Hill Climbing & Simulated Annealing

Rio de Janeiro - RJ

Universidade Federal do Rio de Janeiro
Breno Pontes da Costa - 114036496
Xiao Yong Kong - 114176987

1 - Modelagem

Alguns códigos abaixo usam funções de bibliotecas nativas do python

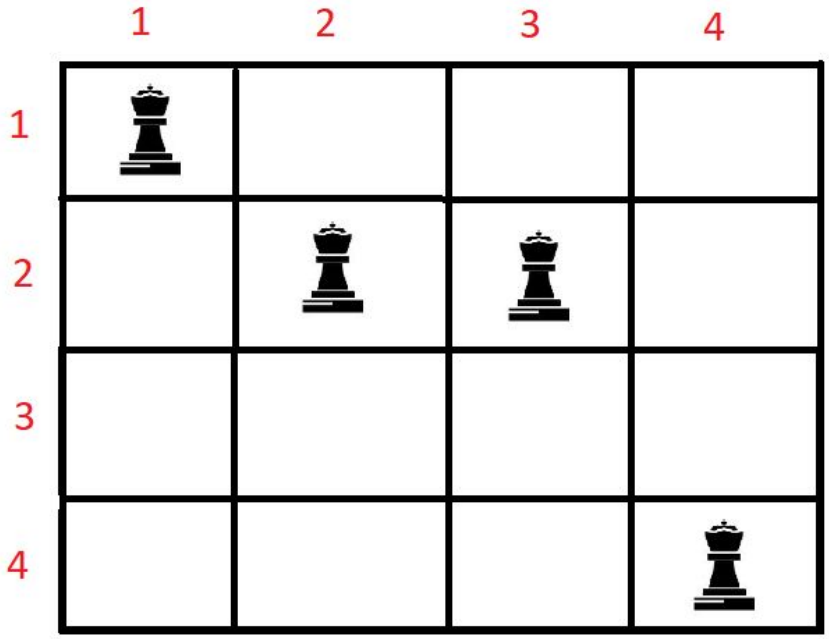
```
import sys
from random import randint
from random import uniform
from math import exp
```

a)

O tabuleiro será representado por uma lista de tamanho = “numeroDeRainhas”. Cada elemento da lista pode assumir valores inteiros entre 0 e “numeroDeRainhas”.

Exemplo:

[1,2,2,4] -> Um tabuleiro 4x4 onde a primeira rainha está na posição 1, a segunda e a terceira estão na posição 2 no tabuleiro e a quarta está na posição 4.



O parâmetro citado será melhor explicado nas seguintes sessões.

2 - Implementação

a) A função responsável por criar o tabuleiro aleatório N x N será a criaTabuleiro(numeroDeRainhas).

```
def criaTabuleiro(numeroDeRainhas):
        tabuleiro = []
        for i in range(0, numeroDeRainhas):
            tabuleiro.append(randint(1, numeroDeRainhas))
        print("O tabuleiro criado foi:")
        print(tabuleiro)
        return tabuleiro
```

b)

A função que retorna todos os vizinhos dado um tabuleiro será:

```
# Busca todos os vizinhos do tabuleiro em qustão
def recuperaVizinhosDeTabulero(tabuleiro):
    vizinhosDeTabuleiro = []

    for identificadorDaRainha in range (0, len(tabuleiro)):
        for posicaoDaRainha in range (1, len(tabuleiro)+1):

            if posicaoDaRainha == tabuleiro[identificadorDaRainha] :
                continue

            vizinho = tabuleiro.copy()
            vizinho[identificadorDaRainha] = posicaoDaRainha
            vizinhosDeTabuleiro.append(vizinho)

    print("Nesta iteração, o tabuleiro possui os seguintes vizinhos")
    print(vizinhosDeTabuleiro)

    return vizinhosDeTabuleiro
```

O primeiro loop com parâmetro de iteração *identificadorDaRainha* percorre todas as rainhas do tabuleiro. Já o segundo loop com parâmetro *posicaoDaRainha* cria um novo tabuleiro possível a cada iteração. Esta função retorna todos os tabuleiros vizinhos possíveis dado um tabuleiro.

c)

```
#Retorna um vizinho alearório dado um tabuleiro
def buscaVizinhoAleatorio(tabuleiro):
    vizinhos = recuperaVizinhosDeTabulero(tabuleiro)
    vizinhoAleatorio = vizinhos[randint(0,len(tabuleiro))]
    print("O vizinho aleatório retornado será:")
    print(vizinhoAleatorio)
    return vizinhoAleatorio
```

Usa a função desenvolvida na letra b para retornar um vizinho aleatório a partir da função randint.

d)

A heurística será o número de ataques possíveis no tabuleiro. Sendo assim, foram implementadas funções para os seguintes casos:

- Buscar ataques na horizontal

```
# Para cada rainha no tabuleiro, busca quantos ataques são possíveis na horizontal
def buscarNumeroDeAtaquesNaHorizontal(tabuleiro):
    return buscarNumeroDeAtaquesNaDireita(tabuleiro)

# Para cada rainha no tabuleiro, busca quantos ataques são possíveis
def buscarNumeroDeAtaquesNaDireita(tabuleiro):
    numeroDeAtaques = 0

    for posicaoRainha in range (0,len(tabuleiro)):
        for possivelAtaqueNaDireita in range (posicaoRainha,len(tabuleiro)):
            if possivelAtaqueNaDireita == posicaoRainha:
                continue

            elif tabuleiro[posicaoRainha] == tabuleiro[possivelAtaqueNaDireita]:
                numeroDeAtaques += 1
                break
    return numeroDeAtaques
```

- Buscar ataques no nordeste

```python
# Analisa o tabuleiro no sentido nordeste de cada rainha, contando o número de ataques possíveis
def buscarNumeroDeAtaquesNoNordeste(tabuleiro):
    numeroDeAtaques = 0

    for posicaoRainha in range (0,len(tabuleiro)):
        mapeiaDiagonal = tabuleiro[posicaoRainha]
        for posicaoHorizontal in range (posicaoRainha + 1, len(tabuleiro)):
            mapeiaDiagonal -= 1

            if tabuleiro[posicaoHorizontal] == mapeiaDiagonal:
                numeroDeAtaques += 1
                break
    return numeroDeAtaques
```

- Buscar ataques no sudeste

```python
# Analisa o tabuleiro no sentido sudeste de cada rainha, contando o número de ataques possíveis
def buscarNumeroDeAtaquesNoSudeste(tabuleiro):
    numeroDeAtaques = 0

    for posicaoRainha in range (0,len(tabuleiro)):
        mapeiaDiagonal = tabuleiro[posicaoRainha]
        for posicaoHorizontal in range (posicaoRainha + 1, len(tabuleiro)):
            mapeiaDiagonal += 1

            if tabuleiro[posicaoHorizontal] == mapeiaDiagonal:
                numeroDeAtaques += 1
                break
    return numeroDeAtaques
```

O resultado de cada função será somado e assim será gerado o número de ataques totais de um tabuleiro. Essa soma é feita pela função *buscarNumeroDeAtaquesNoTabuleiro.*

```python
# Como a heurística é baseada em números de ataques. Esta função retorna quantos ataques são possíveis no tabuleiro
def buscarNumeroDeAtaquesNoTabuleiro(tabuleiro):
    numeroDeAtaques = buscarNumeroDeAtaquesNaHorizontal(tabuleiro) + buscarNumeroDeAtaquesNoNordeste(tabuleiro) + buscarNumeroDeAtaquesNoSudeste(tabuleiro)
    print(numeroDeAtaques)
    return numeroDeAtaques
```

2 - Implementação

O algoritmo principal do Hill climbing será:

```python
#Simula o problema de N_Rainhas usando Hill-climbing
def simuladorProblemaN_RainhasComHillClimbing(numeroDeRainhas):
    tabuleiro = criaTabuleiro(numeroDeRainhas)

    tabuleiroAtual = tabuleiro
    heuristicaAtual = sys.maxsize
    contadorDeHeuristicaRepetida = 0
    while(True):

        melhorVizinhoComHeuristica = buscarMelhorVizinhoAleatorio(tabuleiroAtual, recuperaVizinhosDeTabulero(tabuleiroAtual))
        if heuristicaAtual == melhorVizinhoComHeuristica[1]:
            contadorDeHeuristicaRepetida += 1

        if heuristicaAtual > melhorVizinhoComHeuristica[1]:
            contadorDeHeuristicaRepetida = 0
            tabuleiroAtual = melhorVizinhoComHeuristica[0]
            heuristicaAtual = melhorVizinhoComHeuristica[1]

        if contadorDeHeuristicaRepetida > 10:
            print("foi encontrado um ombro, terminando a execução...")
            break

        if heuristicaAtual == 0 :
            print("SOLUÇÃO ENCONTRADA")
            print("A solução do problema será:")
            print(tabuleiroAtual)
            break

    return tabuleiroAtual
```

a) A função que retorna o melhor vizinho aleatório será:

```python
#Entre um conjunto de tabuleiro, retorna o tabuleiro com a melhor heurística
def buscarMelhorVizinhoAleatorio( tabuleiro, vizinhos ):
    melhorVizinho = tabuleiro
    melhorHeuristica = buscarNumeroDeAtaquesNoTabuleiro(tabuleiro)

    melhoresVizinhos = []

    for vizinho in vizinhos :
        heuristicaVizinho = buscarNumeroDeAtaquesNoTabuleiro(vizinho)
        if heuristicaVizinho < melhorHeuristica:
            melhorHeuristica = heuristicaVizinho
            melhorVizinho = vizinho
            melhoresVizinhos = []

        if  heuristicaVizinho == melhorHeuristica :
            melhoresVizinhos.append([vizinho,heuristicaVizinho])

    melhorVizinhoAleatorio = melhoresVizinhos[randint(0, len(melhoresVizinhos) - 1)]

    print("O melhor vizinho encontrado foi:")
    print(melhorVizinhoAleatorio[0])
    print("Com a heurística de valor: ")
    print(melhorVizinhoAleatorio[1])
    return [melhorVizinhoAleatorio[0], melhorVizinhoAleatorio[1]]
```

b) Na função principal do hill climbing, é possível substituir a função buscar melhor vizinho aleatório pela função buscarPrimeiroMelhorVizinho.

```python
#Entre um conjunto de tabuleiro, retorna o tabuleiro com a melhor heurística
def buscarPrimeroMelhorVizinho(tabuleiro, vizinhos ):
    melhorVizinho = tabuleiro
    melhorHeuristica = buscarNumeroDeAtaquesNoTabuleiro(tabuleiro)

    for vizinho in vizinhos :
        heuristicaVizinho = buscarNumeroDeAtaquesNoTabuleiro(vizinho)
        if heuristicaVizinho < melhorHeuristica:
            melhorHeuristica = heuristicaVizinho
            melhorVizinho = vizinho
            break

    print("O primeiro melhor vizinho encontrado foi:")
    print(melhorVizinho)
    print("Com a heurística de valor: ")
    print(melhorHeuristica)
    return [melhorVizinho, melhorHeuristica]
```

c)

Os dados abaixo foram obtidos a partir de 100 execuções de cada implementação.

Para a implementação de melhor vizinho aleatório:

1. **N = 4:**

46 sucessos	90,84 tabuleiros gerados em média

2. **N = 8:**

9 sucessos	727,44 tabuleiros gerados em média

3. **N = 16:**

2 sucessos	4144,80 tabuleiros gerados em média

4. **N = 32:**

0 sucessos	61248,60 tabuleiros gerados em média

Para a implementação do primeiro melhor vizinho?

1. **N = 4**:

32 sucessos	11076 tabuleiros gerados

2. **N = 8:**

19 sucessos	77112 tabuleiros gerados

3. **N = 16:**

4 sucessos	510480 tabuleiros gerados

4. **N = 32:**

0 sucessos	52468,30 tabuleiros gerados em média

d) É possível observar que a implementação do primeiro melhor vizinho possui uma execução mais rápida e pode levar a resultados melhores, uma vez que não é percorrida toda a lista de vizinhos em cada execução, porém, é garantido que o novo tabuleiro corrente terá uma heurística melhor ou igual a do tabuleiro anterior.

4 - Simulated Annealing

a) O algoritmo implementado foi:

```python
#Simula o problema de N_Rainhas usando SimulatedAnnealing
def simuladorProblemaN_RainhasSimulatedAnnealing(numeroDeRainhas, maximoDeIteracoes,temperaturaInicial, alpha):
    tabuleiro = criaTabuleiro(numeroDeRainhas)

    tabuleiroAtual = tabuleiro
    solucao = tabuleiroAtual

    temperaturaAtual = temperaturaInicial

    totalDeVizinhosGerados = 0

    for i in range(1,maximoDeIteracoes):
        if(temperaturaAtual <= 0):
            break

        vizinhoAleatorio = buscaVizinhoAleatorio(solucao)
        heuristicaVizinhoAleatorio = buscarNumeroDeAtaquesNoTabuleiro(vizinhoAleatorio[0])

        diferrencaDeCusto =  heuristicaVizinhoAleatorio - buscarNumeroDeAtaquesNoTabuleiro(tabuleiroAtual)

        totalDeVizinhosGerados += vizinhoAleatorio[1]

        if(diferrencaDeCusto < 0):
            tabuleiroAtual = vizinhoAleatorio[0]
            if(buscarNumeroDeAtaquesNoTabuleiro(vizinhoAleatorio[0]) <= buscarNumeroDeAtaquesNoTabuleiro(solucao)):
                solucao = vizinhoAleatorio[0]
                if(heuristicaVizinhoAleatorio == 0 ):
                    break
        else:
            if(uniform(0,1) < exp(-(diferrencaDeCusto / temperaturaAtual))):
                tabuleiroAtual = vizinhoAleatorio[0]
        temperaturaAtual = temperaturaAtual * alpha

    print("A solução encontrada foi:")
    print(solucao)
    print("Com a heurística: ")
    print(buscarNumeroDeAtaquesNoTabuleiro(solucao))
    return [solucao, totalDeVizinhosGerados]
```

b) Nas primeiras execuções foram passados os parâmetros:
maximoDeIteracoes = 100
temperaturaInicial = 9
alpha = 0.9

Em 20 execuções, foi possível encontrar 6 soluções.

Apesar da alta probabilidade de sucessos por execuções acima, não foi possível encontrar parâmetros exatos que sempre levam há uma execução. Porém, é possível afirmar que quanto maior a quantidade de iterações, maior a probabilidade de se chegar em uma solução.

5 - Conclusão

O Simulated Annealing é interessante pois é possível ajustar os parâmetros de entrada de acordo com o problema a ser resolvido. Em alguns casos, não será necessário um grande número de iterações até chegar em uma solução.

O Simulated annealing executa em direção de uma solução sempre, já o hill climbing pode ficar parado em um mínimo local (que pode estar longe de uma solução).

De qualquer forma, para problemas de entradas menores que possibilitam uma rápida execução, Hill Climbing com escolha do primeiro melhor vizinho é interessante.