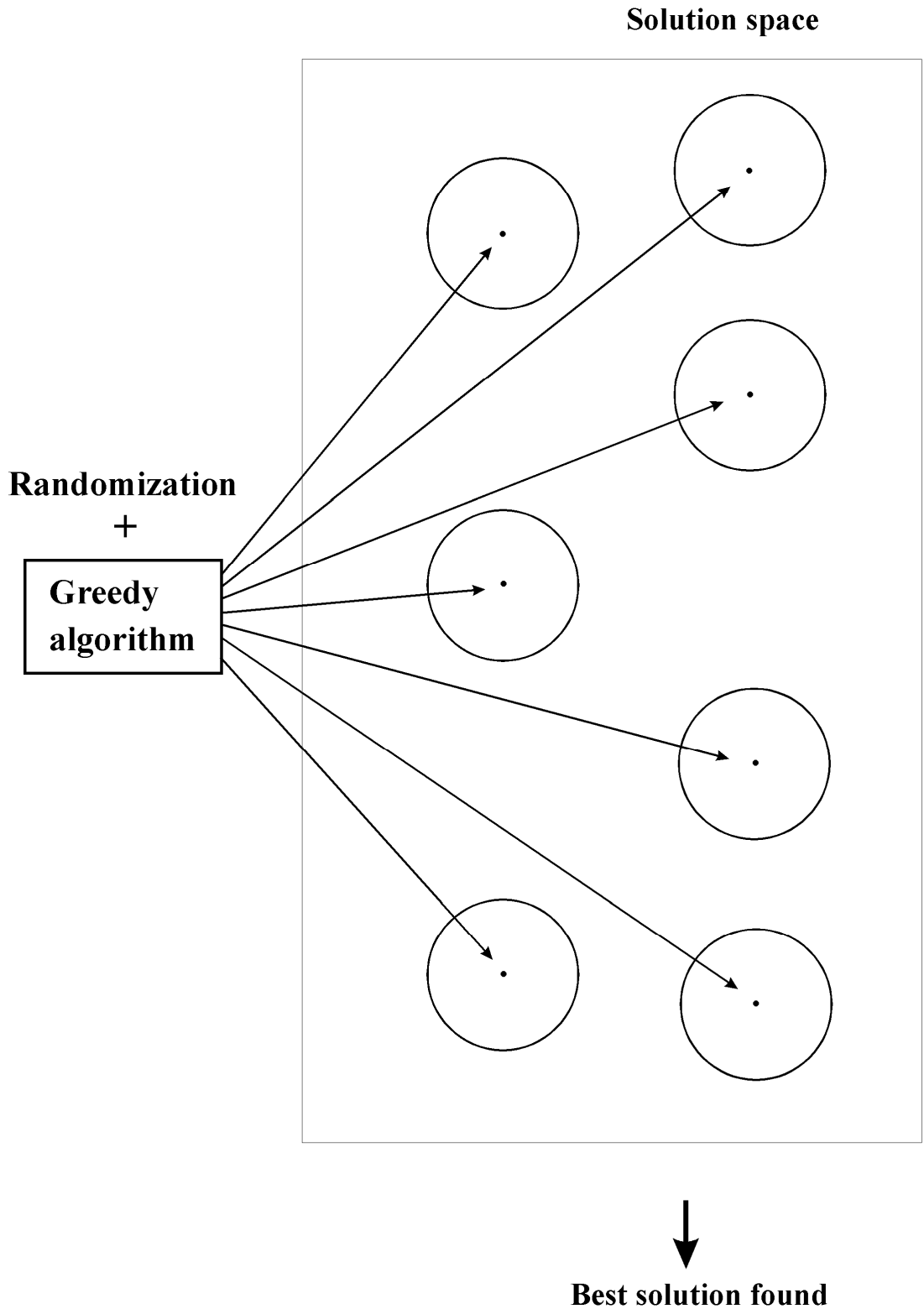
- Restrição dos elementos na lista de candidatos baseada em:
  - número máximo de elementos na lista
  - qualidade dos elementos na lista (em relação à escolha puramente gulosa)
- Seleção aleatória é feita entre os melhores elementos da lista de candidatos (não necessariamente o melhor, como na escolha puramente gulosa)
  - qualidade média da solução depende da qualidade dos elementos na lista
  - diversidade das soluções construídas depende da cardinalidade da lista
- Diversificação baseada em aleatorização controlada: diferentes soluções construídas em diferentes iterações GRASP

# GRASP

- Construção gulosa: boas soluções (próximas a ótimos locais), acelerando a busca local
- Busca local: melhorar as soluções construídas
  - escolha da vizinhança
  - estruturas de dados eficientes para acelerar a busca local
  - boas soluções iniciais permitem acelerar a busca local
- Utilização de um algoritmo guloso aleatorizado na fase de construção permite acelerar muito cada aplicação de busca local
- Comprovadamente mais rápido e encontra soluções melhores do que multi-partida simples

# GRASP

- Técnica de amostragem no espaço de busca



# GRASP

- Qualidade da solução encontrada: técnica de amostragem repetitiva no espaço de busca
  - cada iteração GRASP age como se estivesse obtendo uma amostra de uma distribuição desconhecida
  - média e variância da amostragem dependem das restrições impostas na criação da lista restrita de candidatos
- $|\text{LRC}| = 1$ 
  - algoritmo puramente guloso encontra a mesma solução em todas iterações
  - média = custo da solução gulosa
  - variância = 0
- $|\text{LRC}|$  aumenta
  - qualidade da solução diminui, devido a escolhas gulosas ruins
  - média < custo da solução gulosa
  - variância aumenta
  - possível melhorar a solução gulosa

# GRASP

- Exemplo:  
MAXSAT, 1391 variáveis, 3026 cláusulas  
100000 iterações

LRC	pior	vezes	melhor	vezes
1	3024	100000	3024	100000
4	3021	75	3025	2
16	3019	16	3026	25
64	3017	4	3026	163
256	3016	1	3026	689

LRC	média
1	3024.00
4	3023.79
16	3023.27
64	3023.00
256	3022.89



# GRASP

- Exemplo:  
MAXSAT, 1391 variáveis, 3026 cláusulas  
100000 iterações

LRC	3021	3022	3023	3024	3025
1				100000	
2		151	6053	93796	
4	75	1676	17744	80503	2



# GRASP

- Implementação simples:  
algoritmo guloso e busca local
- Heurísticas gulosas são simples de projetar e implementar
- Poucos parâmetros a serem ajustados
  - restritividade da lista de candidatos
  - número de iterações
- Depende de boas soluções iniciais: baseado apenas em aleatorização de uma iteração para outra, cada iteração se beneficia da qualidade da solução inicial
- Desvantagem da versão original: não utilizava memória das informações coletadas durante a busca

# GRASP

- Uso de filtros: aplicar busca local apenas
  - à melhor solução construída ao longo de uma seqüência de aplicações do algoritmo guloso aleatorizado
  - às soluções construídas que satisfazem um determinado limiar de aceitação
- Parâmetro  $\alpha$  para controlar a qualidade dos elementos da lista restrita de candidatos (caso de minimização)
  - $c_{\min}$  menor elemento pendente (guloso)
  - $c_{\max}$  maior elemento pendente (pior)
  - $$\text{LRC} = \{e \in E \setminus S: c_e \leq c_{\min} + \alpha \cdot (c_{\max} - c_{\min})\}$$
  - $\alpha = 0$ : guloso puro
  - $\alpha = 1$ : puramente aleatorio

# GRASP

- Ajuste do parâmetro  $\alpha$ : valores maiores de  $\alpha$  tendem a aumentar a diversidade das soluções geradas, a diminuir sua qualidade média e a aumentar os tempos de processamento
  - fixo: em geral, próximo ao guloso para garantir qualidade média, mas suficientemente grande para gerar diversidade
  - selecionado de forma aleatoria no intervalo  $[0,1]$  (discretizado ou contínuo)
  - auto-ajustável (reativo): inicialmente, como acima, de forma discretizada. Após um certo número de iterações, reavalia-se periodicamente a qualidade (média, melhor) das soluções obtidas com cada valor de  $\alpha$  e aumenta-se a probabilidade de seleção de valores de  $\alpha$  que estejam permitindo a obtenção de melhores soluções.



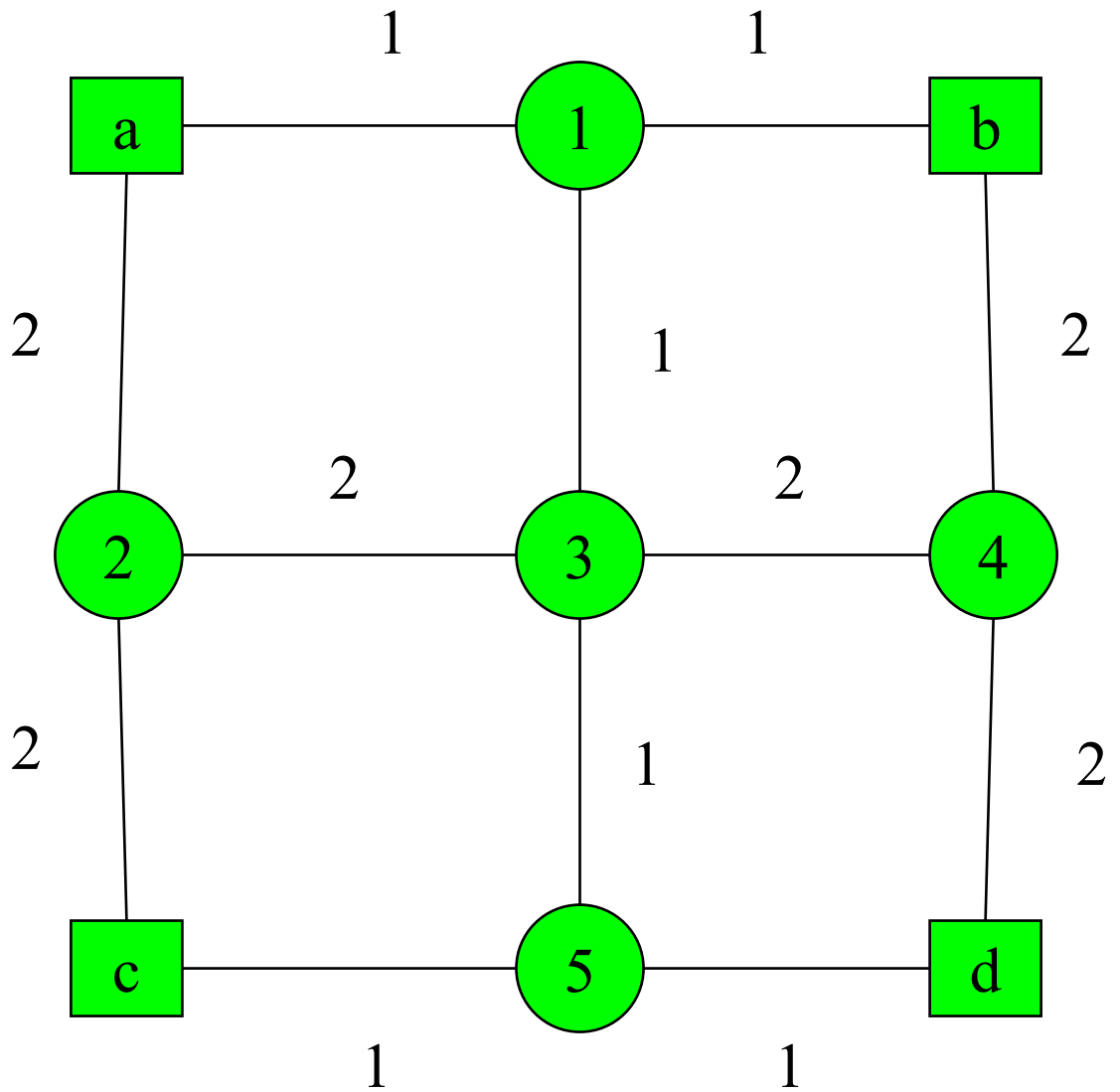
# GRASP

- Paralelização: simples e direta
- N iterações, p processadores: cada processador executa  $N/p$  iterações GRASP
  - pouca comunicação: ao final, cada processador informa sua melhor solução
  - Problema: desbalanceamento de carga
- Implementação mestre-escravo com distribuição de tarefas sob demanda:
  - dividir as iterações em  $p \cdot q$  blocos de  $(N/p)/q$  iterações
  - inicialmente, atribuir um bloco de  $N/p$  iterações a cada processador
  - à medida em que cada processador termina um bloco de iterações, solicita outro bloco ao mestre
  - mestre coordena a distribuição de blocos de iterações e a melhor solução
  - Resultados sensivelmente melhores
- Implementações colaborativas via conjunto de soluções de elite

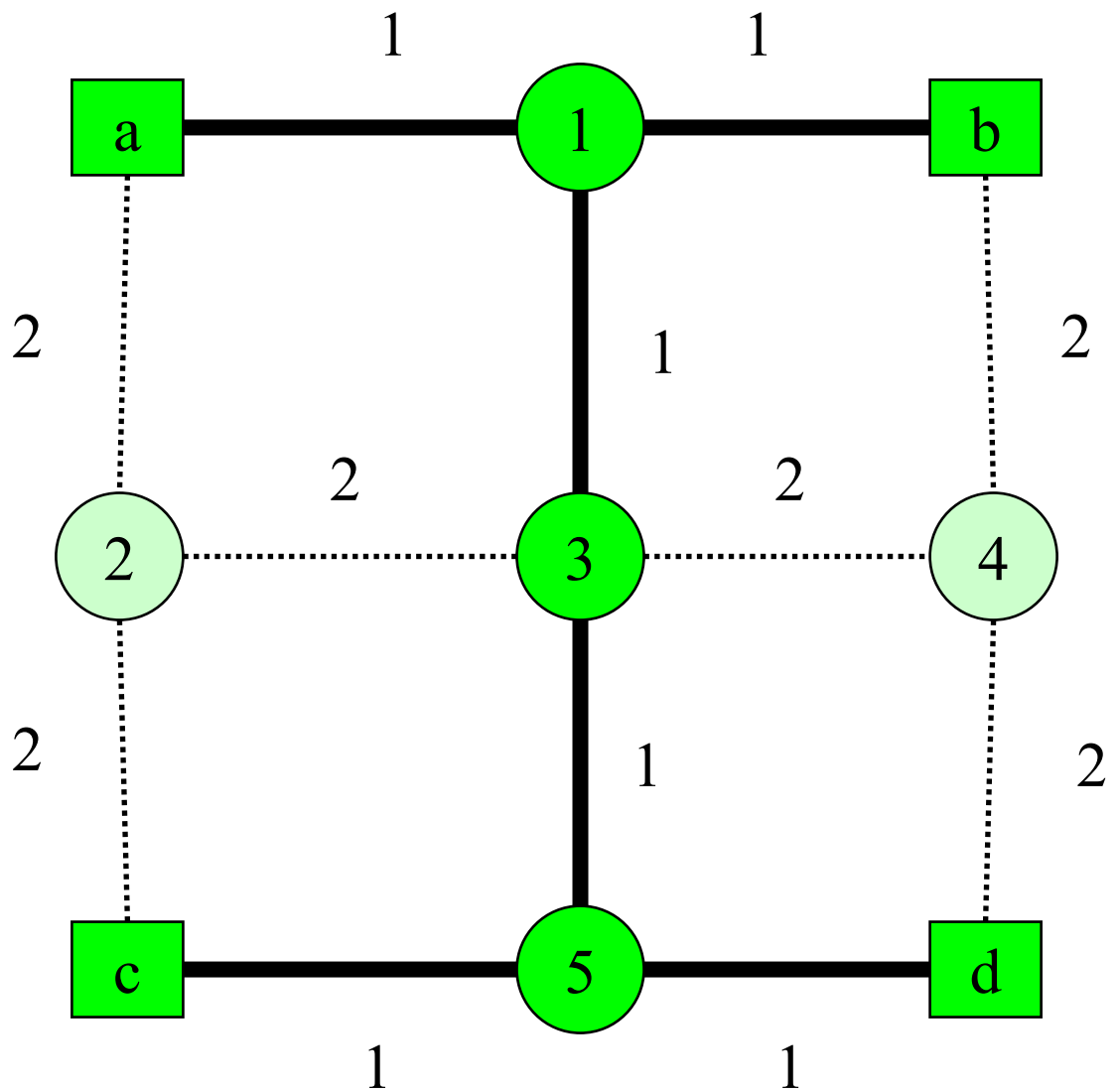
# Problema de Steiner em grafos

- Problema de Steiner em grafos:  
grafo não-orientado  $G=(V,E)$   
 $V$ : vértices  
 $E$ : arestas  
 $T$ : vértices terminais (obrigatórios)  
 $c_e$ : peso da aresta  $e \in E$
- Determinar uma árvore geradora dos vértices terminais com peso mínimo  
(caso particular: se  $T = V$ , problema da árvore geradora de peso mínimo)
- Vértices de Steiner: vértices opcionais que fazem parte da solução ótima
- Aplicações: projeto de redes de computadores, redes de telecomunicações, problema da filogenia em biologia

# Problema de Steiner em grafos



# Problema de Steiner em grafos





# Problema de Steiner em grafos

- Caracterização de uma solução  
 $X \subseteq V$ : subconjunto dos vértices opcionais

- Árvore de Steiner



Árvore geradora conectando os nós terminais (obrigatórios) usando um subconjunto  $X$  de vértices opcionais

- Cada árvore de Steiner pode então ser caracterizada pelo conjunto de nós opcionais utilizados

- Árvore de Steiner ótima



Árvore geradora de peso mínimo conectando os nós terminais (obrigatórios) e usando o subconjunto ótimo  $X^*$  de vértices opcionais

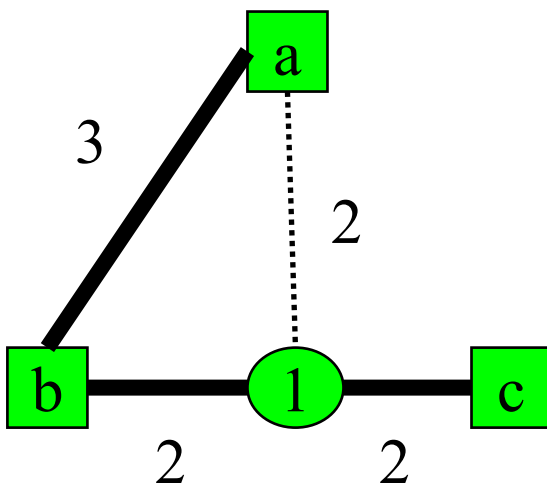
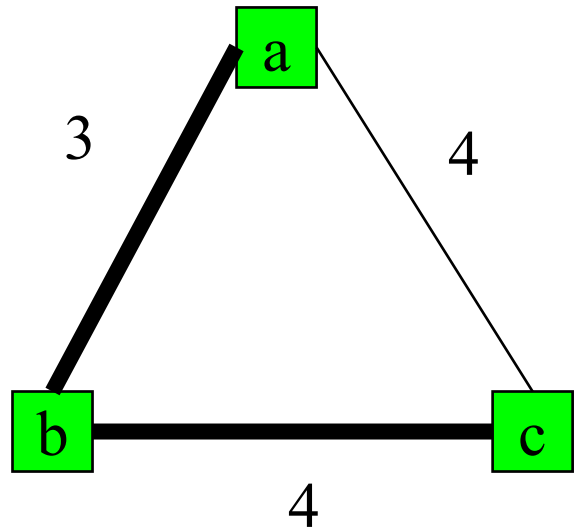
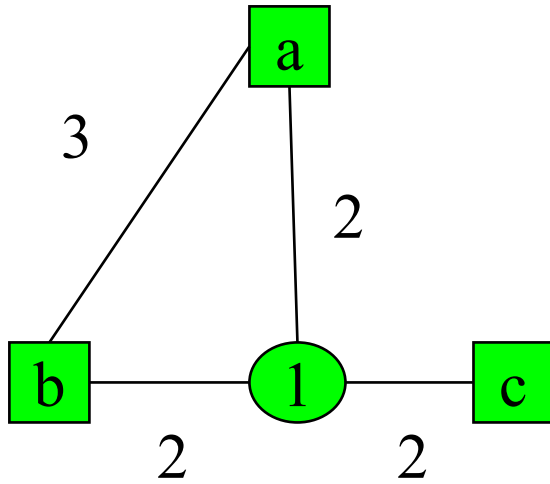
# Problema de Steiner em grafos

- $|V| = p$
- Solução:  $s = (s_1, s_2, \dots, s_i, \dots, s_p) \Leftrightarrow X$
- Representação por um indicador 0-1 de pertinência
- $s_i = 1$ , se o  $i$ -ésimo vértice opcional é selecionado, isto é, se  $v_i \in X$   
 $s_i = 0$ , caso contrário
- Vizinhança: todas as soluções que podem ser alcançadas inserindo-se ou eliminando-se um vértice opcional da solução corrente
- Movimentos de inserção
- Movimentos de eliminação

# Problema de Steiner em grafos

- GRASP:
  - fase de construção
  - busca local
- Fase de construção: baseada na aleatorização da heurística de distâncias
  - (a) construir uma rede cujos nós são apenas os nós obrigatórios, existe uma aresta entre cada par de nós obrigatórios e o peso de cada aresta é igual ao comprimento do caminho mais curto entre os dois nós obrigatórios correspondentes no grafo original
  - (b) obter uma árvore geradora de peso mínimo desta rede, usando o algoritmo guloso de Kruskal aleatorizado
  - (c) desmembrar as arestas da solução em caminhos do grafo original, obtendo uma árvore de Steiner

# Problema de Steiner em grafos



# Problema de Steiner em grafos

- Busca local:
  - (a) avaliar o movimento de inserção de cada nó opcional que não faz parte da solução corrente
  - (b) selecionar o melhor movimento de inserção e, se for aprimorante, atualizar a solução corrente e retornar a (a)
  - (c) avaliar o movimento de eliminação de cada nó opcional que faz parte da solução corrente
  - (d) selecionar o melhor movimento de eliminação e, se for aprimorante, alterar a solução corrente e retornar a (a)
  - (e) caso não existam movimentos aprimorantes, fim da busca

# Problema de Steiner em grafos

- Avaliação dos movimentos de inserção é bem mais rápida (atualização da solução corrente): movimentos de eliminação só são avaliados caso não existam movimentos de inserção aprimorantes
- Aceleração por lista de candidatos:
  - Após ter avaliado todos os movimentos de determinado tipo e ter selecionado o melhor, manter uma lista na memória com todos os movimentos aprimorantes.
  - Próxima iteração da busca: reavaliar apenas os movimentos que fazem parte desta lista, selecionando o primeiro que continuar sendo aprimorante.
  - Eliminar da lista tanto os movimentos que deixam de ser aprimorantes quanto os selecionados.
  - Reavaliar todos os movimentos quando a lista tornar-se vazia.

# Problema de Steiner em grafos

- Resultados computacionais:
- 60 problemas-teste da OR-Library
- Soluções ótimas: 47 problemas
  - Distância 1: quatro problemas
  - Distância 2: seis problemas
- Maior erro relativo: 2 %
  - Erro relativo maior que 1 %: 5 problemas
- Melhor solução encontrada na primeira iteração: 37 problemas



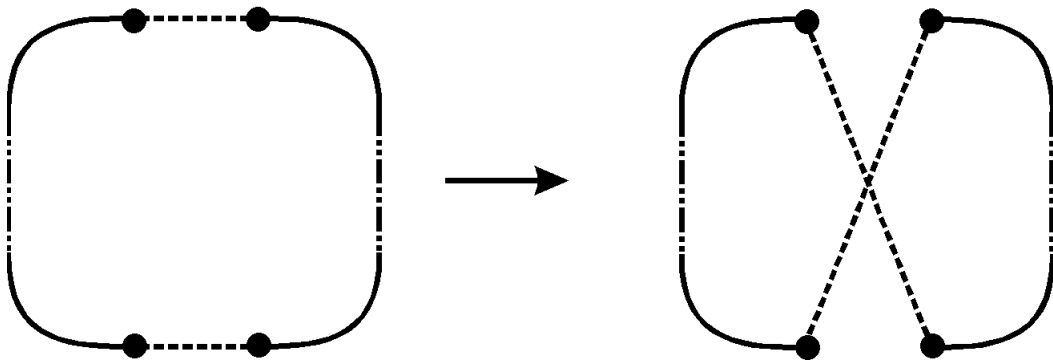
# Variable Neighborhood Search (VNS)

- Princípio: aplicar busca local amostrando vizinhos da solução corrente. Quando não for possível encontrar uma solução vizinha melhor, utilizar uma vizinhança diferente (“maior”).
- Conjunto de vizinhanças pré-selecionadas  
 $N = \{N_1, N_2, \dots, N_{\max}\}$   
 $N_k(s)$ : vizinhos de  $s$  na  $k$ -ésima vizinhança
- Exemplo: problema do caixeiro viajante  
 $N_1$ : 2-opt,  $N_2$ : 3-opt,  $N_3$ : 4-opt...  
mudar de  $k$ -opt para  $(k+1)$ -opt
- Exemplo: otimização 0-1  
 $N_k$ : complementar  $k$  variáveis da solução corrente



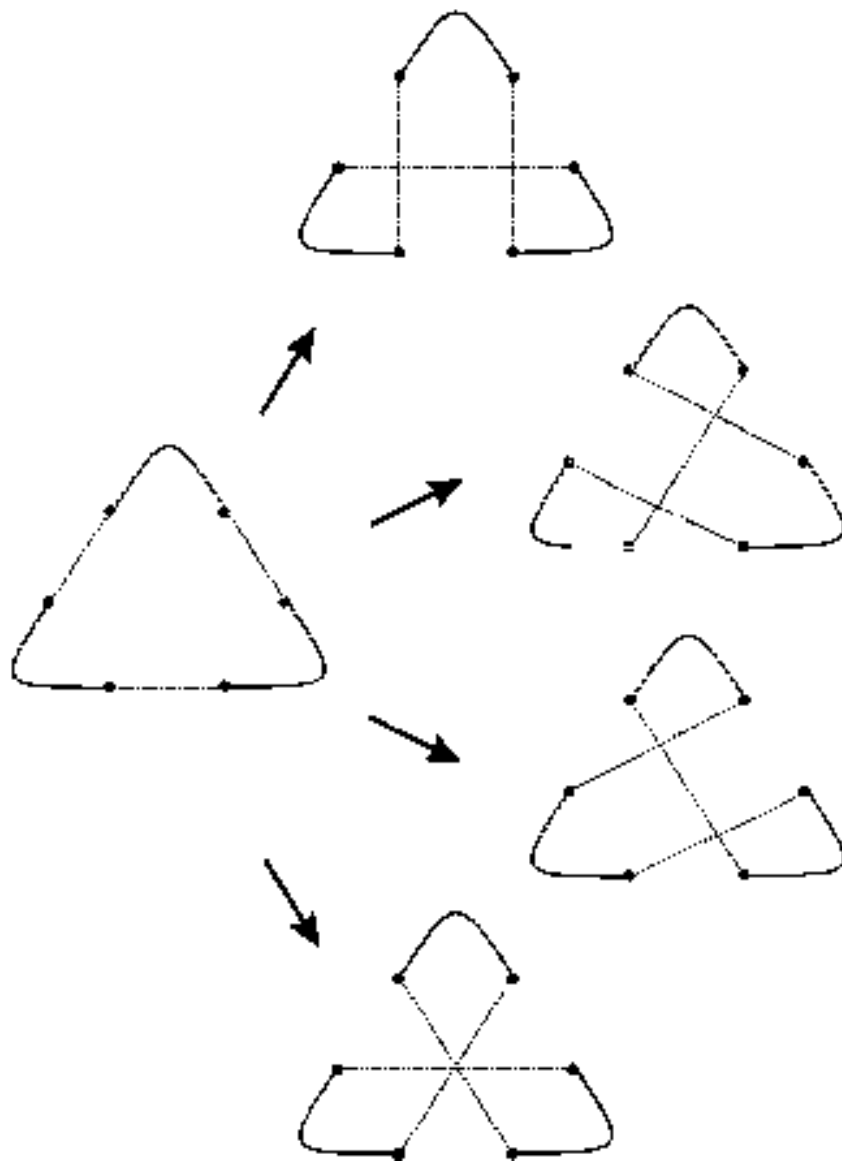
# VNS

- Vizinhaça 2-opt para o problema do caixeiro viajante:



# VNS

- Vizinhaça 3-opt para o problema do caixeiro viajante:



# VNS

- Algoritmo básico:

$s \leftarrow s_0$

**while** (critério de parada) **do**

$k \leftarrow 1$

**while** ( $k \leq \text{max}$ ) **do**

        Gerar aleatoriamente uma solução  
         $s'$  pertencente à vizinhança  
         $N_k(s)$

        Aplicar busca local a partir de  $s'$ ,  
        obtendo a solução  $s''$

**if**  $f(s'') < f(s)$

**then**

$s \leftarrow s''$

$k \leftarrow 1$

**else**  $k \leftarrow k + 1$

**end-if**

**end-while**

**end-while**

# VNS

- Critérios de parada:
  - tempo máximo de processamento
  - número máximo de iterações
  - número máximo de iterações sem melhoria
- Em geral, vizinhanças aninhadas
- GRASP: esforço de aleatorização apenas na construção de soluções
- VNS: aleatorização empregada na busca local, como forma de escapar de ótimos locais



# Variable Neighborhood Descent (VND)

- Princípio: aplicar a mudança de vizinhanças durante a busca local.

- Conjunto de vizinhanças pré-selecionadas

$$N = \{N_1, N_2, \dots, N_{\max}\}$$

$N_k(s)$ : vizinhos de  $s$  na  $k$ -ésima vizinhança

# VND

- Algoritmo básico:

$s \leftarrow s_0$

melhoria  $\leftarrow$  .verdadeiro.

**while** (melhoria) **do**

$k \leftarrow 1$

    melhoria  $\leftarrow$  .falso.

**while** ( $k \leq \text{max}$ ) **do**

        Aplicar busca local a partir de  $s$   
        utilizando a vizinhança

$N_k(s)$ , obtendo a solução  $s'$

**if**  $f(s') < f(s)$

**then**

$s \leftarrow s'$

            melhoria  $\leftarrow$  .verdadeiro.

**else**  $k \leftarrow k + 1$

**end-if**

**end-while**

**end-while**

# Algoritmos genéticos

- Algoritmo probabilístico baseado na analogia entre o processo de evolução natural.
- Durante a evolução, populações evoluem de acordo com os princípios de seleção natural e de “sobrevivência dos mais adaptados” (evolução das propriedades genéticas da população).
- Indivíduos mais bem sucedidos em adaptar-se a seu ambiente terão maior chance de sobreviver e de reproduzir. Genes dos indivíduos mais bem adaptados vão espalhar-se para um maior número de indivíduos em sucessivas gerações.
- Espécies evoluem tornando-se cada vez mais adaptadas ao seu ambiente.

# Algoritmos genéticos

- População representada pelos seus cromossomos: cromossomo  $\Leftrightarrow$  indivíduo  
Novos cromossomos gerados a partir da população corrente e incluídos na população, enquanto outros são excluídos. Geração de novos cromossomos através de mecanismos de reprodução e de mutação.
- Representação do cromossomo (em sua versão original): sequência de *bits* (equivalente ao vetor de pertinência)
- Reprodução: selecionar os cromossomos pais e executar uma operação de *crossover*, que é uma combinação simples das representações de cada cromossomo
- Mutação: modificação arbitrária de uma parte (pequena) do cromossomo



# Algoritmos genéticos

- Algoritmo básico:

Gerar população inicial

**while** critério-de-parada **do**

    Escolher cromossomos reprodutores

    Fazer o *crossover* dos reprodutores

    Gerar mutações

    Avaliar aptidões e atualizar a população

**end-while**

- Parâmetros:

- tamanho da população

- critério de seleção

- critério de sobrevivência dos cromossomos

- taxa de mutação

- critério de parada (estabilização da população, impossibilidade de melhorar a melhor solução, número de gerações)

# Algoritmos genéticos

- Função de aptidão:
  - utilizada para quantificar a qualidade genética dos cromossomos, correspondendo à função de custo em problemas de otimização combinatória.
  - utilizada para selecionar os indivíduos reprodutores.
  - utilizada para decidir se um cromossomo gerado através de um *crossover* substitui ou não um cromossomo reprodutor.
- Crossover: operação probabilística (originalmente), onde os indivíduos mais adaptados têm maior chance de participar
- Crossover uniforme: cada *bit* de um filho é gerado escolhendo-se aleatoriamente um dos pais e repetindo-se o *bit* do pai escolhido

# Algoritmos genéticos

- *Crossover* de um ponto:

reprodutores: 2 cromossomos de  $n$  bits

$$a = (a_1, \dots, a_k, \dots, a_n) \quad b = (b_1, \dots, b_k, \dots, b_n)$$

operação:  $k \in \{1, \dots, n\}$  aleatório

filhos:

$$(a_1, \dots, a_k, b_{k+1}, \dots, b_n) \quad (b_1, \dots, b_k, a_{k+1}, \dots, a_n)$$

- *Crossover* de dois (ou mais) pontos
- *Crossover* por fusão: como o uniforme, mas a probabilidade de escolha de cada pai é proporcional à sua aptidão
- Utilizar consenso no *crossover*: repetir *bits* comuns aos dois reprodutores

# Algoritmos genéticos

- Dificuldade: como tratar restrições e soluções inviáveis?
  - utilizar uma representação que automaticamente garanta que todas as soluções representáveis sejam viáveis
  - utilizar um operador heurístico que garanta a transformação de qualquer solução inviável em outra viável em tempo polinomial (exemplo: retirar itens selecionados no problema da mochila)
  - separar a avaliação da adaptação e da viabilidade
  - utilizar uma função de penalização para penalizar a adaptação de qualquer solução inviável
- Nada proíbe (e recomenda-se!) a utilização de técnicas de otimização e sua incorporação em algoritmos genéticos (mais inteligência)

# Algoritmos genéticos

- Mutação: normalmente implementada com a complementação de bits da população.
- Seleção aleatória dos *bits* a serem modificados: percentual muito baixo do total de bits na população de cromossomos.
- Mutações não passam por testes de aptidão, isto porque a reprodução permite apenas evolução da população, conduzindo a uma população homogênea. Mutação é o mecanismo para introduzir diversidade na população.
- Critérios de atualização da população podem também permitir que pais e filhos permaneçam na população, removendo-se sempre os menos aptos, sendo possível a utilização de cromossomos pais e filhos como reprodutores.

# Algoritmos genéticos

- Exemplo: problema da mochila

*Crossover* com  $k = 4$

Pai1 = (0,1,0,1,1,0,1,1)

Pai2 = (1,0,0,0,0,1,1,0)

Filho1 = (0,1,0,1,0,1,1,0)

Filho2 = (1,0,0,0,1,0,1,1)

Mutação do 5o. *bit*:

(1,0,0,0,1,0,1,1)  $\rightarrow$  (1,0,0,0,1,0,1,1)

# Algoritmos genéticos

- Representações mais eficientes
  - representação de soluções por vetores de pertinência: em muitos problemas, a modificação dos cromossomos (por *crossover* ou mutação) leva freqüentemente a soluções inviáveis
  - tratamento de soluções inviáveis
  - utilizar outras representações
- Geração da população inicial:
  - originalmente, aleatória
  - utilizar heurísticas, e.g. algoritmos gulosos aleatorizados
- Crossover otimizado: utilizar mais “inteligência” nesta operação, por exemplo, por meio de um algoritmo relacionado ao problema (busca local).

# Algoritmos genéticos

- Seleção de pais por compatibilidade: escolher um pai, em seguida encontrar o pai mais compatível
- Operador de melhoria (por exemplo, busca local) que tenta melhorar cada filho gerado
- Mutação estática (probabilidade constante de alterar um *bit*) ou mutação adaptativa (aplicar mutação a *bits* selecionados, para tentar garantir viabilidade)



# Algoritmos genéticos

- Exemplo: Problema de Steiner em grafos

Cromossomos: vetor de pertinência indicando se um vértice não-terminal pertence ou não à árvore

*Crossover* otimizado: encontrar a árvore geradora de peso mínimo sobre o grafo de caminhos mais curtos obtidos utilizando os vértices não-terminais presentes em pelo menos um dos cromossomos reprodutores.

O filho gerado conterá os vértices não-terminais utilizados nas arestas da árvore geradora de peso mínimo obtida.

# Algoritmos genéticos

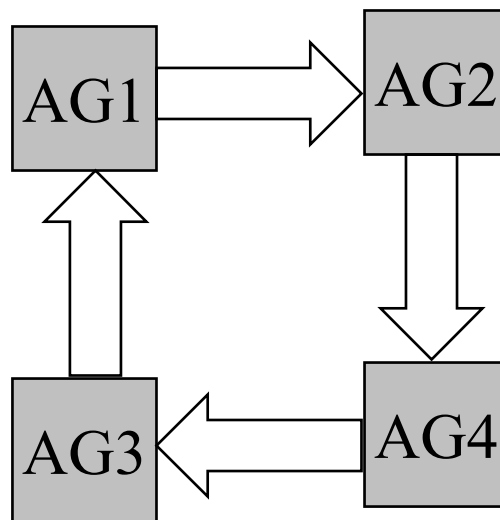
- Níveis possíveis de paralelismo:
  - (a) ao nível das fases: por exemplo, calcular em paralelo a função de aptidão para os elementos da população
  - (b) ao nível das populações: dividir a população original em diversas subpopulações, cujas evoluções ocorrem em paralelo com eventuais trocas dos melhores indivíduos entre as subpopulações
  - (c) ao nível dos processos: executar diversos algoritmos genéticos em paralelo, diferentes e independentes entre si
- Níveis mais altos (b) e (c) constituem a base para algoritmos genéticos paralelos

# Algoritmos genéticos

- Modelo centralizado: cálculos básicos ou fases do algoritmo genético seqüencial realizados por processadores escravos, enquanto o mestre mantém a população (e.g., o mestre envia um ou mais indivíduos para cada escravo para que seja efetuado o cálculo da função de aptidão e coleta informações deles provenientes)
- Modelo distribuído: aplicação direta do paralelismo ao nível das populações (modelo de ilhas, pois a população é dividida em diversas subpopulações, cada uma evoluindo quase isoladamente, como se estivesse em sua “ilha”). Um processador central é necessário para controlar a troca de indivíduos entre as populações, assim como o término da execução.

# Algoritmos genéticos

- Modelo cooperativo: diversos algoritmos genéticos diferentes em paralelo atuam sobre a mesma população original, eventualmente cooperando e trocando informações.



# Algoritmos genéticos

- Modelo de redes ou de difusão: derivado do modelo distribuído pelo aumento do número de subpopulações, de modo que exista um único indivíduo em cada ilha. Os indivíduos são conectados entre si por uma de muitas possíveis redes de interconexão (e.g., malha bidimensional, hipercubo, anel). Cada indivíduo “vive” no seu local e ocasionalmente interage com alguns de seus vizinhos, i.e., com certa probabilidade ocorre o *crossover* de um indivíduo com algum de seus vizinhos e ele é substituído pelo “filho” assim gerado (ocorre a difusão progressiva de bons indivíduos através da rede de interconexão).

# Colônias de formigas

- Princípio: simulação do comportamento de um conjunto de agentes que cooperam para resolver um problema de otimização através de comunicações simples.
- Formigas parecem ser capazes de encontrar seu caminho (do formigueiro para uma fonte de alimentos e de volta, ou para contornar um obstáculo) com relativa facilidade.
- Estudos entmológicos constataram em muitos casos que esta capacidade resulta da interação entre a comunicação química entre formigas (através de uma substância chamada feromônio) e um fenômeno emergente causado pela presença de muitas formigas.
- Exemplo: caixeiro viajante

# Colônias de formigas

- Exemplo: caixeiro viajante

Cada formiga age da seguinte maneira:

- (a) Em cada passo, usa uma regra probabilística para escolher a cidade para onde irá mover-se, dentre aquelas ainda não visitadas.
- (b) A probabilidade de escolher a aresta  $(i,j)$  é diretamente proporcional:
  - à quantidade de feromônio na aresta
  - ao inverso do comprimento da aresta
- (c) Uma formiga se lembra das cidades já visitadas.
- (d) Após um ciclo ter sido completado, a formiga deixa um rastro positivo  $\Delta t_{ij}$  de feromônio em cada aresta  $(i,j)$  do ciclo ( $\Delta t_{ij} = 0$  para as arestas fora do ciclo).
- (e) Calcule a evaporação de feromônio em cada aresta  $(i,j)$ , considerando-se um fator de evaporação  $\beta < 1$ :  $t_{ij} \leftarrow \beta \cdot t_{ij} + \Delta t_{ij}$ .

# Colônias de formigas

- Como os valores iniciais de  $t_{ij}$  são os mesmos para todas as arestas, inicialmente as formigas estão livres para mover-se quase aleatoriamente.
- Considerando-se  $M$  formigas movendo-se simultaneamente,  $t_{ij}$  é o instrumento pelo qual as formigas comunicam-se: as formigas escolhem sempre com maior probabilidade as arestas com maiores quantidades de feromônio. Desta forma, formigas deixando rastro sobre um conjunto de arestas contribuem para torná-las mais “desejáveis” para as demais formigas.
- O algoritmo de colônias de formigas pode ser visto como a superposição concorrente de  $M$  procedimentos de formigas isoladas.



# Colônias de formigas

- Cada formiga, caso o rastro não tivesse efeito, mover-se-ia de acordo com uma regra local, gulosa e probabilística, que provavelmente levaria para uma solução final ruim (como no caso do algoritmo usando a regra de “inserção mais próxima”: as primeiras arestas são boas, mas as escolhas nos passos finais ficam restringidas pelos passos anteriores).
- Considerando-se agora o efeito da presença simultânea de muitas formigas, então cada uma delas contribui para uma parte do rastro distribuído.
- Bons conjuntos de arestas serão seguidos por muitas formigas e, em consequência, receberão uma maior quantidade do rastro.

# Colônias de formigas

**Passo 1.** Sejam  $Q$  e  $t_0$  constantes, e seja  $c(S^*) \leftarrow \infty$ .

**Passo 2.** Fazer  $\Delta t_{ij} \leftarrow 0$  e  $t_{ij} \leftarrow 0$  para cada aresta  $(i,j)$ .

**Passo 3.** Para cada formiga  $k = 1,2,\dots,M$ :

- (a) Selecionar a cidade inicial da formiga  $k$
- (b) Obter um ciclo  $T_k$ , usando o procedimento seguido individualmente por cada formiga.
- (c) Seja  $L_k$  o comprimento do ciclo  $T_k$ .
- (d) Se  $L_k < c(S^*)$ , então fazer  $S^* \leftarrow T_k$ .
- (e) Calcular a quantidade de rastro deixado por esta formiga:  $\Delta t_{ij}(k) \leftarrow Q/L_k$ , se a aresta  $(i,j)$  pertence ao ciclo  $T_k$ ;  $\Delta t_{ij}(k) \leftarrow 0$  caso contrário.

(f) Fazer  $\Delta t_{ij} \leftarrow \Delta t_{ij} + \Delta t_{ij}(k)$ .

**Passo 4.** Fazer  $t_{ij} \leftarrow \beta \cdot t_{ij} + \Delta t_{ij}$

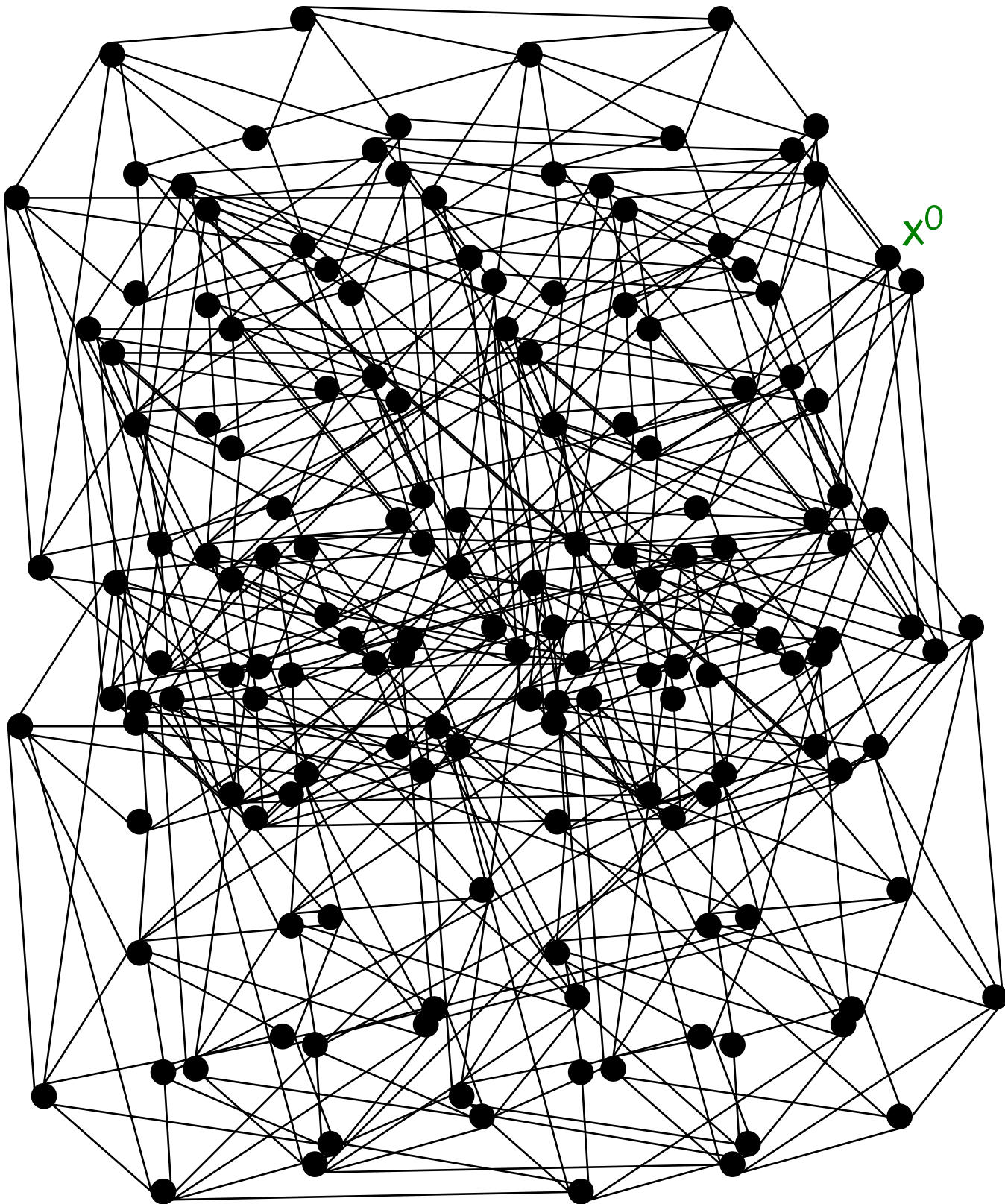
**Passo 5.** Se o melhor ciclo  $S^*$  não tiver sido alterado ao longo das últimas  $n_{max}$  iterações globais, então terminar retornando  $S^*$  como a melhor solução encontrada; caso contrário retornar para o passo 2.

# Colônias de formigas

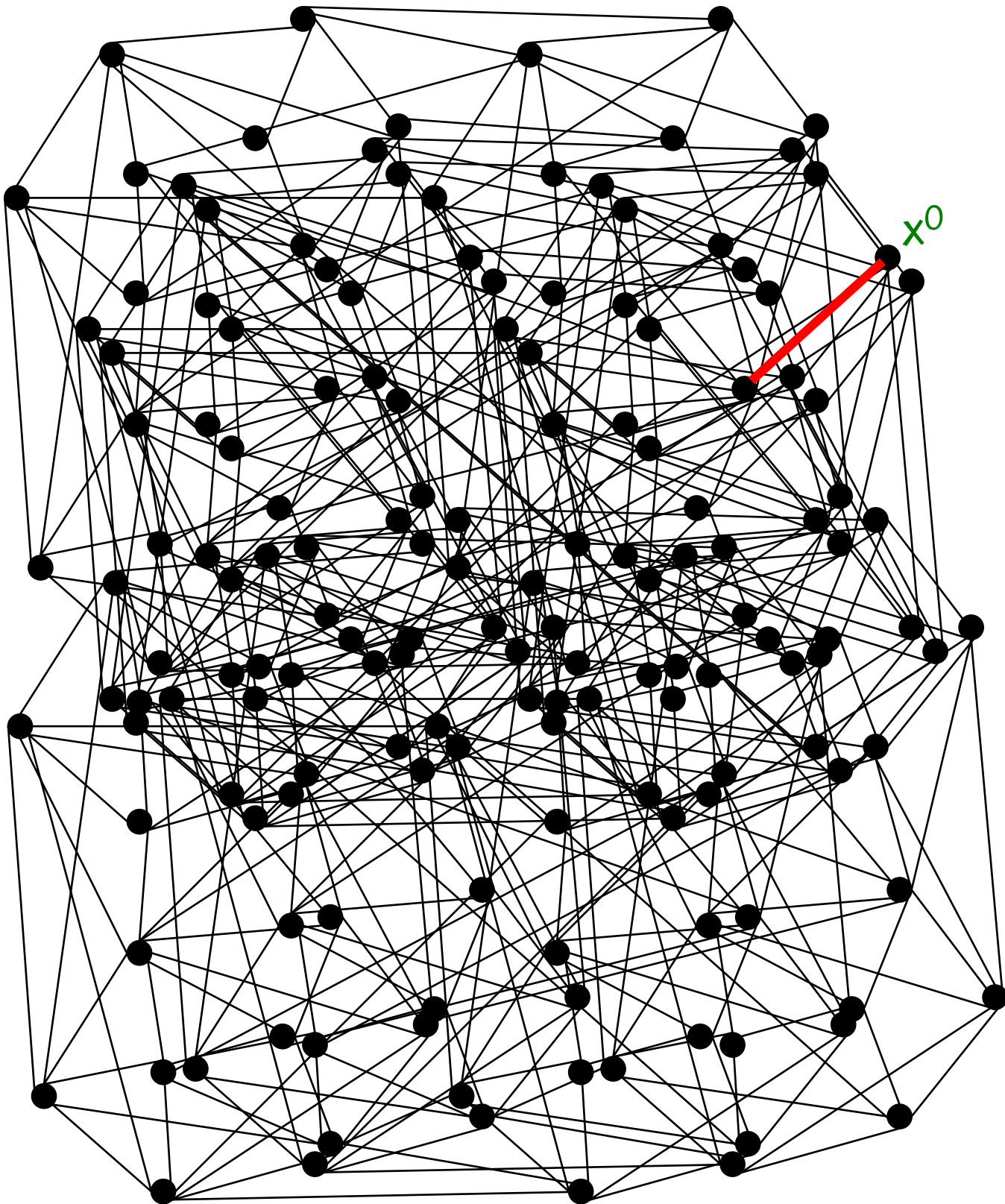
## Comentários:

- (a) O comportamento do modelo computacional não é exatamente o modelo natural de colônias de formigas (e.g., na vida real as formigas deixam rastro durante seu movimento, e não após terem encontrado seu destino final).
- (b) As formigas comunicam-se pela troca de um tipo especial de informação (i.e. pela quantidade de rastro, ou feromônio), sem comunicação direta entre elas, e apenas informações locais são usadas para tomar decisões: o modelo é apropriado para uma paralelização baseada no uso de uma pequena quantidade de informação global.
- (c) Como os algoritmos genéticos, colônias de formigas também são baseadas em uma população de agentes, e não apenas em um único agente (solução).
- (d) Paralelização trivial (um processador para cada formiga)

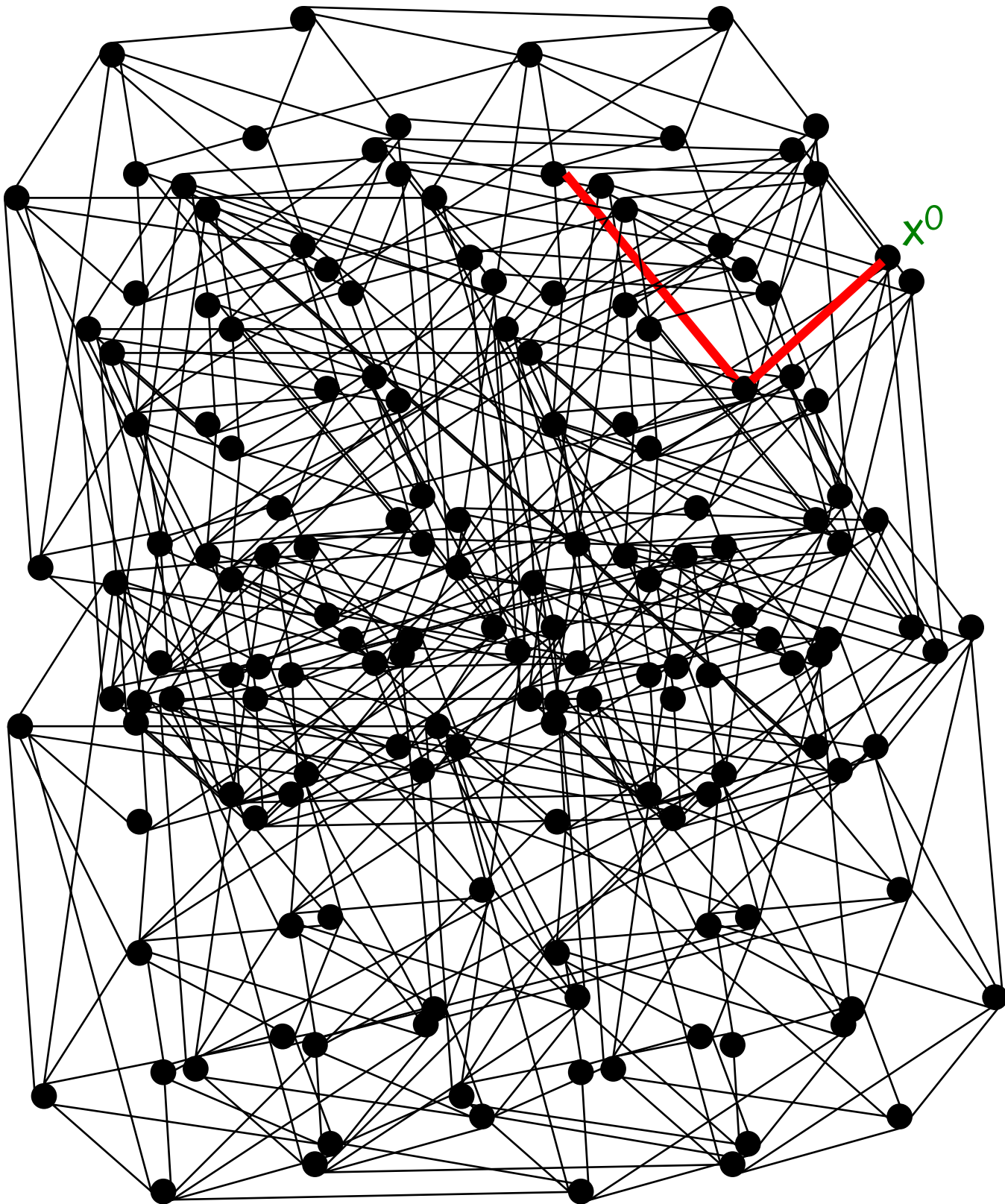
# Métodos de trajetória



# Métodos de trajetória

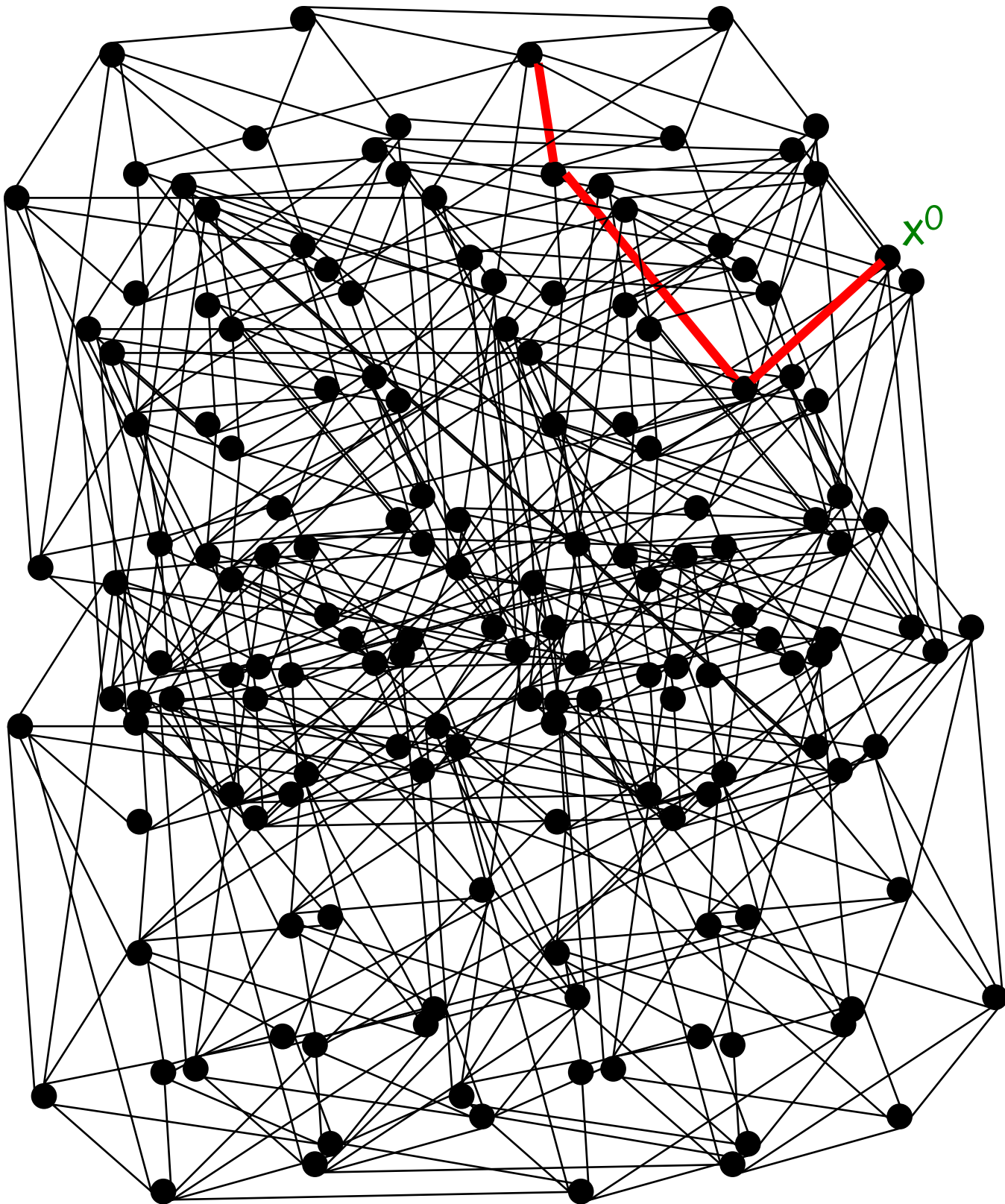


# Métodos de trajetória

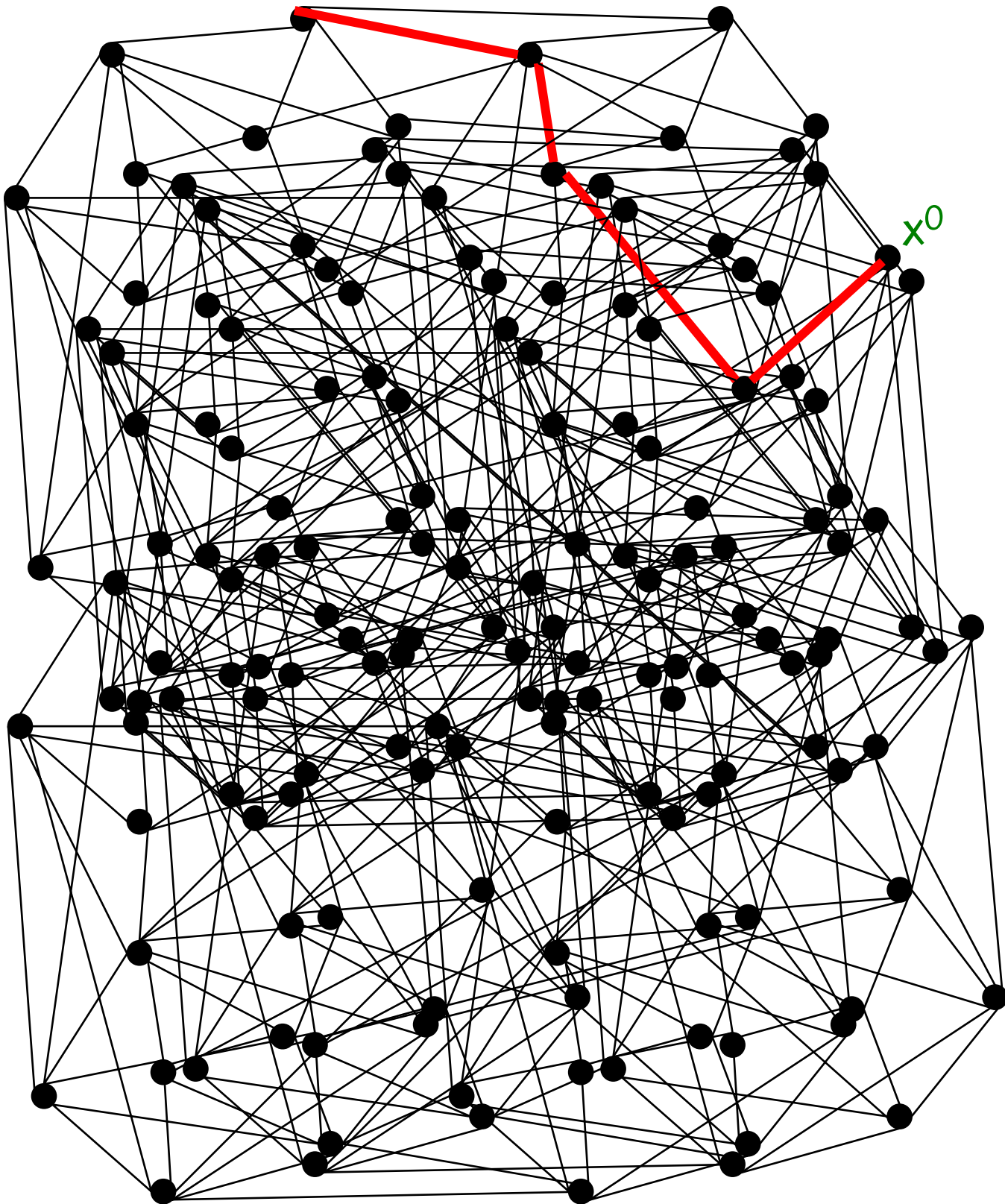




# Métodos de trajetória

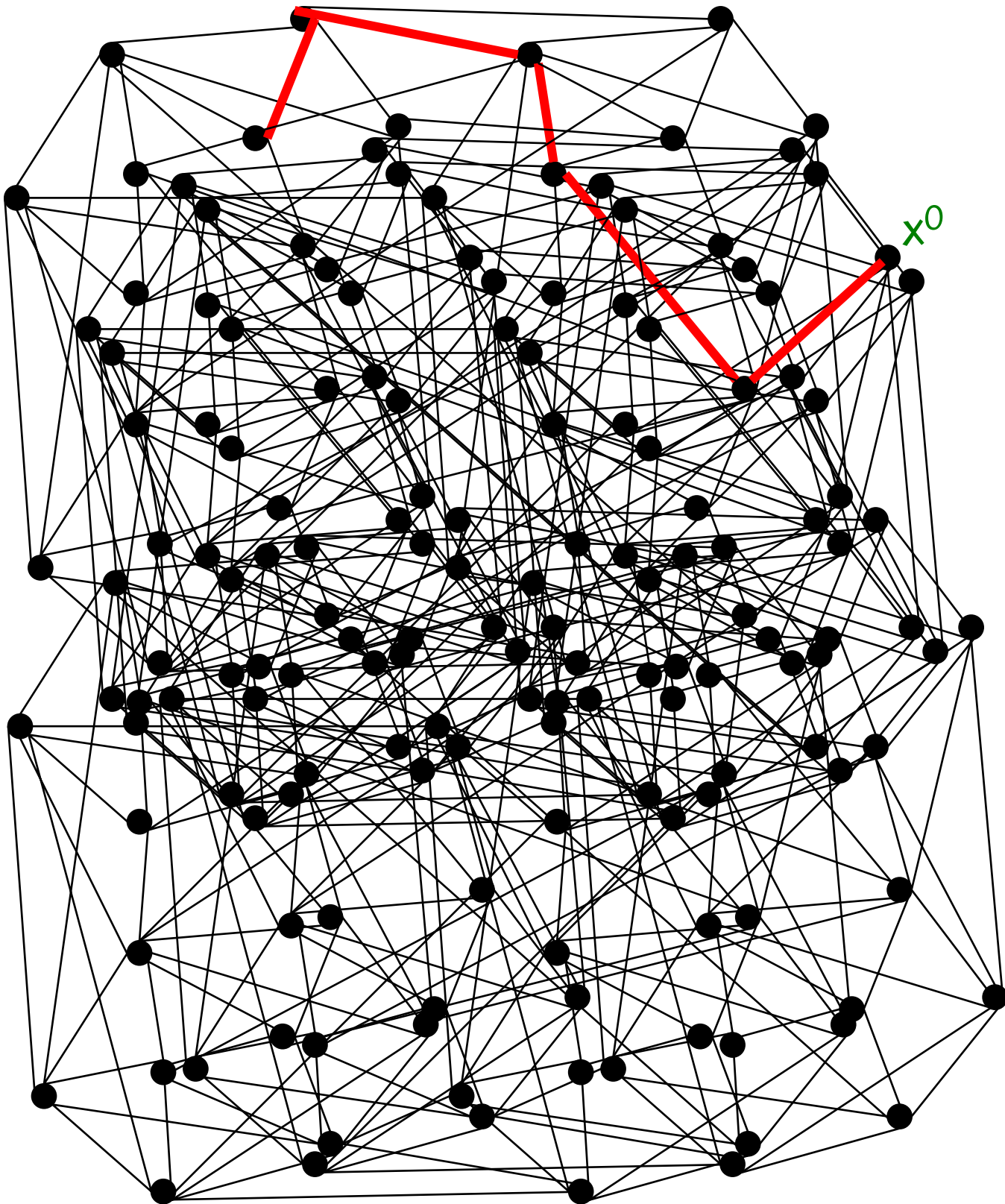


# Métodos de trajetória

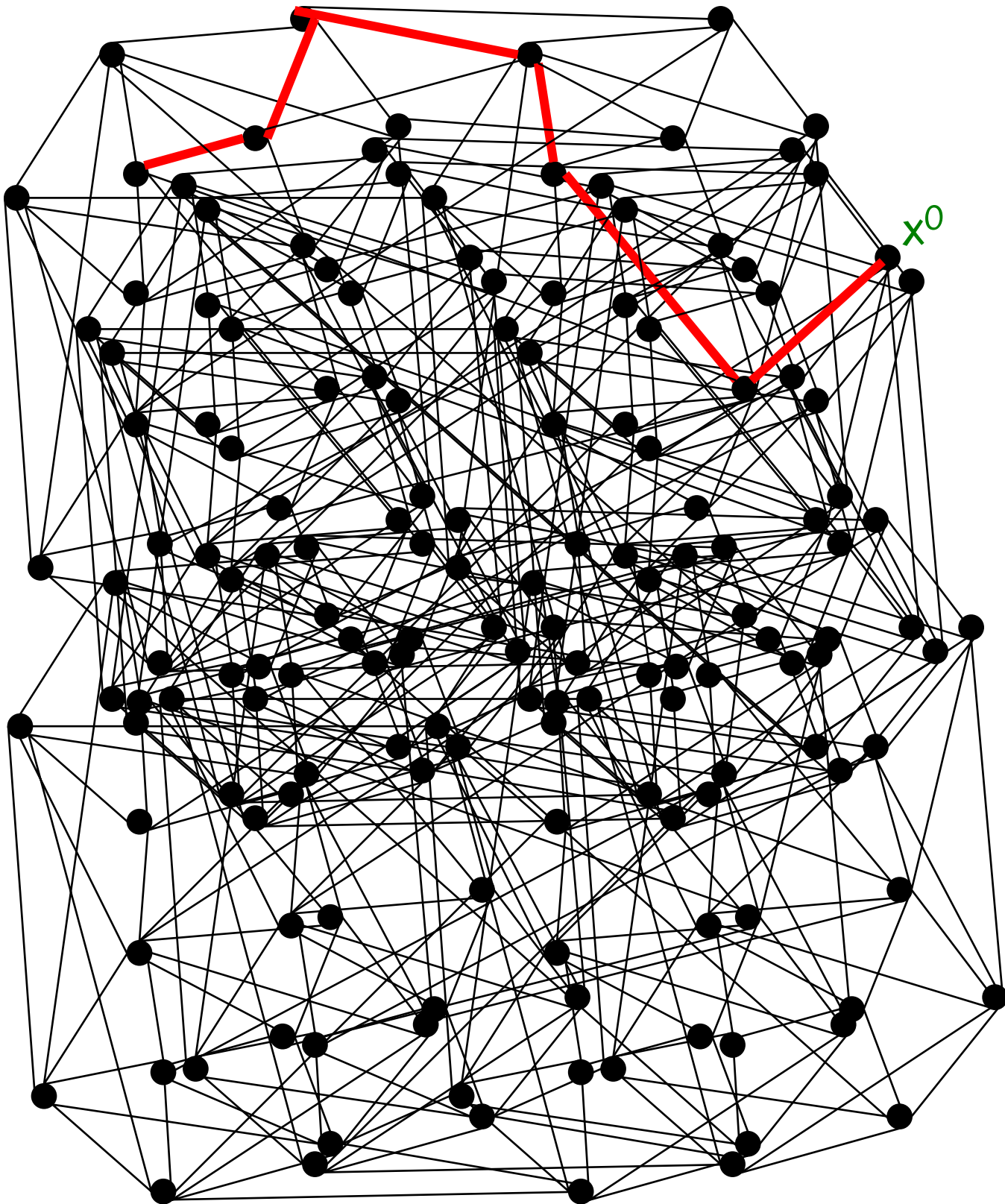




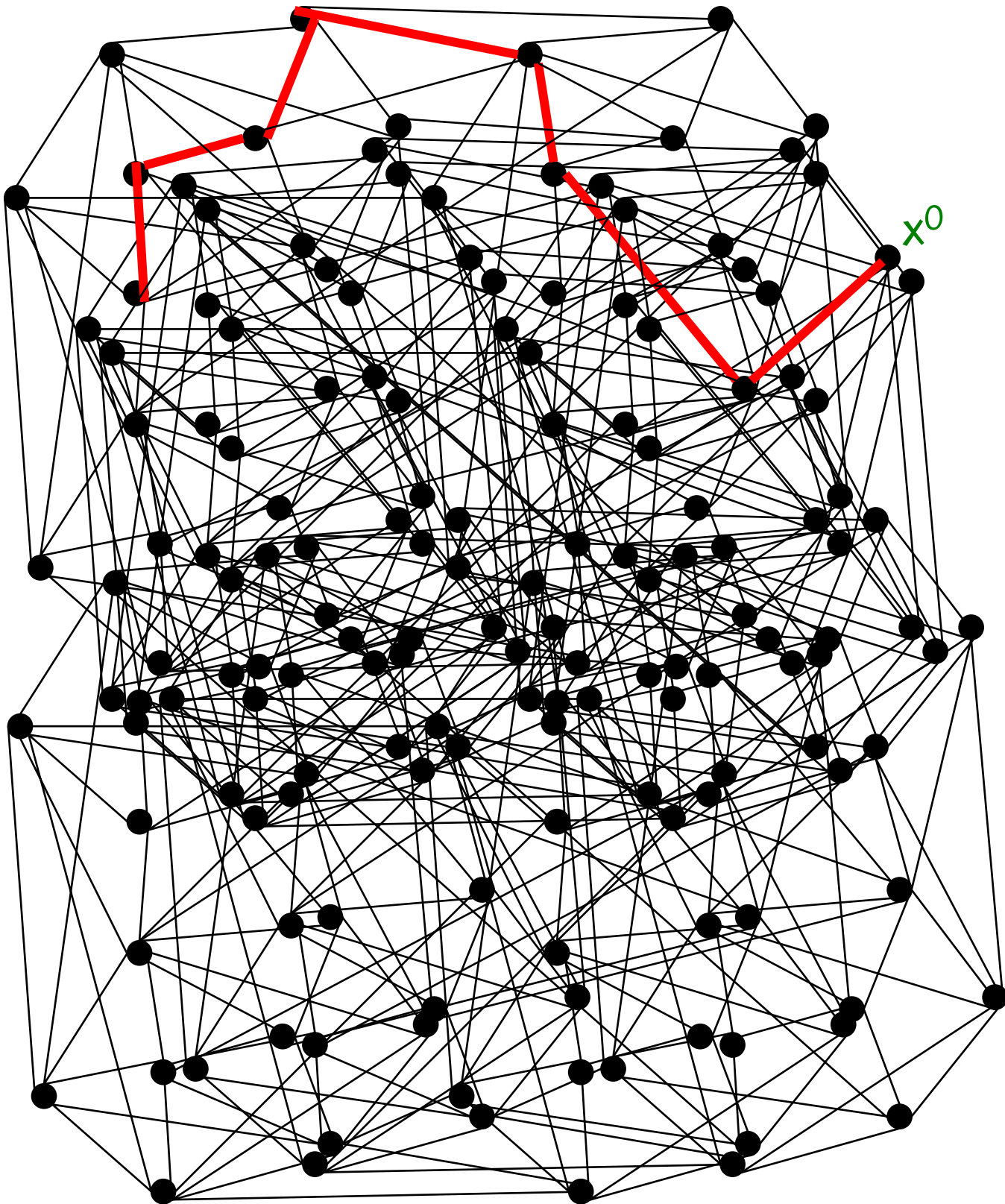
# Métodos de trajetória



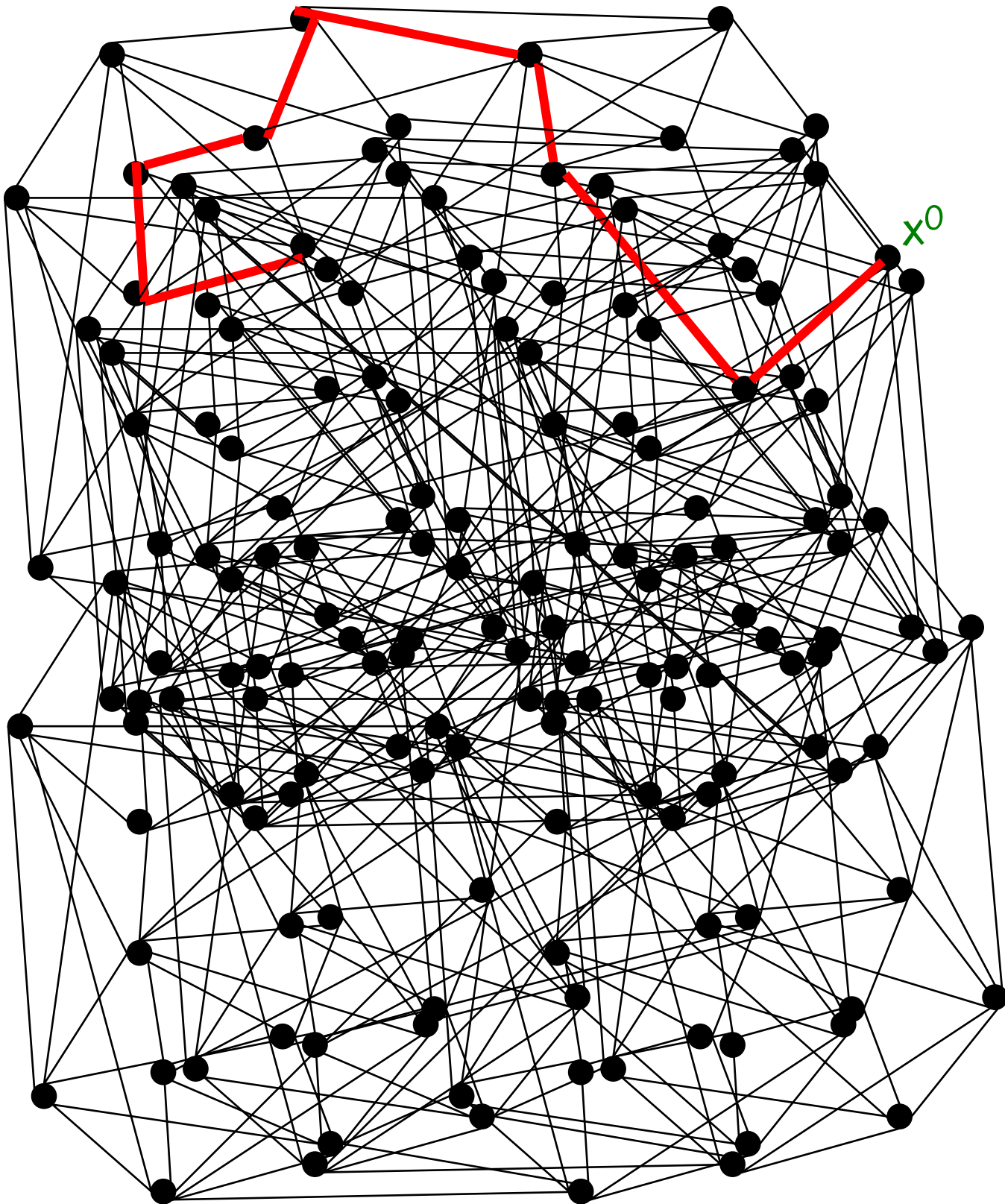
# Métodos de trajetória



# Métodos de trajetória

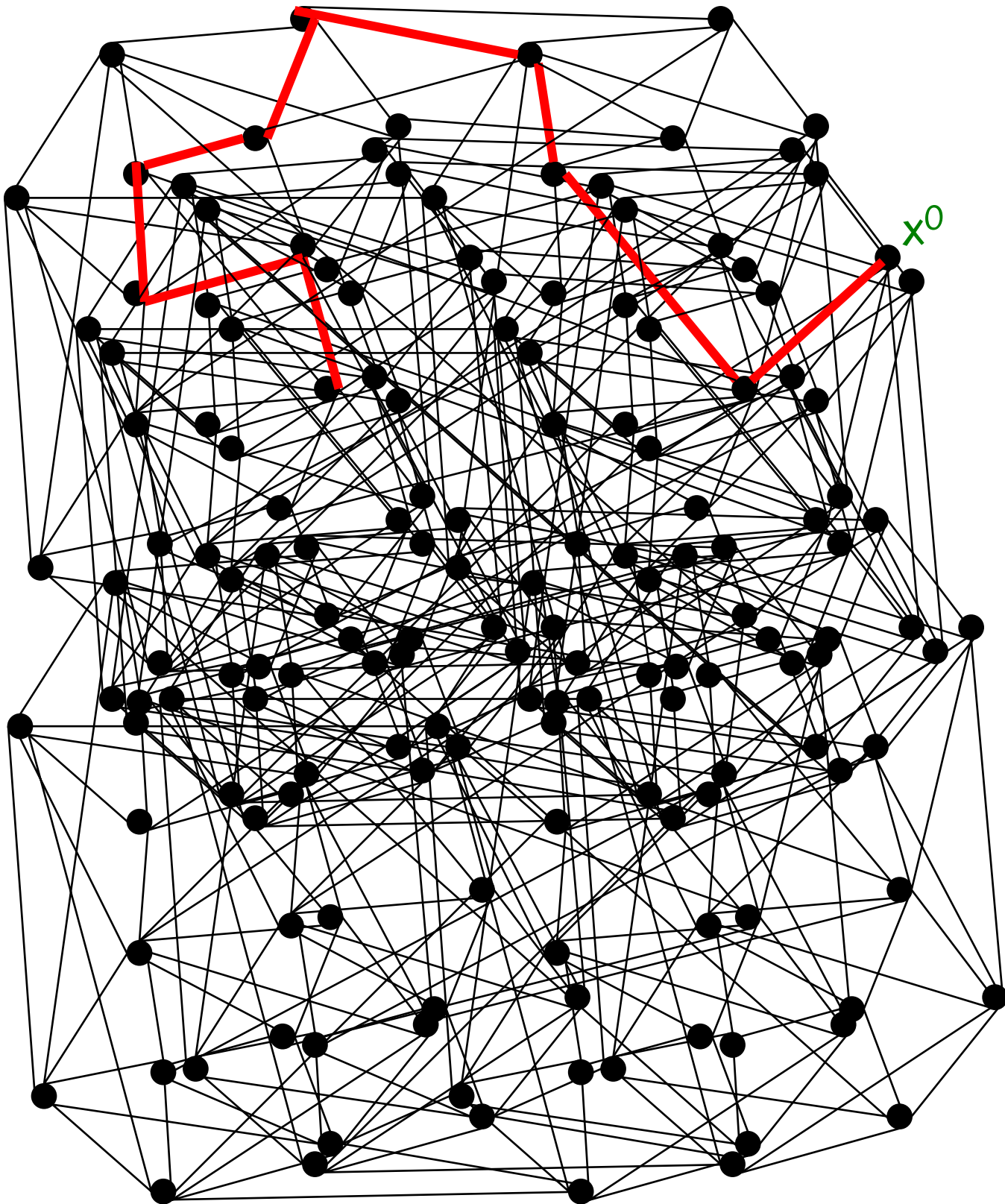


# Métodos de trajetória

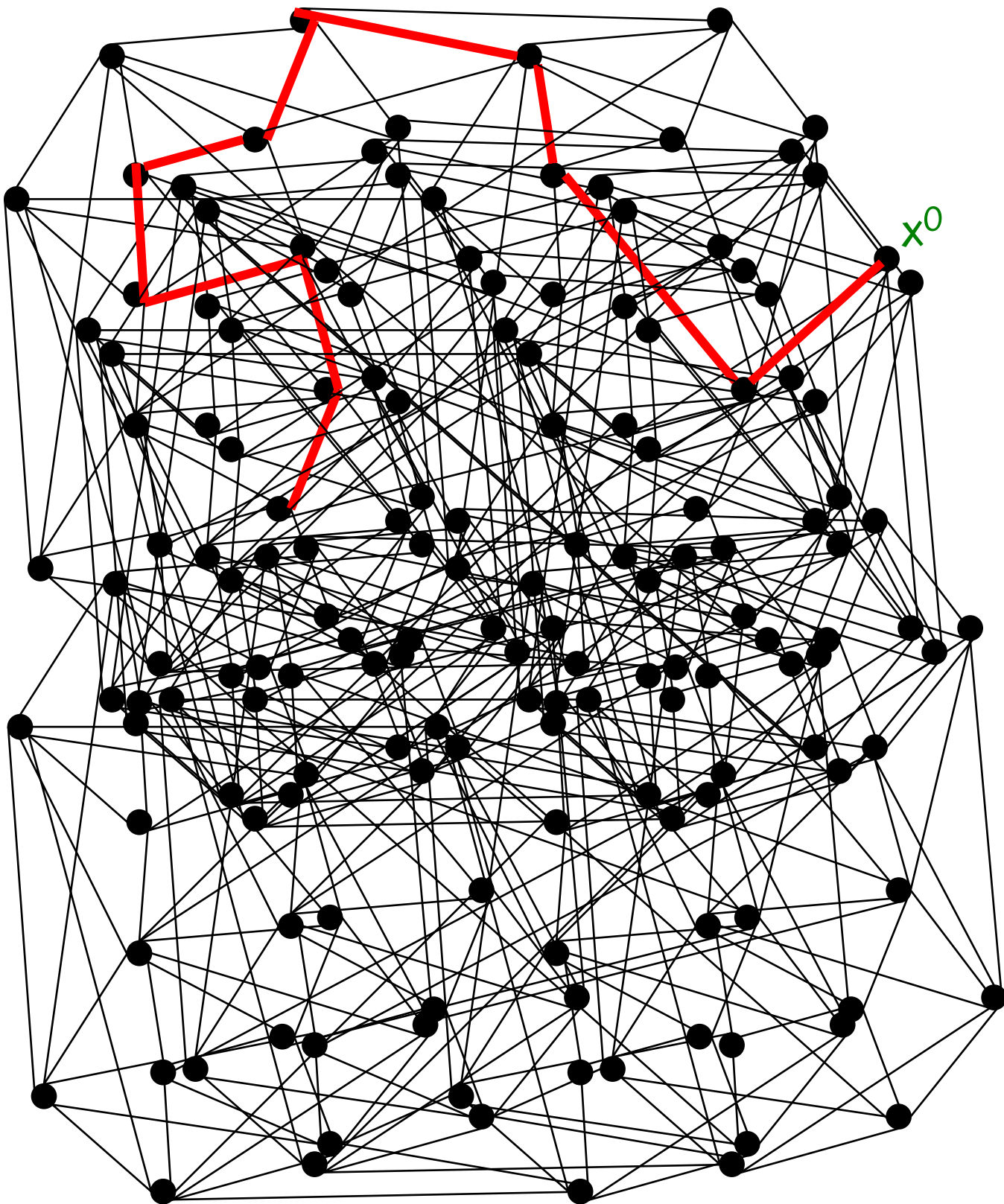




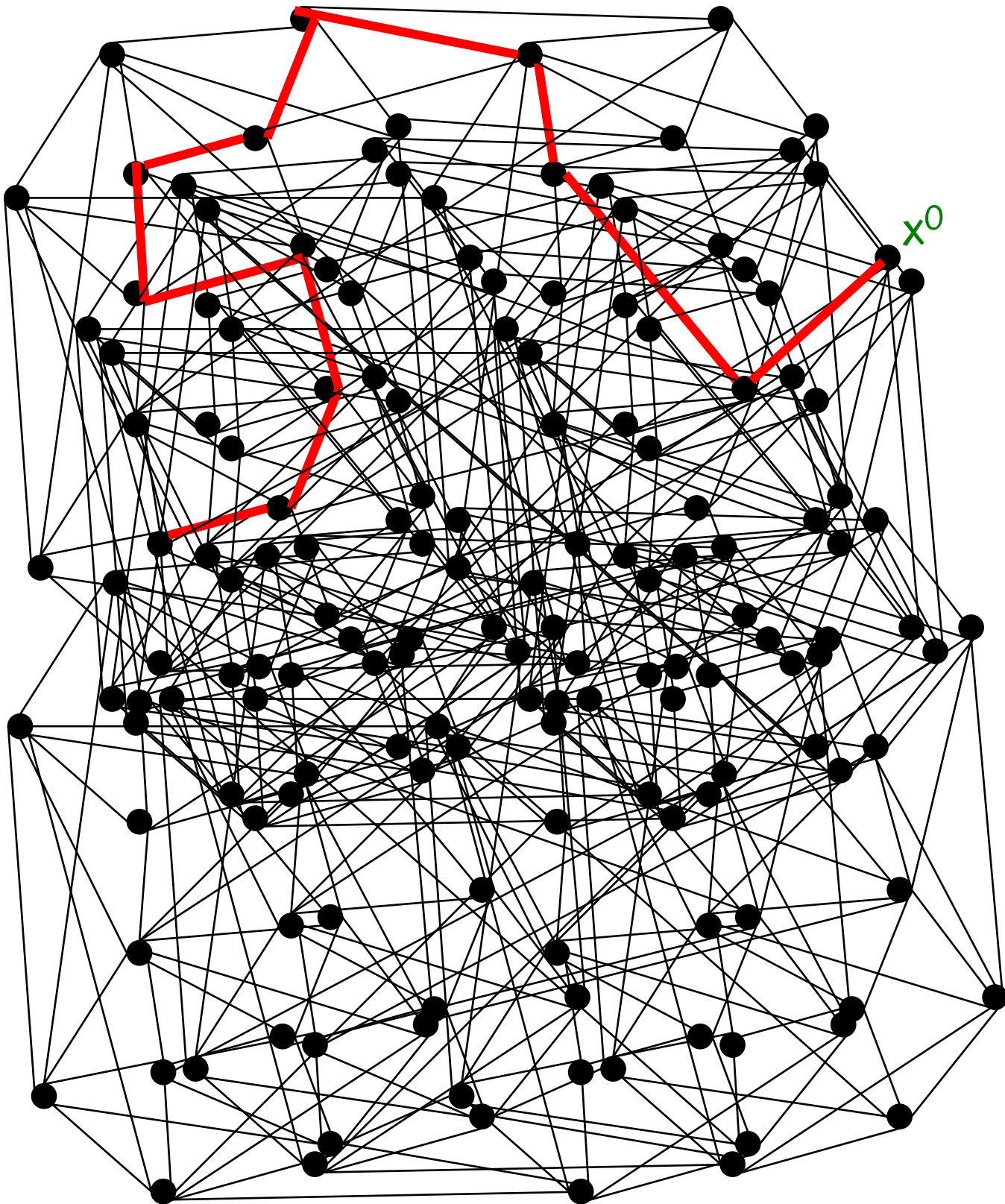
# Métodos de trajetória



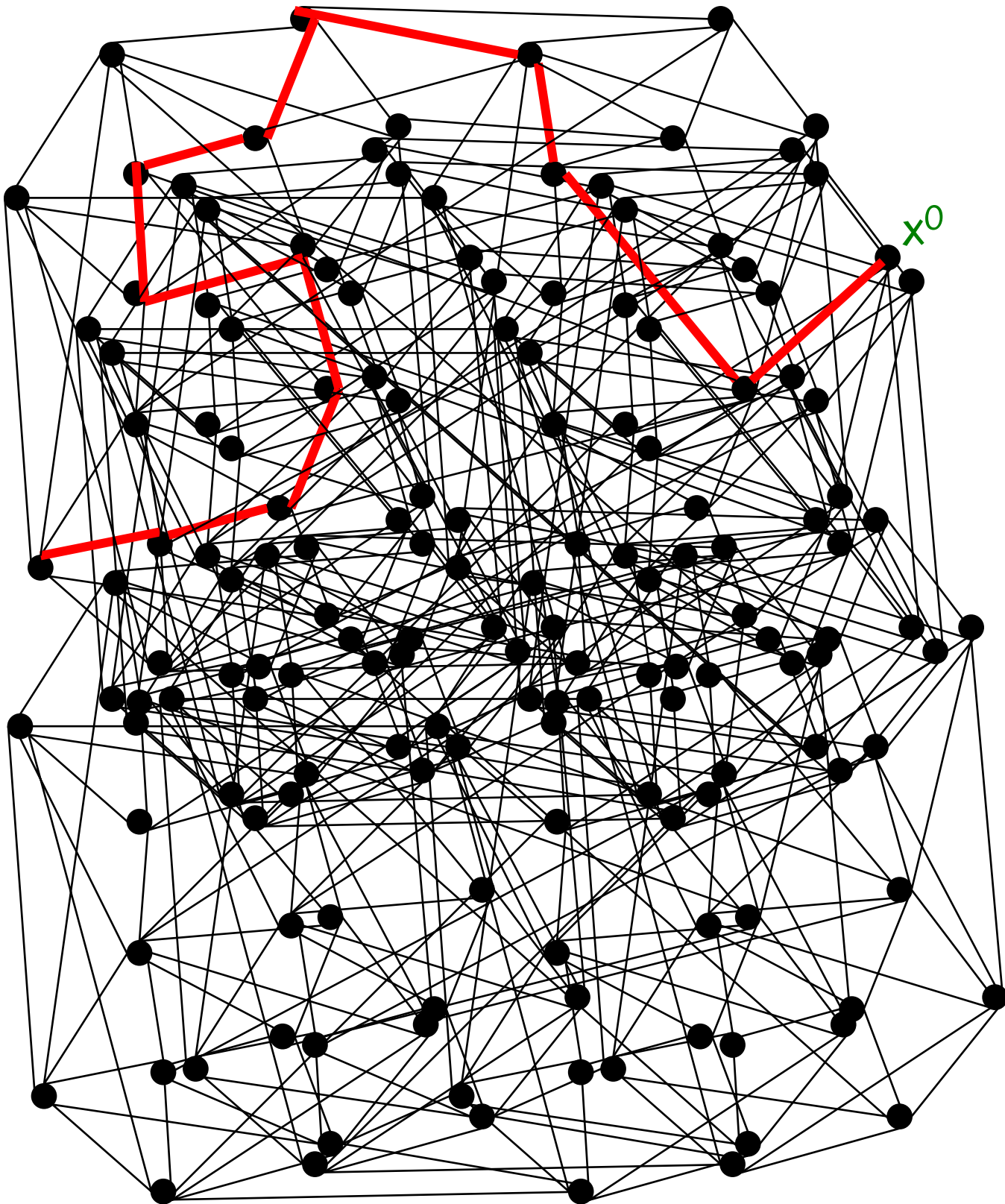
# Métodos de trajetória



# Métodos de trajetória

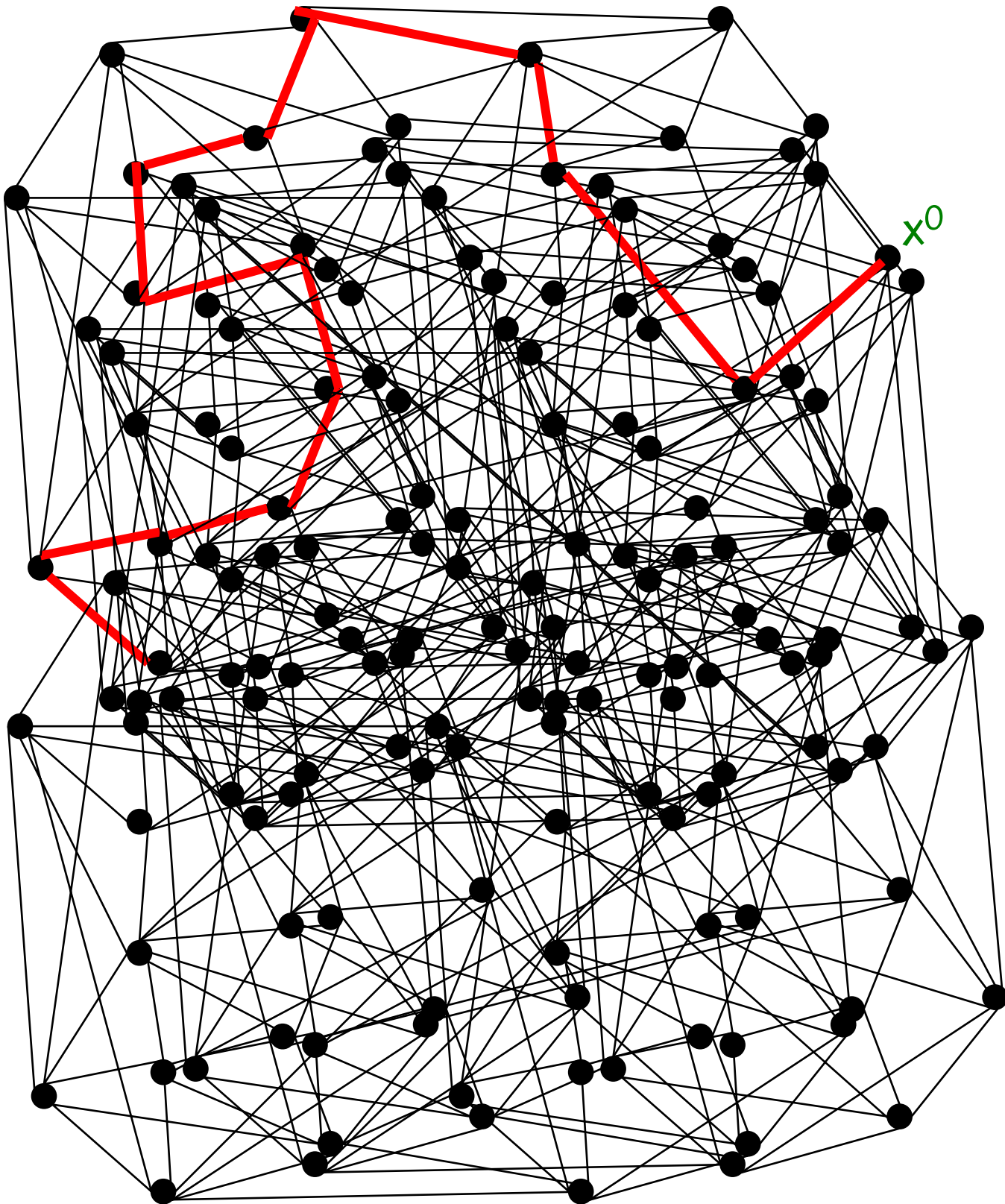


# Métodos de trajetória

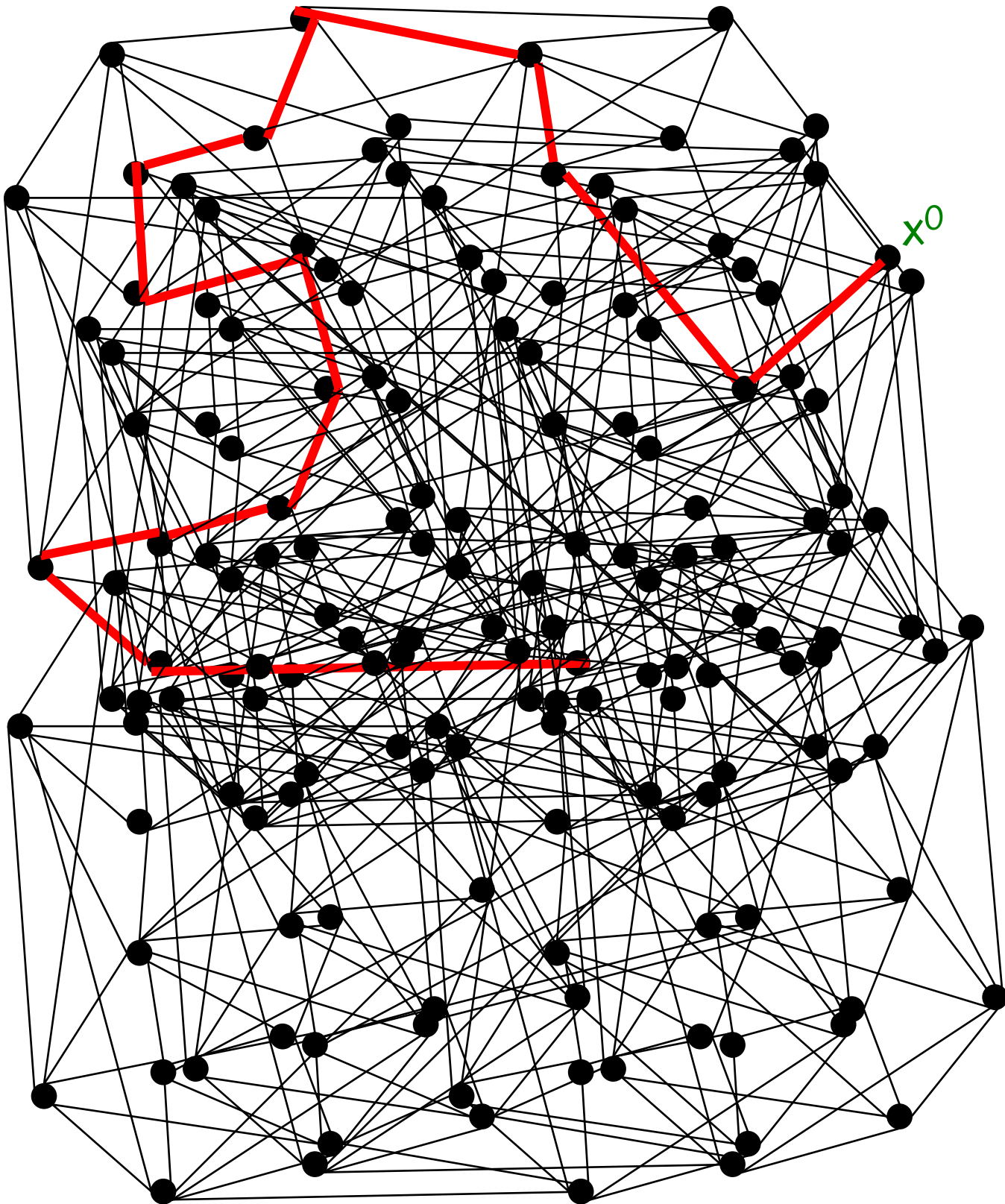




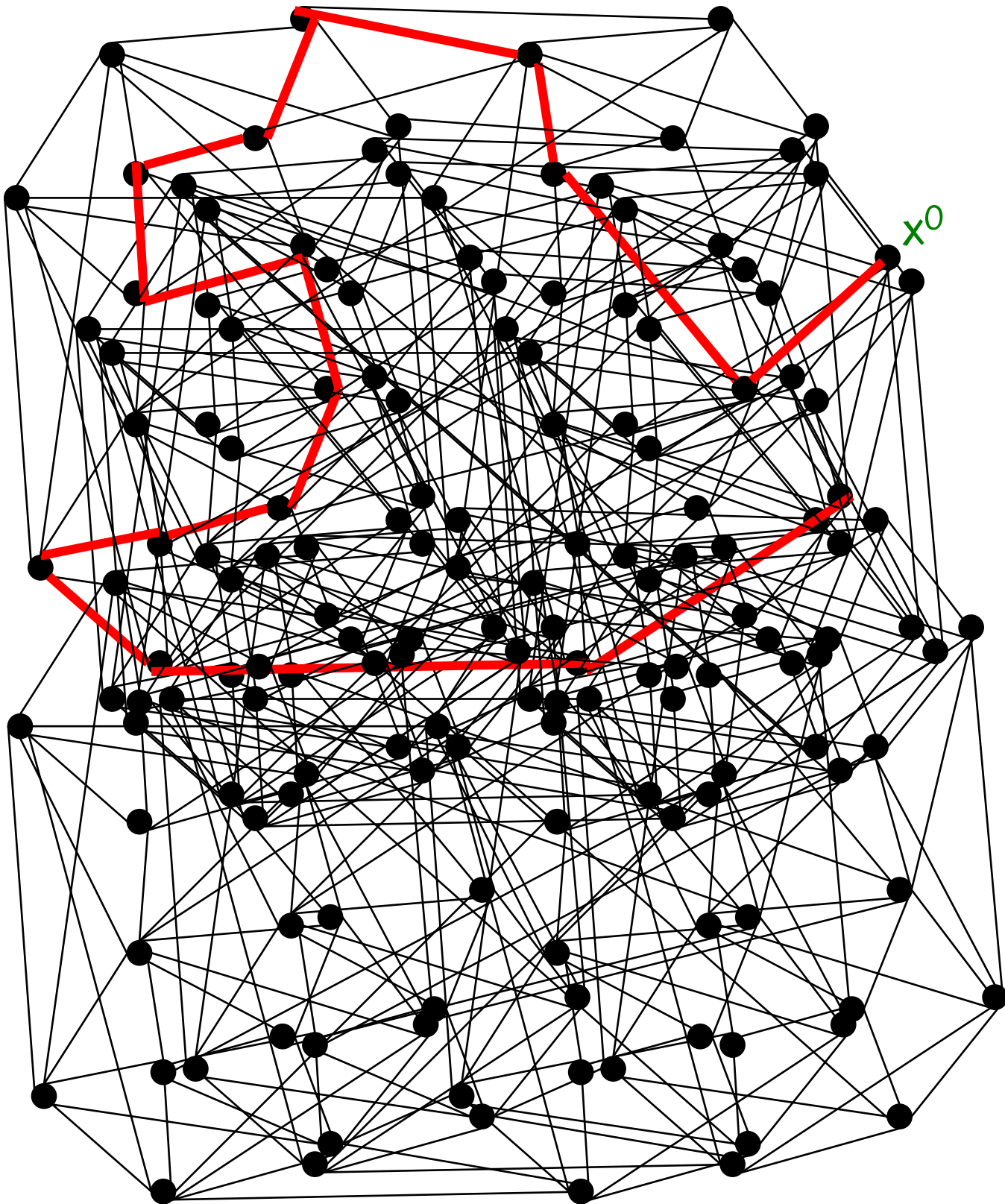
# Métodos de trajetória



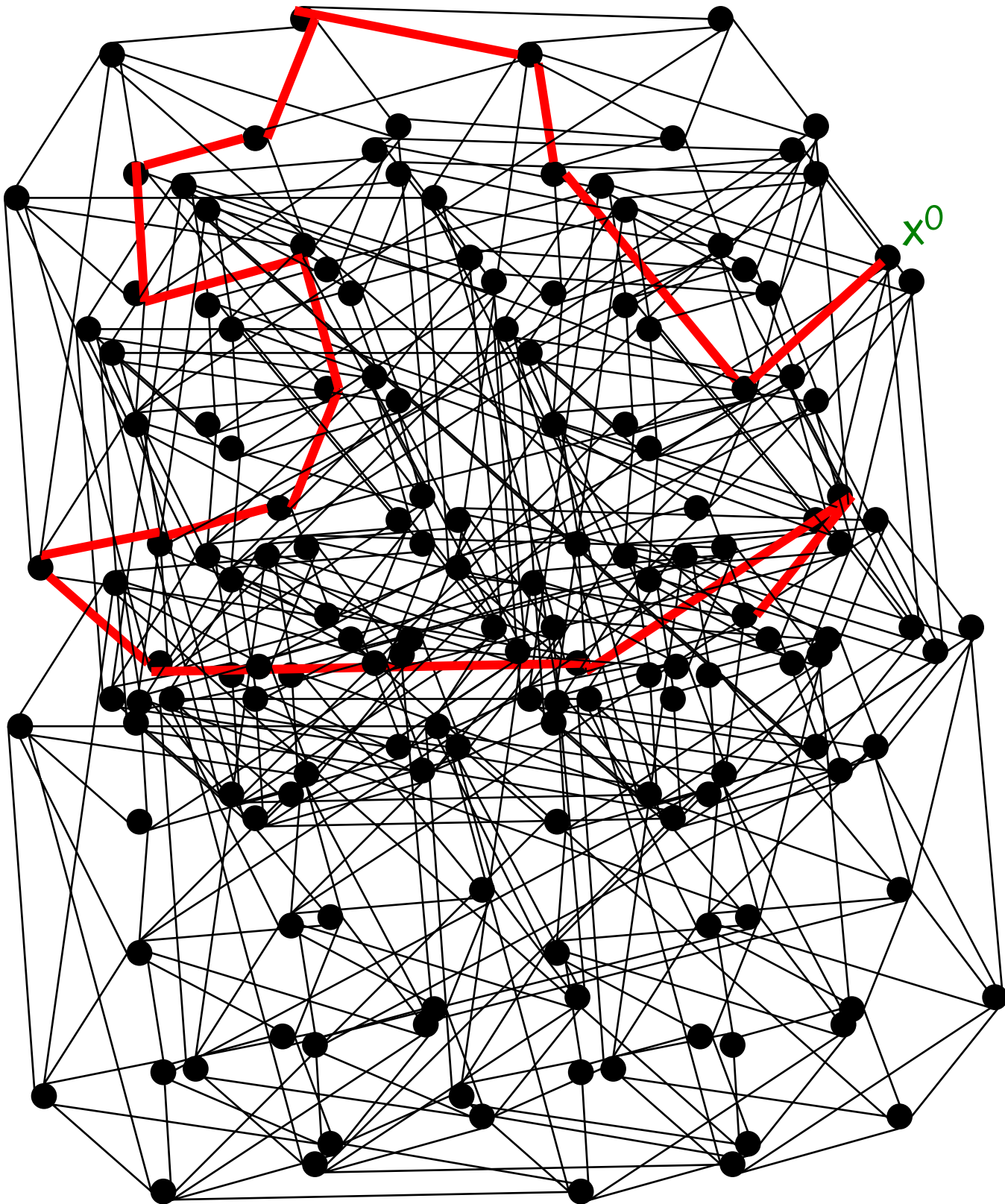
# Métodos de trajetória



# Métodos de trajetória

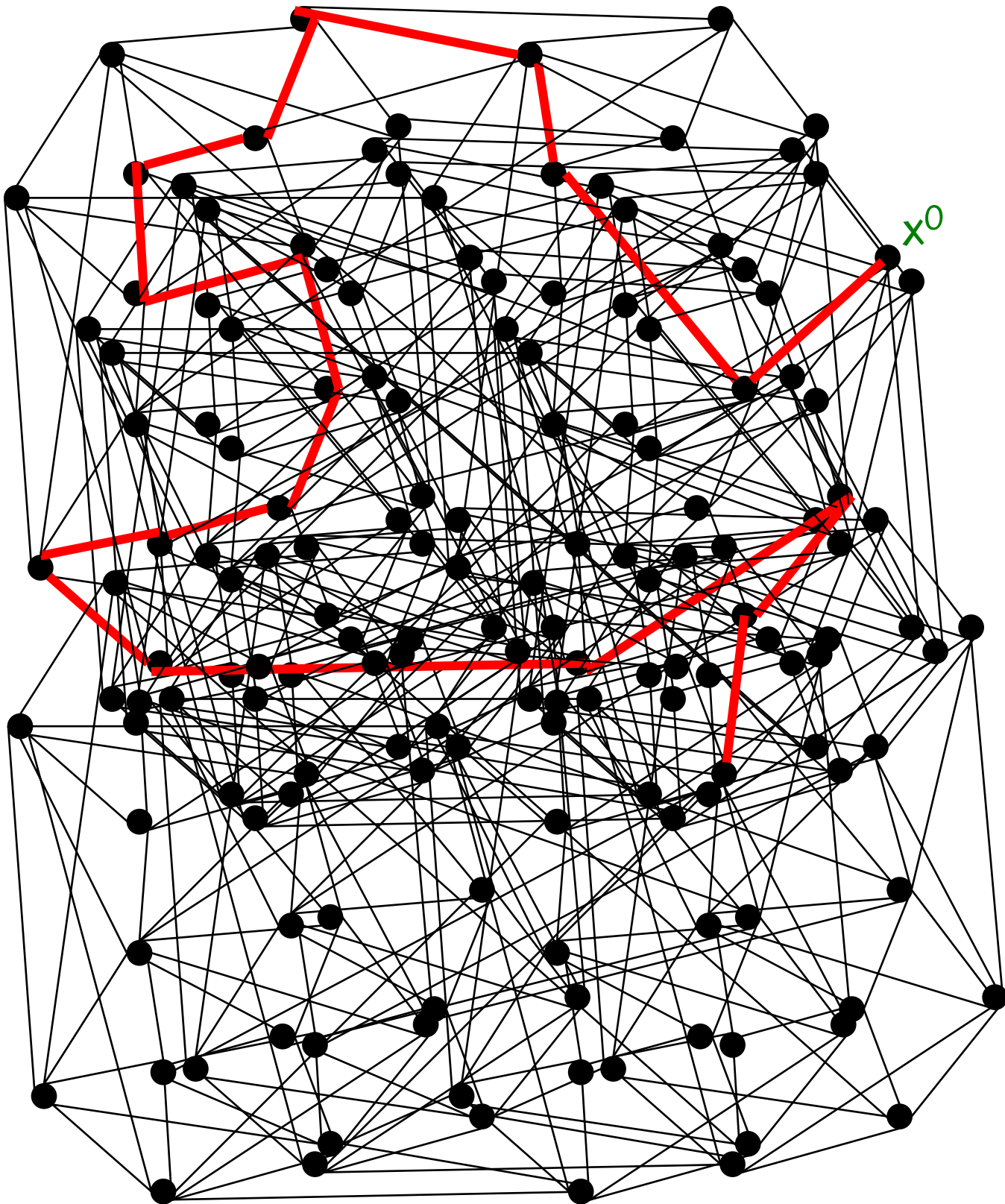


# Métodos de trajetória

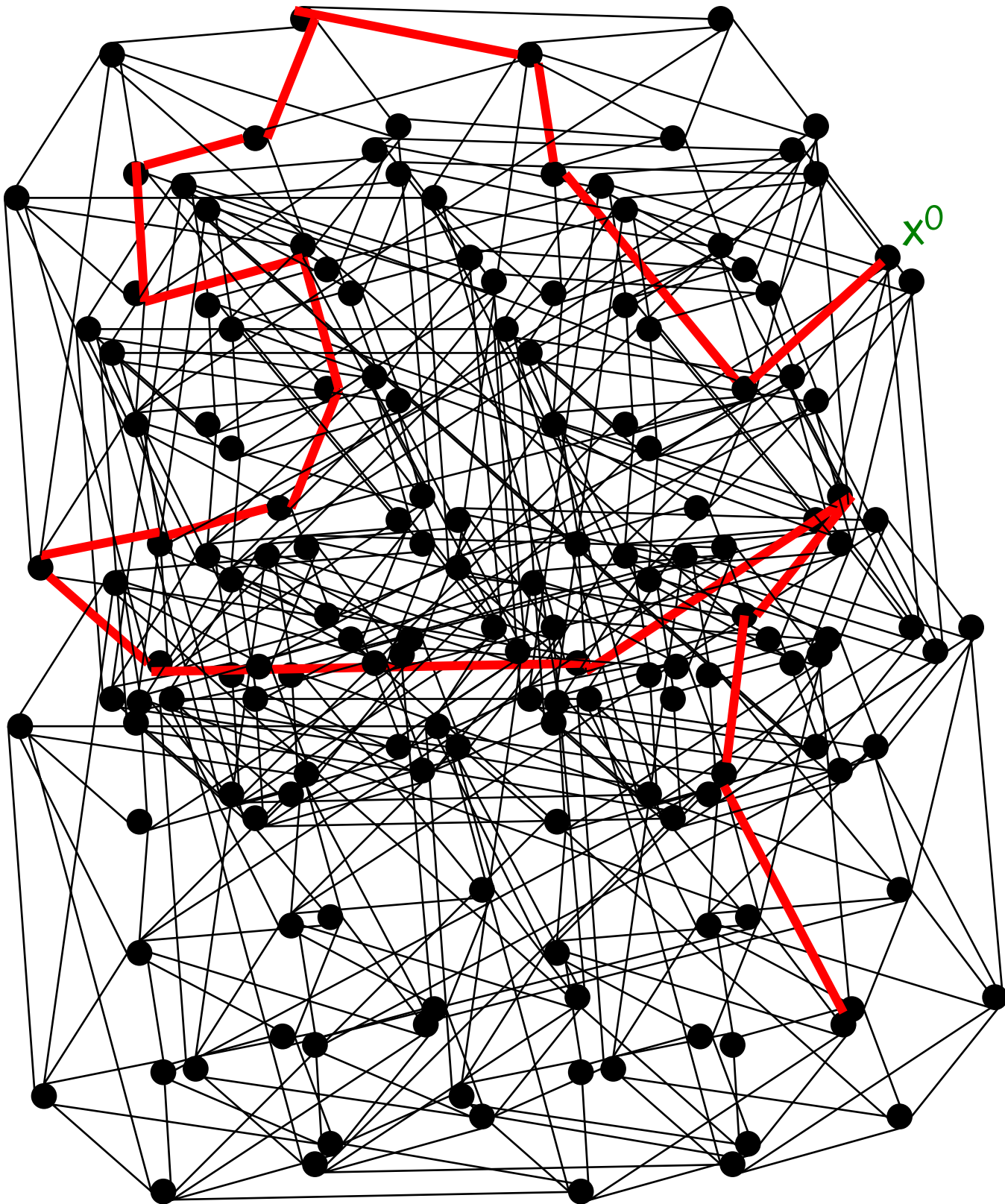




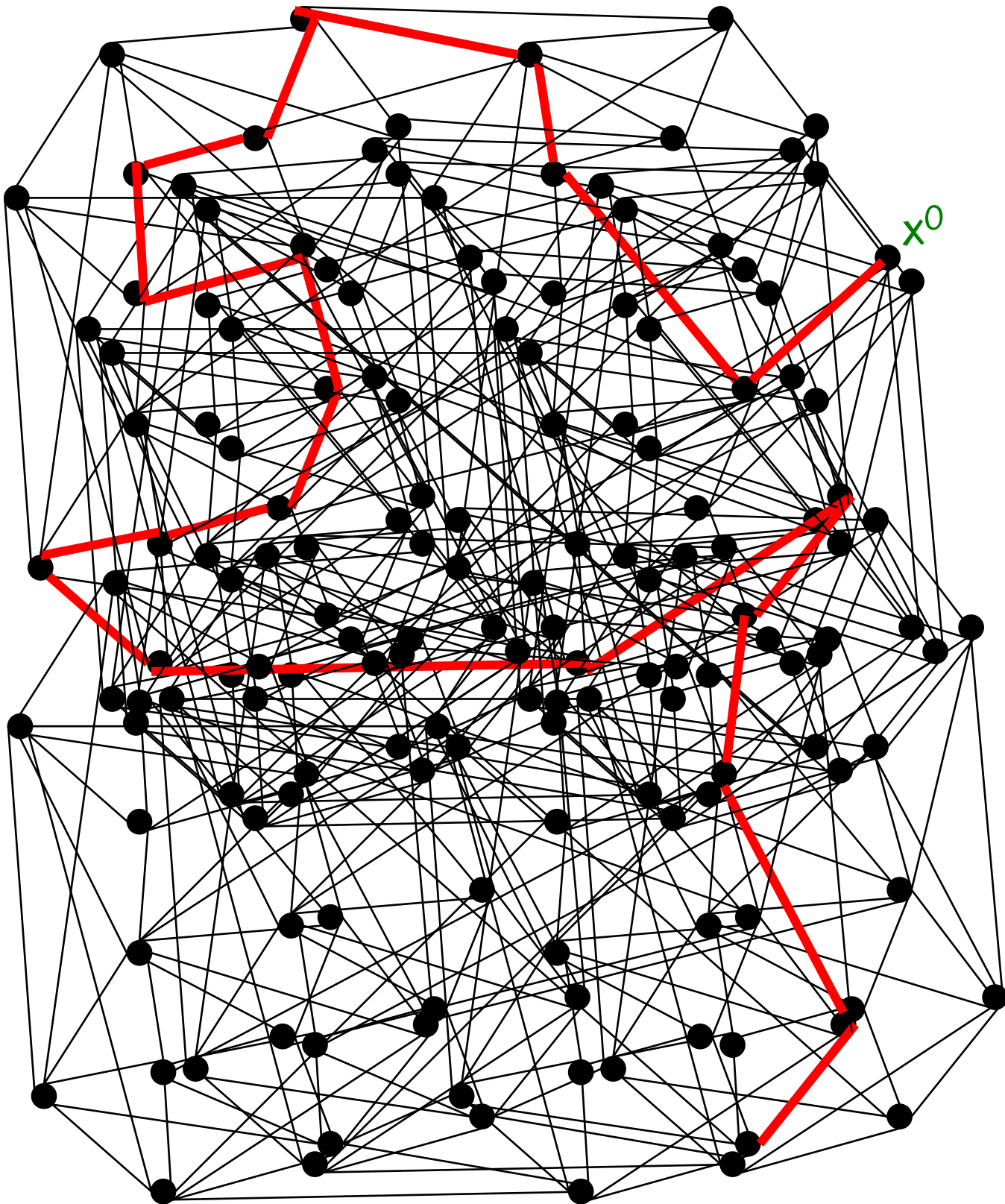
# Métodos de trajetória



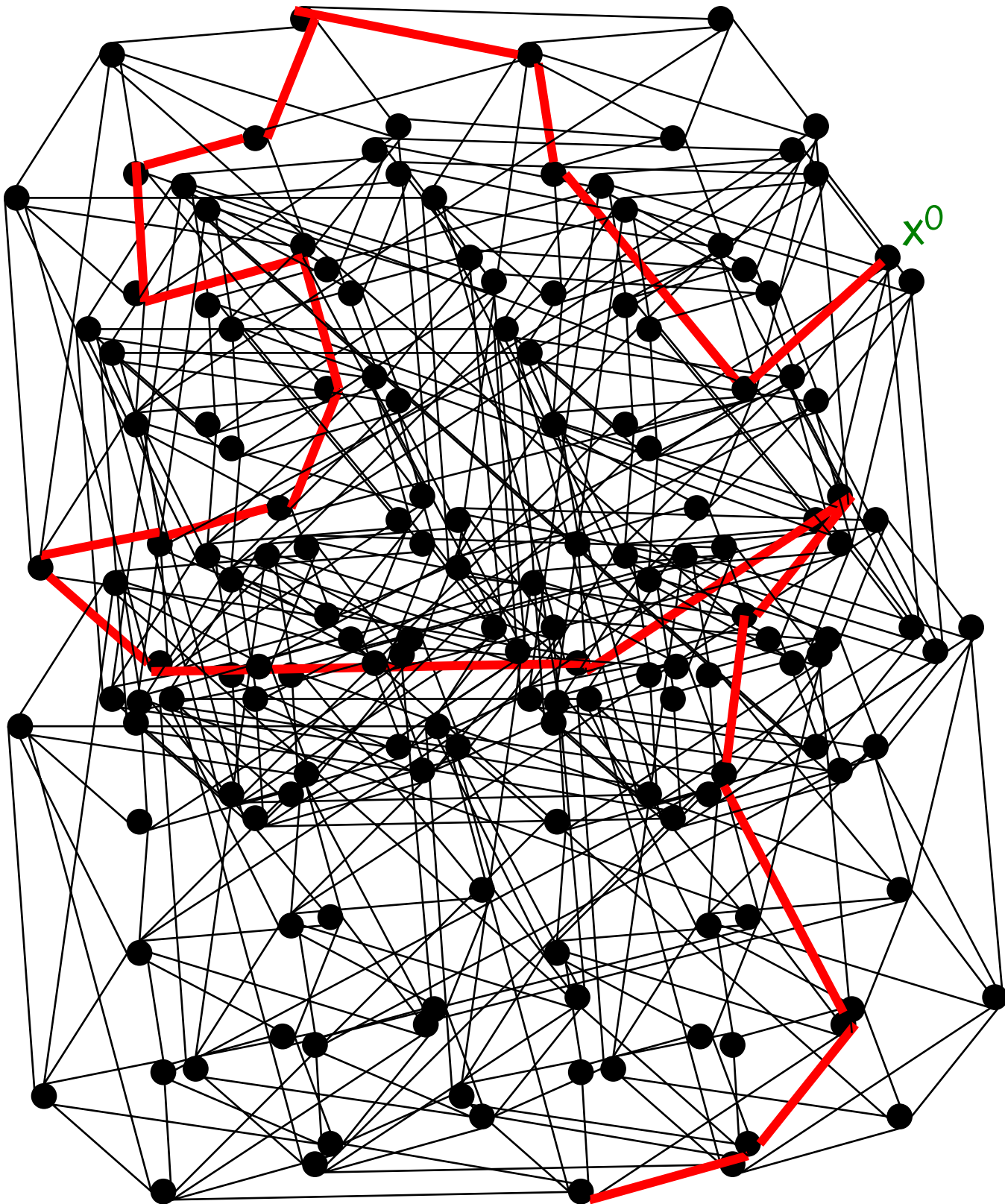
# Métodos de trajetória



# Métodos de trajetória

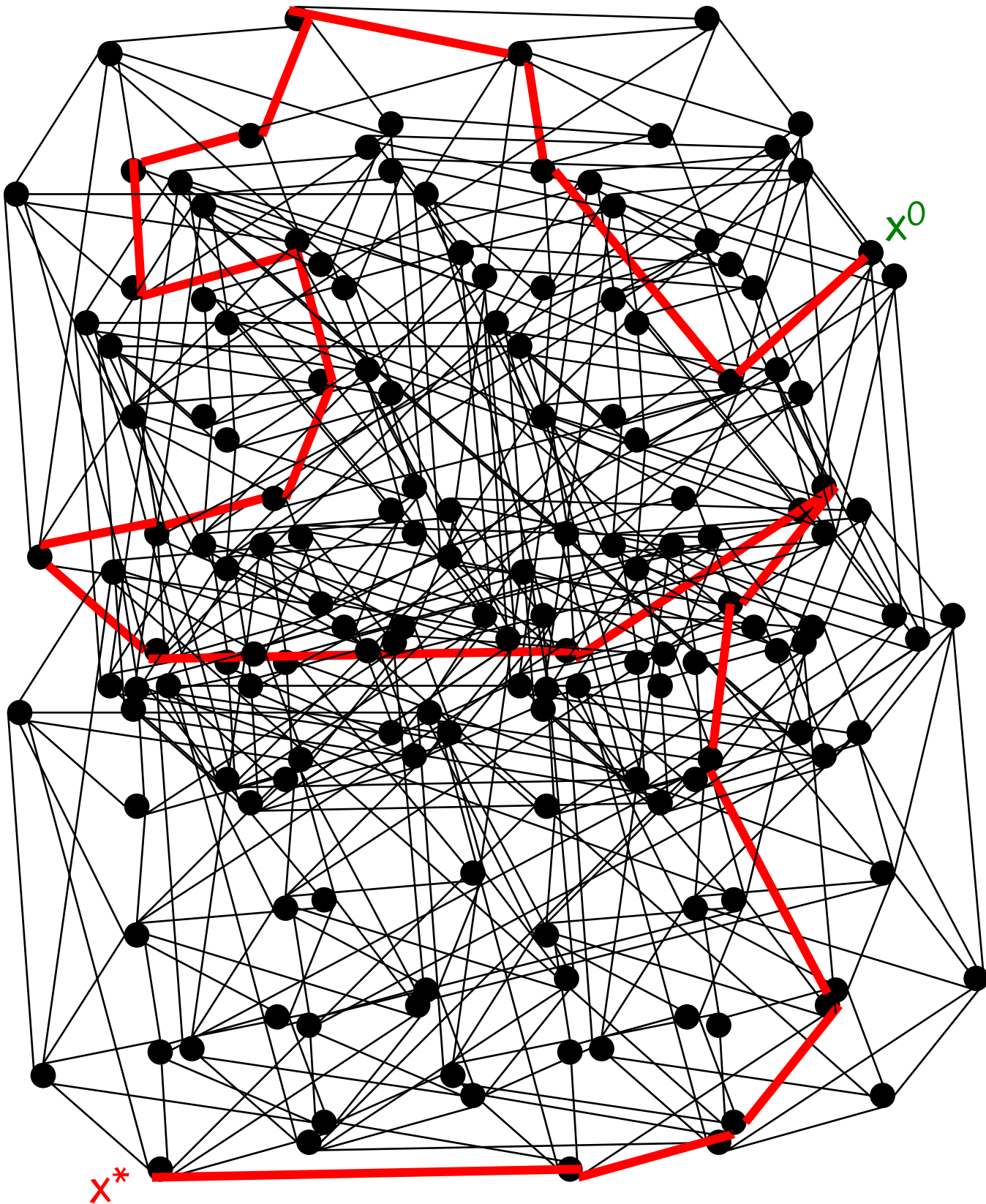


# Métodos de trajetória

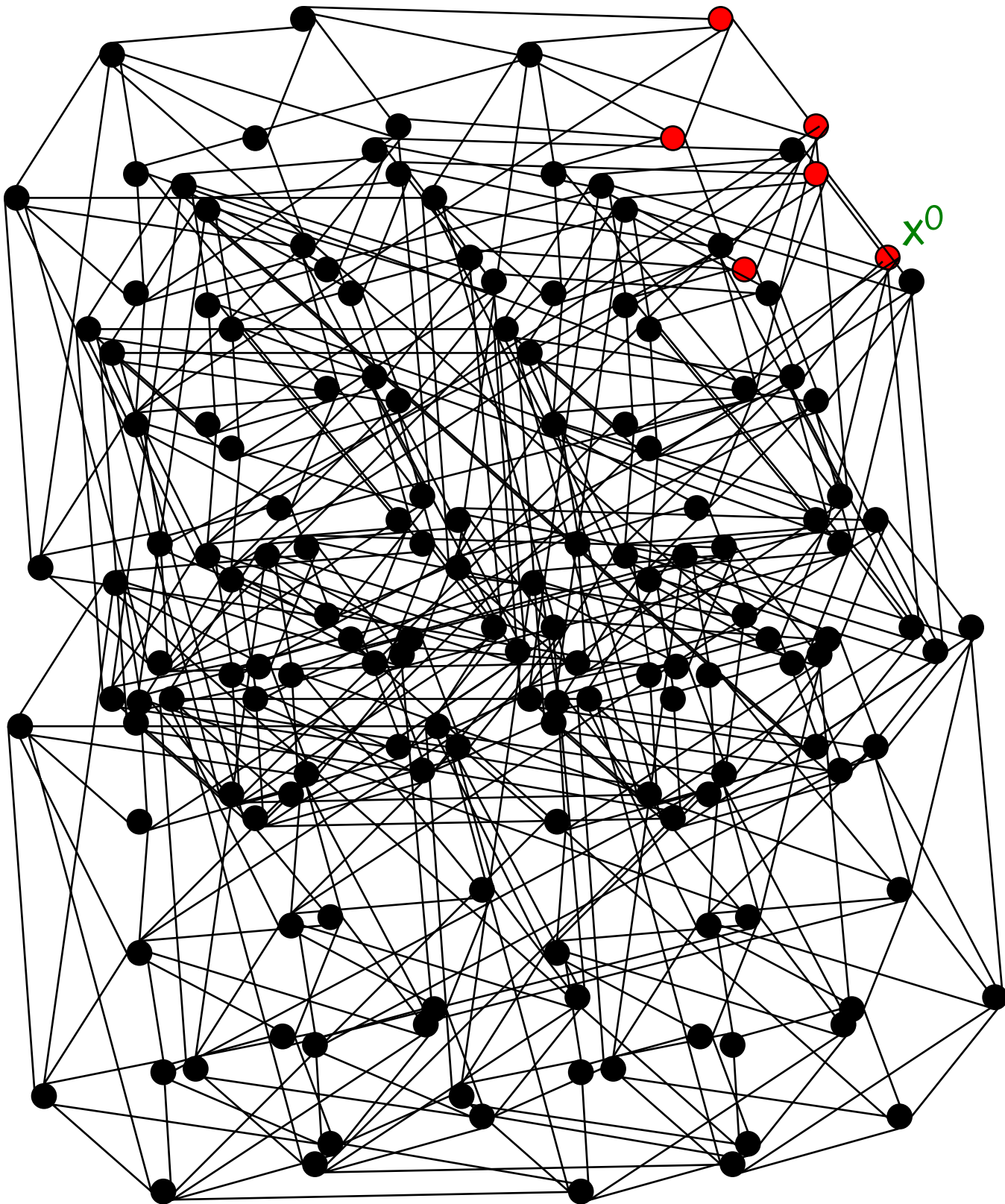




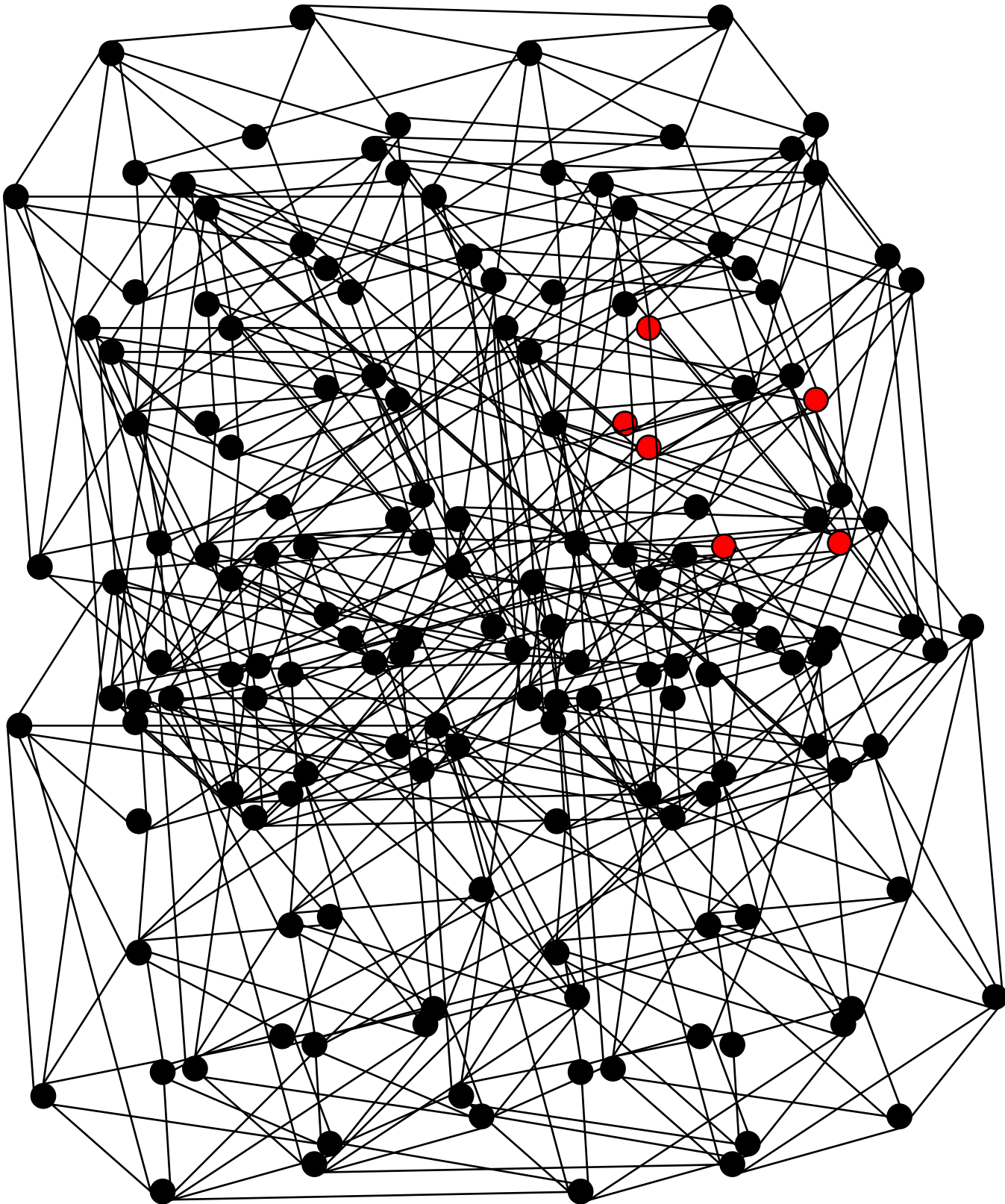
# Métodos de trajetória



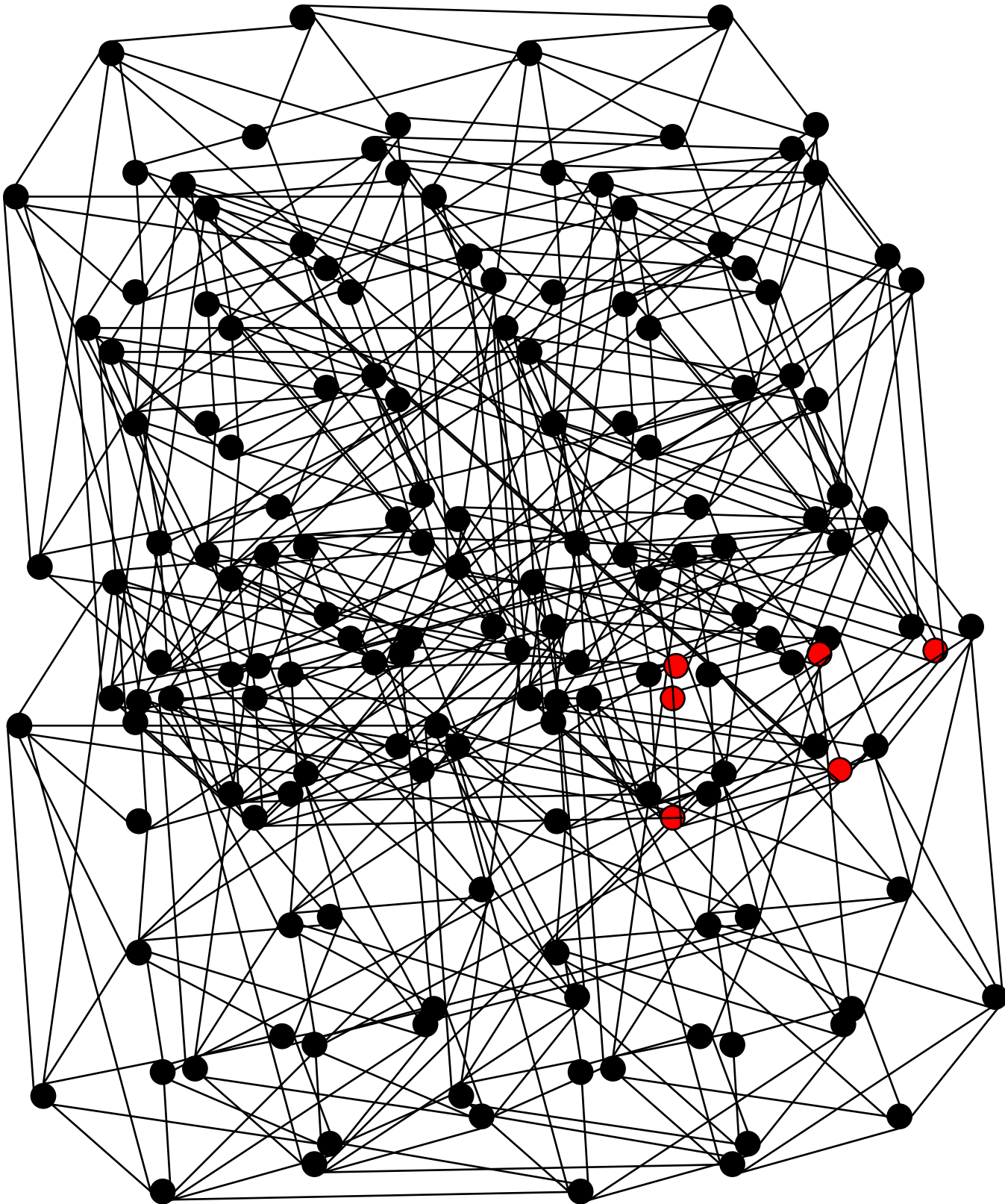
# Métodos populacionais



# Métodos populacionais

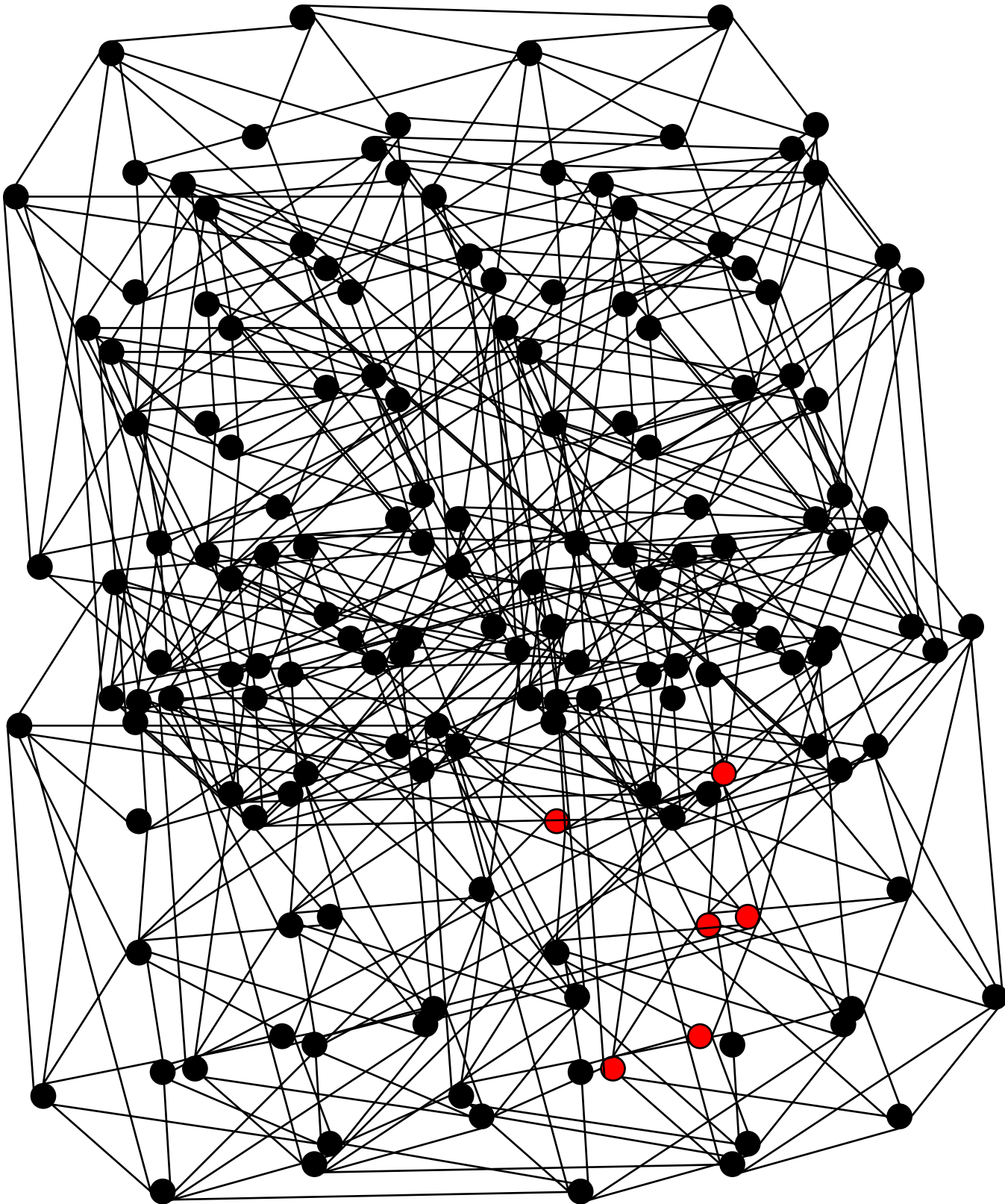


# Métodos populacionais

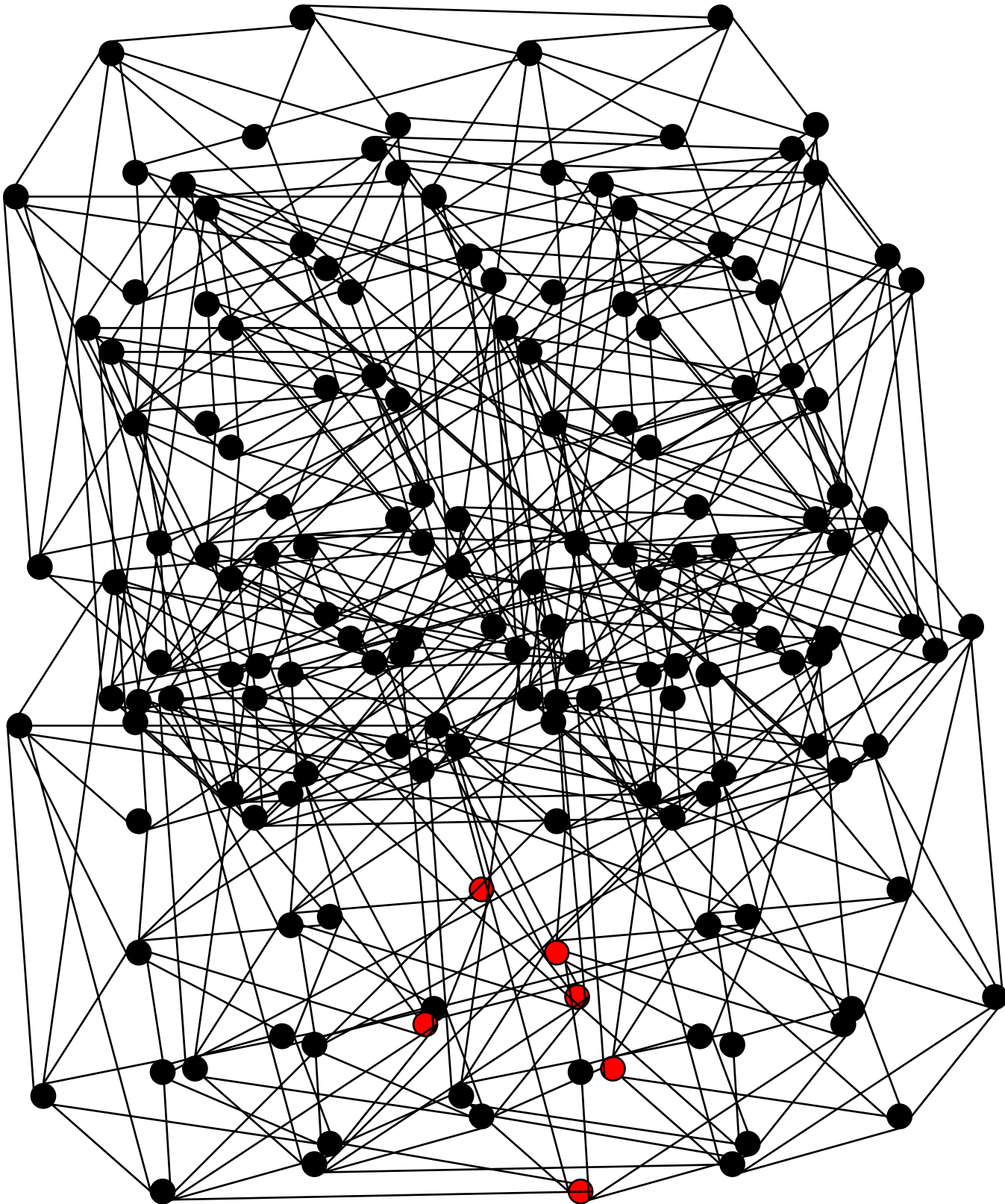




# Métodos populacionais



# Métodos populacionais



# Métodos populacionais

