Busca Local e Hill-Climbing

Alunos:Ícaro Peretti e Vinicius Gazolla Boneto

Algoritmo de Busca Local

- Resolver problemas de otimização
  - **Maximizar** ou **minimizar** uma função
- Em diversos problemas, o caminho até a solução é irrelevante
  - Design de chão de fábrica
    - Posicionar máquinas e peças
  - Design de circuitos integrados
    - Reduzir o tempo de comunicação
  - **n rainhas**
    - O importante é a configuração final das rainhas
- Mantém seu estado atual e tenta melhorá-lo

Algoritmo de Busca Local

- Dada uma solução inicial
  - Gerar novas soluções
- Maximizar a quantidade de alunos em disciplinas do curso
  - Quantidade de alunos cursando
  - Distribuir as disciplinas durante semana

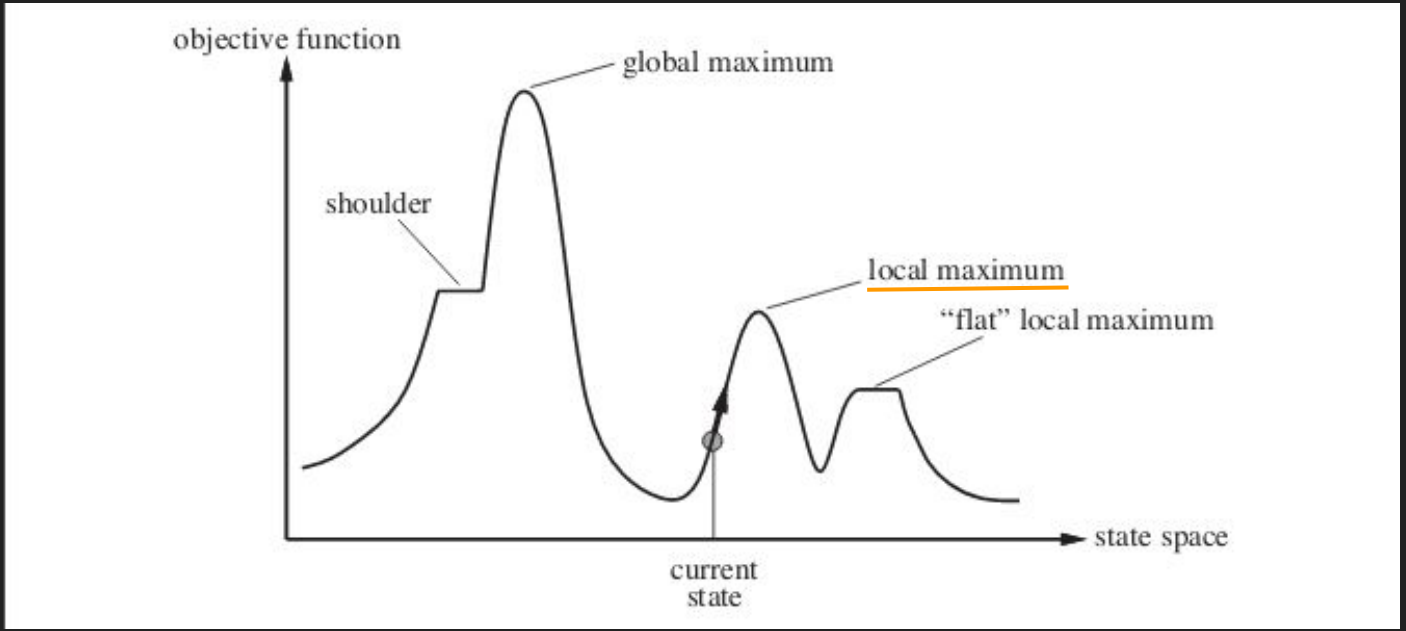
Algoritmo de Busca Local

- Se a busca está em um determinado estado
  - Só enxerga seus estados vizinhos
  - Não armazena o **caminho percorrido**
- O objetivo é escolher qual vizinho a busca irá seguir
  - Medir a qualidade da solução do estado vizinho
  - Mover-se para um estado **melhor** que o estado atual

Algoritmo de Busca Local

- Os algoritmos de busca local podem ser:
  - **Completos**
  - **Ótimos**
- **Vantagens:**
  - Baixa utilização de memória
    - Sempre no estado corrente e gerando seus sucessores
  - Encontram soluções **razoáveis** quando há uma **grande** ou **infinita** quantidade de estados

Algoritmo de Busca Local



https://miro.medium.com/max/1344/1*ymMUv1hzqBYPL-fErH7vIQ.png

- Pode ser que todos os estados vizinhos do estado atual sejam piores

Exemplo *n-rainhas*

- Coloque as *n* rainhas em um tabuleiro *n x n,* as rainhas não podem estar na mesma coluna, linha ou diagonal

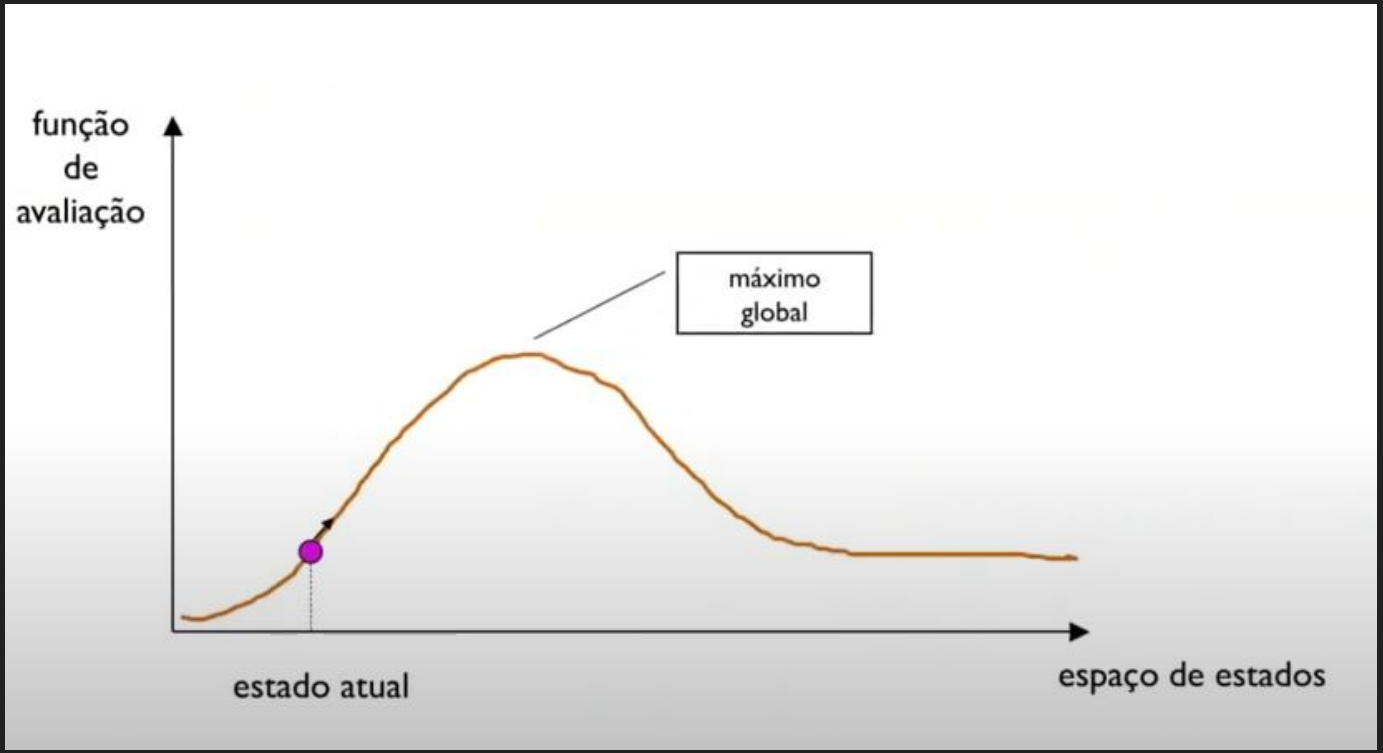
Hill-Climbing (Subida de Encosta)

- Um *loop* que se move sempre em direção de um valor crescente no espaço de estados.
- “É como usar óculos que te limitam a visualizar 3 metros”
  - “Escalar o monte Everest em um nevoeiro denso com amnésia”
  - Russell e Norvig
- O algoritmo **para** quando atinge o **“pico”**
- Observa somente vizinhos imediatos
- Só possui uma função de avaliação h(n), não é considerada a função de custo g(n)

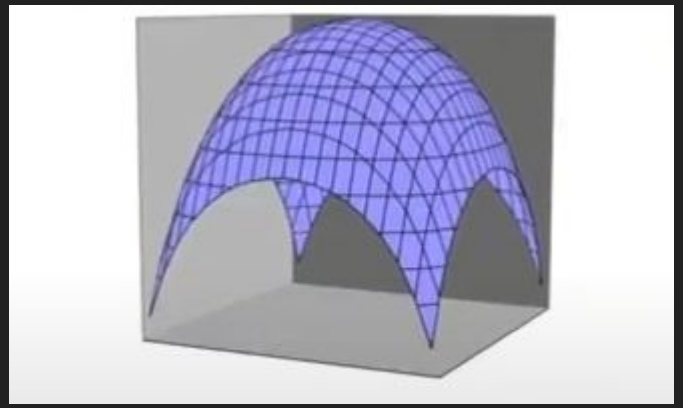
Hill-Climbing (Subida de Encosta)

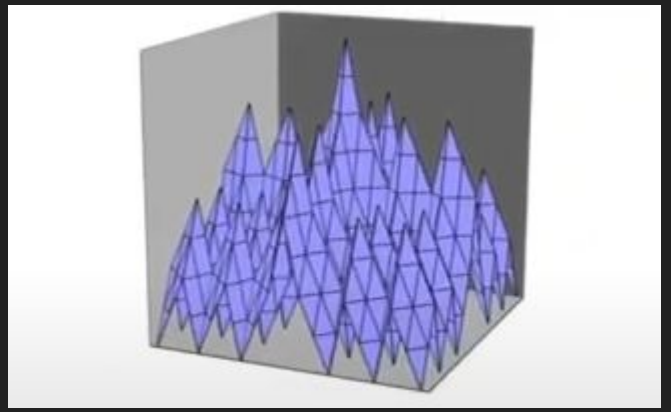
```
1  function HillClimbing(individual):Solution
2
3  newIndividual ← empty solution
4
5  begin
6  while(individual is not improved) do
7    newIndividual ← NeighbourhoodSelectionHeuristic(individual)
8    if(newIndividual is better than individual) then
9      individual ← newIndividual
10   end if
11 end while
12   return individual
13 end
```

Hill-Climbing Exemplo



Hill-Climbing (Subida de Encosta)





Hill-Climbing Problemas

- **Máximo local:** uma vez atingido, o algoritmo termina mesmo que a solução esteja longe de ser satisfatória

- **Platôs (região planas):** regiões onde a função de avaliação é essencialmente plana; a busca torna-se como uma caminhada aleatória

- **Cumes ou “ombros”:** regiões que são alcançadas facilmente mas até o topo a função de avaliação cresce de forma amena; a busca pode tornar-se demorada

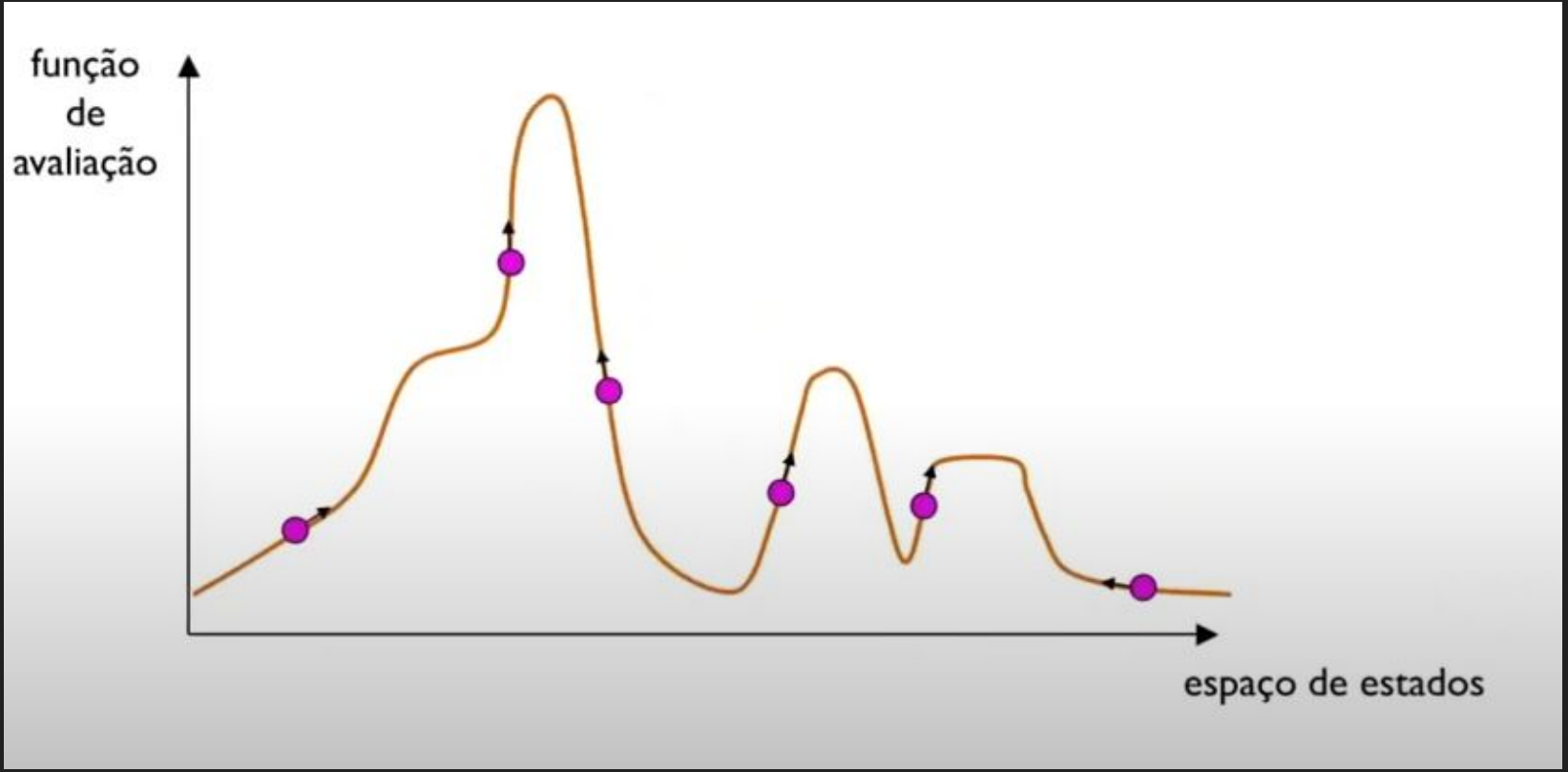
Hill-Climbing Variações

- **Estocástico:** nem sempre escolhe o melhor vizinho

- Primeira escolha: Escolha o primeiro bom vizinho que encontrar
  - Útil se é grande o número de sucessores de um nó

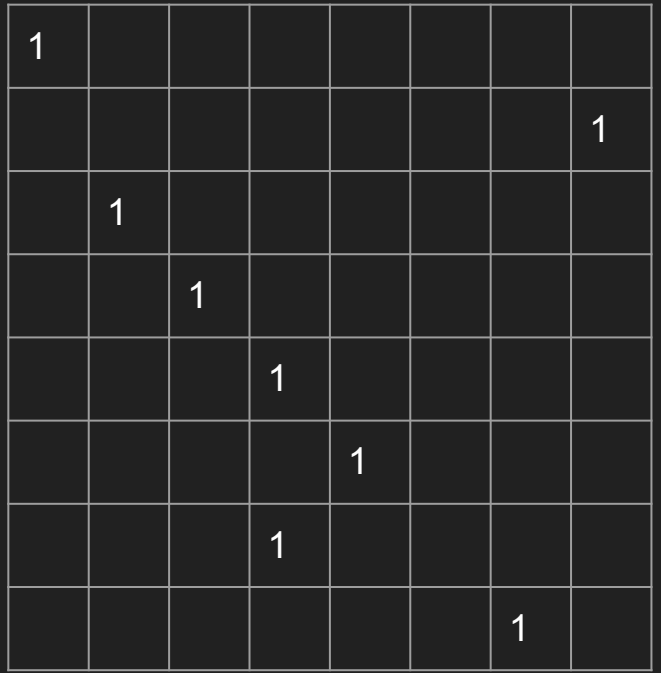
- Reinício aleatório: Conduz uma série de buscas hill climbing a partir de estados iniciais gerados aleatoriamente, executando cada busca até terminar ou até que não exista progresso significativo
  - O melhor resultado de todas a busca é armazenado

Hill-Climbing Variações



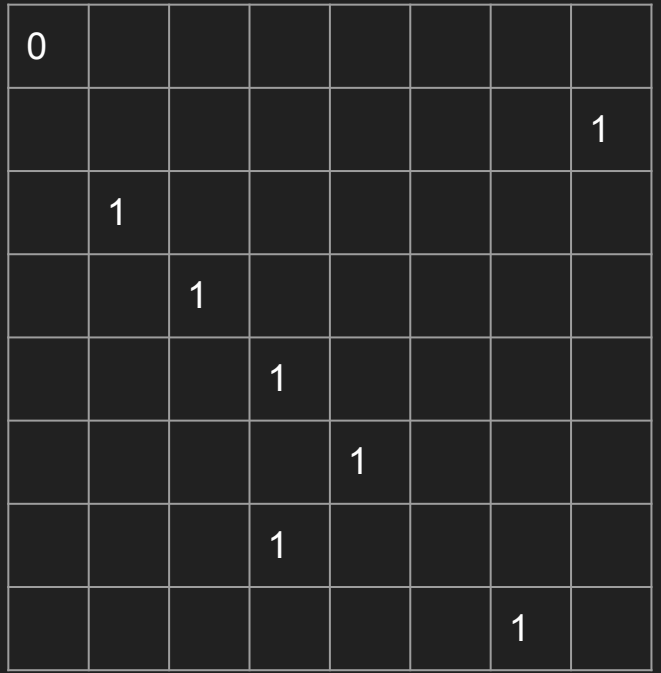
Implementação

```python
for i in range(8):
    # Find index of queen in current row
    queen = board[i].index(1)

    board[i][queen] = 0

    for k in range(8):
        # Place queen at different positions and calculate new score

        if k != queen:
            board[i][k] = 1

            h = heuristic_value(board)

            if h <= min_h:
                min_h = h
                min_board = copy.deepcopy(board)
                sideway_move = True

            board[i][k] = 0

    board[i][queen] = 1
```



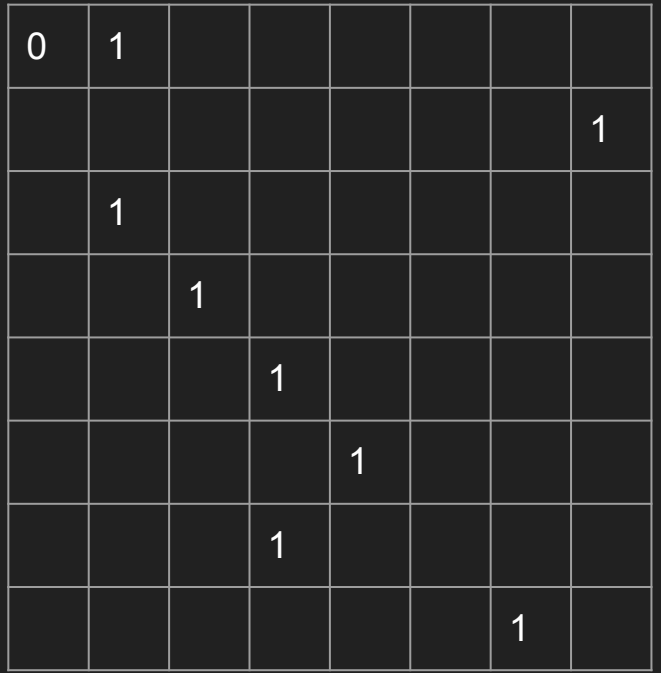
Implementação

```python
for i in range(8):
    # Find index of queen in current row
    queen = board[i].index(1)

    board[i][queen] = 0

    for k in range(8):
        # Place queen at different positions and calculate new score

        if k != queen:
            board[i][k] = 1

            h = heuristic_value(board)

            if h <= min_h:
                min_h = h
                min_board = copy.deepcopy(board)
                sideway_move = True

            board[i][k] = 0

    board[i][queen] = 1
```



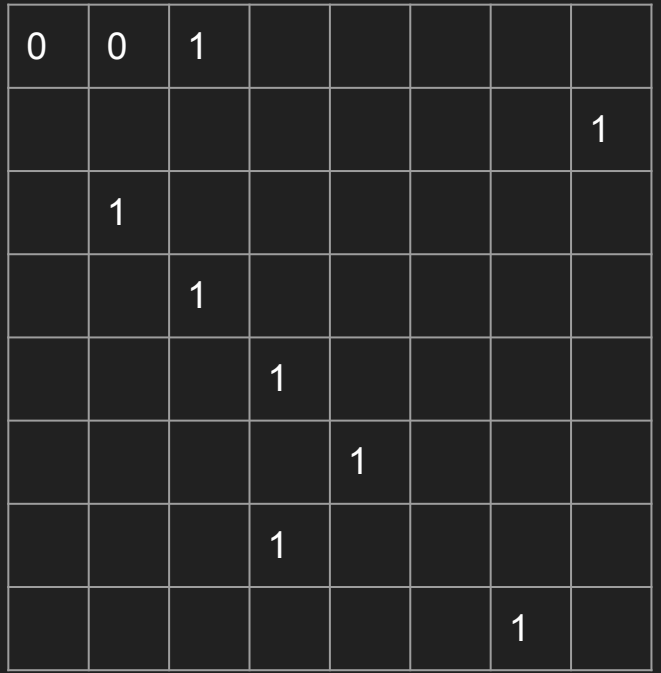
Implementação

```python
for i in range(8):
    # Find index of queen in current row
    queen = board[i].index(1)

    board[i][queen] = 0

    for k in range(8):
        # Place queen at different positions and calculate new score

        if k != queen:
            board[i][k] = 1

            h = heuristic_value(board)

            if h <= min_h:
                min_h = h
                min_board = copy.deepcopy(board)
                sideway_move = True

            board[i][k] = 0

    board[i][queen] = 1
```



Implementação

```python
for i in range(8):
    # Find index of queen in current row
    queen = board[i].index(1)

    board[i][queen] = 0

    for k in range(8):
        # Place queen at different positions and calculate new score

        if k != queen:
            board[i][k] = 1

            h = heuristic_value(board)

            if h <= min_h:
                min_h = h
                min_board = copy.deepcopy(board)
                sideway_move = True

            board[i][k] = 0

    board[i][queen] = 1
```



Implementação

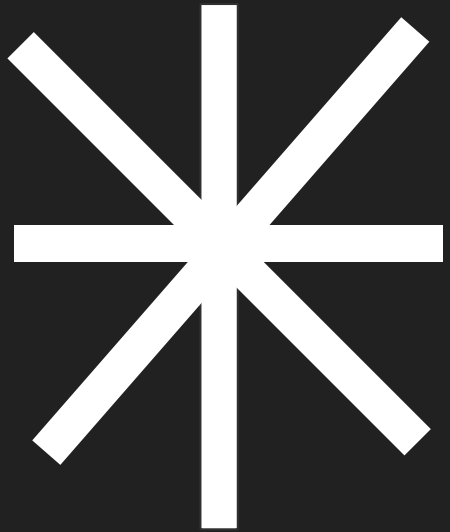
```
if min_h == 0:
    print("Number of steps required: {}".format(n_steps))
    return min_board

return hill_climbing(min_board)
```

Implementação

```
# Check if number of side moves has reached a limit
if n_side_moves == 100:
    return min_board

n_steps += 1
```

Implementação

```python
# Calculate horizontal attacks
for k in range(8):
    if board[i][k] == 1 and k != j and not contains(i, j, i, k, queen_pairs):
        queen_pairs.append((i, j, i, k))
        h += 1

# Calculate vertical attacks
for k in range(8):
    if board[k][j] == 1 and i != k and not contains(i, j, k, j, queen_pairs):
        queen_pairs.append((i, j, k, j))
        h += 1
```



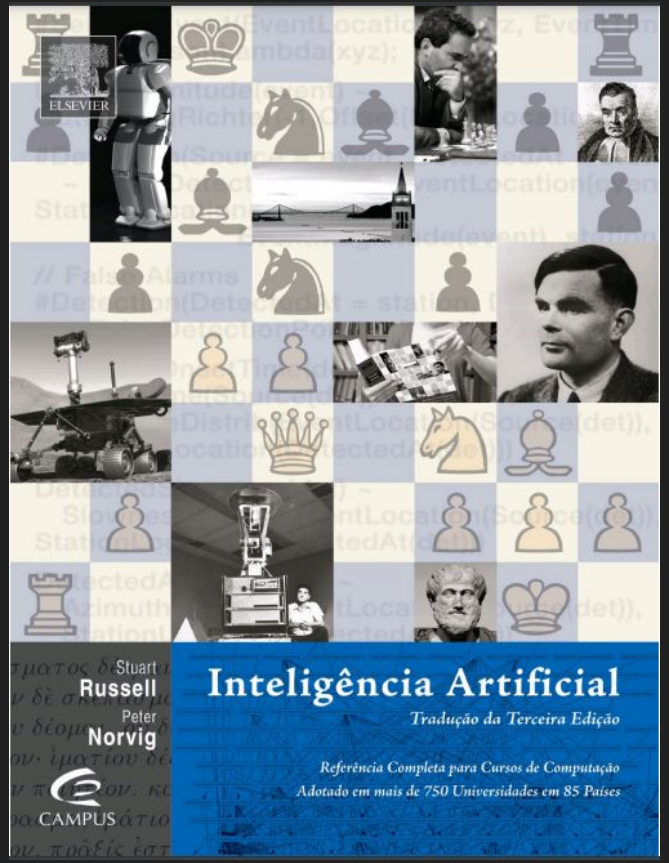
Outros algoritmos de busca local

- **Têmpera simulada**
- **Busca em feixe local**
- **Algoritmos genéticos**

Algoritmo do problema das n-rainhas

- https://colab.research.google.com/drive/1CnaTOCfYyxgr3SyXS8Rf2ApkdqRiG5lO?usp=sharing#scrollTo=nB9FhL8iPick

Referências



Obrigado!