INSTITUTO TECNOLÓGICO DE AERONÁUTICA - ITA

DIVISÃO DE CIÊNCIA DA COMPUTAÇÃO

CTC-17 – Inteligência Artificial

Projeto II – Buscas de melhoria iterativa e PSR

- **Problema das N-Rainhas**
- **Máximo global de função**
- **Problema dos vizinhos de Einstein**

ALUNO

Felipe Tuyama de Faria Barbosa ftuyama@gmail.com

PROFESSOR

Paulo André Lima de Castro pauloac@ita.br

São José dos Campos, 19 de Setembro de 2016

1. INTRODUÇÃO

O objetivo deste projeto é implementar algoritmos de inteligência artificial envolvendo busca de melhoria iterativa e problemas de satisfação de restrição. Em um primeiro momento, é explorado o problema das N-Rainhas, em seguida, temos o problema de encontrar o máximo global de uma função e por fim o problema de satisfação de restrições dos vizinhos de Einstein (5 vizinhos com animais, bebidas, cigarros, cores de casa e nacionalidades diferentes).

As soluções foram implementadas na linguagem de alto nível Python 2.7, devido à facilidade de prototipação conferida pela linguagem.

2. DESCRIÇÃO

O problema das N-Rainhas foi resolvido com uma GUI (interface gráfica) a fim de mostrar a situação/solução do problema. Foi usada a biblioteca graphics para tal, podendo a aplicação ser executada diretamente do código fonte.

O problema das N-Rainhas consiste em dispor N Rainhas em um tabuleiro de xadrez de N linhas e N colunas de modo que nenhuma peça ataque a outra. Foi utilizada a abordagem de melhoria iterativa para resolver o problema, com o algoritmo de hill climbing.

O problema de encontrar o máximo de função é clássico na matemática. Foi utilizado também uma abordagem de busca com melhoria iterativa, com o algoritmo da têmpera simulada.

O problema dos vizinhos de Einstein consiste em dispor os 5 vizinhos com seus animais, bebidas, cigarros, cores de casa e nacionalidades diferentes em casas de uma mesma vizinhança, de modo que satisfaçam a um conjunto de requisitos. Pergunta-se ao final qual vizinho bebe água e qual vizinho tem uma zebra como animal de estimação.

O PSR foi resolvido usando duas modelagens distintas, a primeira usando Orientação a Objetos e outra usando metodologia procedural.

3. IMPLEMENTAÇÃO

A implementação dos problemas teve sempre algo em comum: a tentativa de se otimizar uma certa função, recorrendo a alguma heurística para avaliar a melhor ação a se tomar para chegar à solução o mais rápido possível.

3.1 Implementação das N-Rainhas

O problema das N-Rainhas foi modelado como um problema de minimizar a função correspondente ao número de violações encontradas (ou número de ataques entre rainhas). Usando essa abordagem, foi possível aplicar o algoritmo de hill-climbing para achar o mínimo desta função, correspondente à solução do problema. Na Figura abaixo temos uma possível solução encontrada:

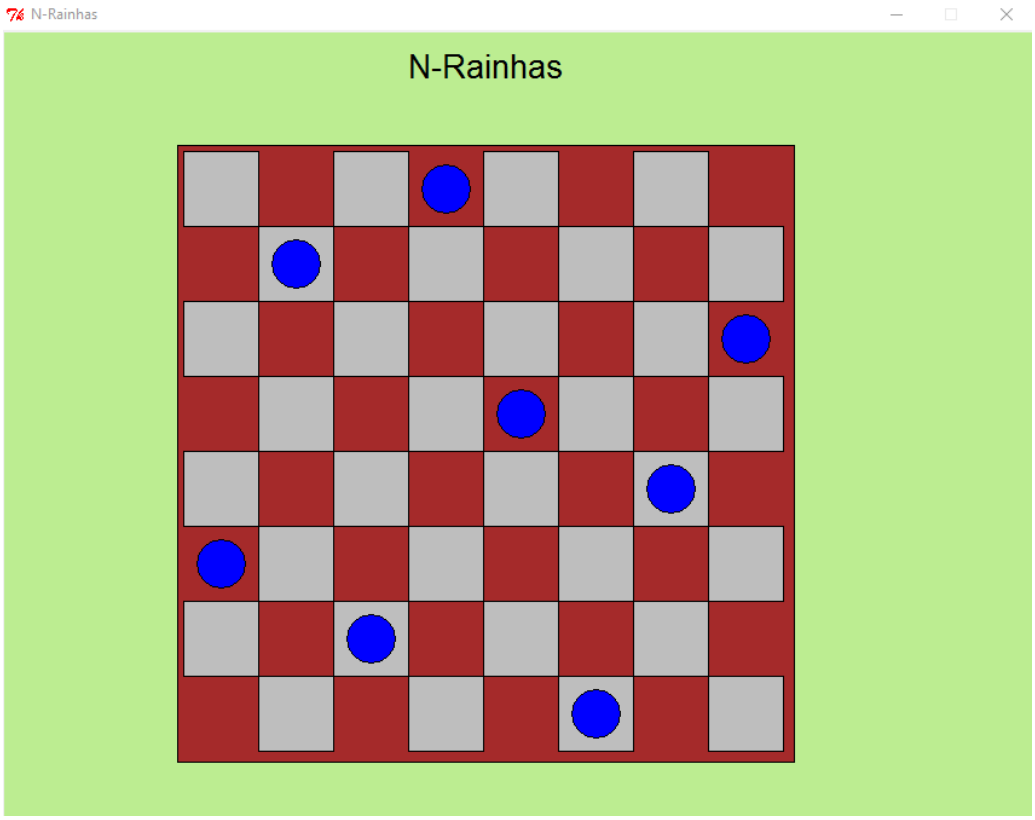

Figura 1 – Output da solução do problema das N-Rainhas.

O problema das N-Rainhas é clássico por chegar a condições de "deadlock" ou mínimos locais, onde na iminência da solução (com apenas duas peças violando restrições) não se consegue mais melhorar o resultado. Algoritmos de backtracking (como utilizados no Prolog) podem resolver este empecilho, assim como reiniciar o problema ao cair em uma destas situações.

Porém, utilizei uma abordagem um pouco distinta: quando o número de restrições é baixo (adotei como sendo 5), existe uma probabilidade de se realizar movimentos aleatórios com alguma rainha, de modo a desordenar um pouco o problema e sair desta condição de travamento. Um número máximo de steps também pode prevenir tal ocorrência.

3.2 Implementação do máximo de função

Analiticamente, não é difícil ver que a função é a soma de 5 exponenciais negativas, sendo que 4 delas possuem valor máximo 1, centrados em (-5, -5), (-5, 5), (5, -5), (5,5). A primeira exponencial tem pico 4 em (0, 0). Assim, devemos possuir 5 máximos nessa função, sendo 4 deles locais. Abaixo temos o gráfico dessa função:

```
In[9]:= Plot3D[{4 e^-(x^2 + y^2) + e^-((x-5)^2 + (y-5)^2) + e^-((x+5)^2 + (y-5)^2) + e^-((x-5)^2 + (y+5)^2) + e^-((x+5)^2 + (y+5)^2)},
        {x, -10, 10}, {y, -10, 10}, PlotRange → All,
        ColorFunction → (ColorData["DarkRainbow"][#3] &), ImageSize → 500]
```

Out[9]=

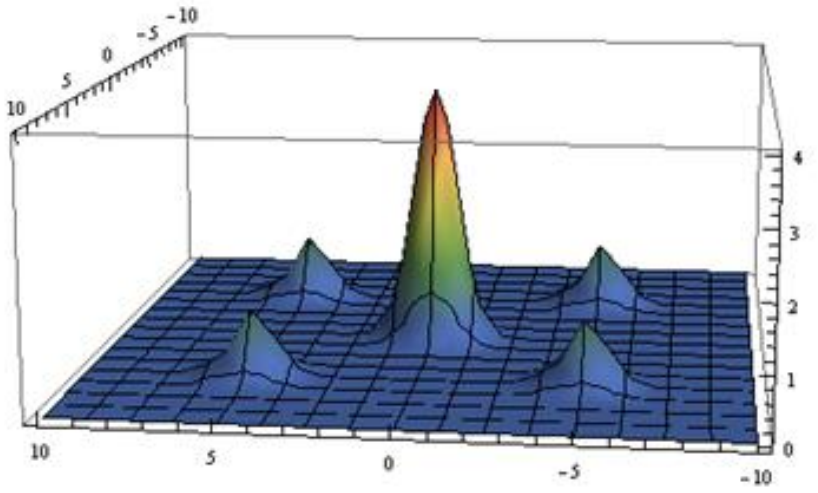

Figura - Gráfico da função analisada.

Assim, algoritmos diretos como hill-climbing tendem a ficar presos nesses máximos locais, sendo necessário resetar a solução (em algum ponto inicial distinto) para alcançar o máximo global. Mesmo a têmpera simulada pode acabar caindo em algum destes picos locais, apesar de ela considerar também caminhos menos favoráveis em seu algoritmo (com menor peso de probabilidade).

Em minha solução por diversas vezes me deparei com o travamento nos máximos locais, o que ocorre conforme o ponto inicial escolhido e os parâmetros da têmpera simulada escolhidos.

3.3 Implementação da satisfação de restrições

Há diversas maneiras de se atacar o problema dos vizinhos de Einstein. Nas duas modelagens que desenvolvi, tive de descrever as restrições existentes e com base nelas determinar o domínio da variável que eu gostaria de atribuir. Em seguida, tive de criar uma heurística para selecionar a melhor variável para solucionar o problema, optei por aquela que tivesse o menor domínio.

Em ambos os casos, há retorno da solução quando esta é encontrada e retorno nulo quando nada é encontrado (backtracking), continuando a iteração dos possíveis valores em um DFS (busca em profundidade).

3.3.1 Implementação procedural

Na implementação procedural, modelei os dados como um único dicionário de informações, onde as chaves são divididas por grupos de informação. O valor do mapa para uma dada chave de informação corresponde ao número da casa atribuído. Isso facilitou bastante as consultas e também determinar os domínios de cada variável. Abaixo temos a modelagem:

```
cores = ["vermelha", "amarela", "azul", "marfim", "verde"]
pessoas = ["ingles", "espanhol", "noruegues", "ucraniano", "japones"]
marcas = ["kool", "chesterfield", "winston", "lucky_strike", "parliament"]
bebidas = ["suco_laranja", "cha", "cafe", "leite", "agua"]
animais = ["cachorro", "raposa", "caramujos", "cavalo", "zebra"]
groups = [cores, pessoas, marcas, bebidas, animais]

neigh = {}
```

Por exemplo, neight[“vermelha”] = 2 quer dizer que a casa número dois é vermelha, neigh[“cachorro”] = -1 quer dizer que a variável cachorro ainda não foi definida.

Como próximo passo criei uma função que descreve todos os requisitos do problema, retornando quantas foram quebradas no total. Observe que é muito fácil inserir novas restrições ou alterar as que já existem, conforme os desejos do roteiro, apesar de não faze uso de orientação a objeto.

A ideia do PSR é atribuir os valores de modo que essa função seja sempre zero:

```
def restricoes(info):
    u"""Nro restrições quebradas."""
    restricoes = [
        undefined(["ingles", "vermelha"], info) or
        info["ingles"] == info["vermelha"],
```

```
        undefined(["espanhol", "cachorro"], info) or
        info["espanhol"] == info["cachorro"],

        undefined(["noruegues"], info) or
        info["noruegues"] == 0,

        undefined(["kool", "amarela"], info) or
        info["kool"] == info["amarela"],

        undefined(["chesterfield", "raposa"], info) or
        abs(info["chesterfield"] - info["raposa"]) == 1,

        undefined(["noruegues", "azul"], info) or
        abs(info["noruegues"] - info["azul"]) == 1,

        undefined(["winston", "caramujos"], info) or
        info["winston"] == info["caramujos"],

        undefined(["lucky_strike", "suco_laranja"], info) or
        info["lucky_strike"] == info["suco_laranja"],

        undefined(["ucraniano", "cha"], info) or
        info["ucraniano"] == info["cha"],

        undefined(["japones", "parliament"], info) or
        info["japones"] == info["parliament"],

        undefined(["kool", "cavalo"], info) or
        abs(info["kool"] - info["cavalo"]),

        undefined(["cafe", "verde"], info) or
        info["cafe"] == info["verde"],

        undefined(["verde", "marfim"], info) or
        info["verde"] == info["marfim"] + 1,

        undefined(["leite"], info) or
        info["leite"] == 2,
    ]
    return len(restricoes) - sum(restricoes)
```

A solução do problema é quando temos então todas as variáveis atribuídas:

```
def is_solved(info):
    u"""Verifica se foi resolvido."""
    return not (-1 in info.values())
```

Basta então aplicar o algoritmo de backtracking visto no curso. Escolhe-se primeiro a variável com mais restrições (aquela que possui menor domínio), itera-se os valores de seu domínio (certamente um destes é a solução, pois existe uma solução) recursivamente, desde que essa análise ainda não tenha sido feita anteriormente.

```
def backtracking(info):
    u"""Verifica se algum campo é nulo."""
    if is_solved(info):
        return info
    for (var, group, domain) in select_var(info):
        for value in domain:
            new_info = copy.deepcopy(info)
            new_info[groups[group][var]] = value
            if not (new_info in visited):
                visited.append(new_info)
                result = backtracking(new_info)
                if result is not None:
                    return result
    return None
```

3.3.2 Implementação orientação a objetos

A ideia para resolver o problema usando orientação a objetos foi bem mais elegante que a procedural. Cada restrição foi modelada como um objeto "peça" de um quebra-cabeça. O objetivo deste algoritmo é encaixar essas peças (dentro das restrições) de modo a montar um quebra-cabeça com 5 peças (ou casas) no final. Observe a elegância para definir as restrições:

```
   def create_pieces(self):
        u"""Cria as peças iniciais."""
        self.pieces = [
            Piece({'pessoa': 'ingles', 'casa': 'vermelha'}),
            Piece({'pessoa': 'espanhol', 'animal': 'cachorro'}),
            Piece({'pessoa': 'noruegues', 'numero': 1}),
            Piece({'cigarro': 'kool', 'casa': 'amarela'}),
            Piece({'cigarro': 'winston', 'animal': 'caramujos'}),
            Piece({'cigarro': 'lucky_strike', 'bebida': 'suco_laranja'}),
            Piece({'pessoa': 'ucraniano', 'bebida': 'cha'}),
            Piece({'pessoa': 'japones', 'cigarro': 'parliament'}),
            Piece({'bebida': 'cafe', 'casa': 'verde'}),
            Piece({'bebida': 'leite', 'numero': 3}),

            Piece({'casa': 'marfim'}), Piece({'casa': 'azul'}),
            Piece({'cigarro': 'chesterfield'}), Piece({'bebida': 'agua'}),
            Piece({'animal': 'raposa'}), Piece({'animal': 'cavalo'}),
            Piece({'animal': 'zebra'}), Piece({'numero': 2}),
            Piece({'numero': 4}), Piece({'numero': 5}),
        ]
```

Um pouco menos elegante foi definir as condições do tabuleiro do quebra-cabeça (propriedades de vizinho). Foi preciso implementar um método separado para tal:

```
    def board_restriction(self, piece1, piece2):
        u"""Verifica restrições do tabuleiro."""
        if piece1.has({'cigarro': 'chesterfield'}) and \
                piece2.has({'animal': 'raposa'}):
            if not self.neighboors(piece1, piece2):
                return False

        if piece1.has({'pessoa': 'noruegues'}) and \
                piece2.has({'casa': 'azul'}):
            if not self.neighboors(piece1, piece2):
                return False

        if piece1.has({'cigarro': 'kool'}) and \
                piece2.has({'animal': 'cavalo'}):
            if not self.neighboors(piece1, piece2):
                return False

        if piece1.has({'casa': 'marfim'}) and \
                piece2.has({'casa': 'verde'}):
            if not self.neighboors_right(piece1, piece2):
                return False

        return True
```

Com base nestes métodos para verificar restrições, foram elaborados outros dois, um para determinação do domínio de uma dada peça (conjunto de outras peças às quais ela pode se encaixar) e seleção da peça com menor domínio (que será usada na próxima iteração do algoritmo). Por fim, temos o núcleo do algoritmo, que utiliza de backtracking para determinar a solução:

```
    def backtracking(self, pieces):
        u"""Usa backtracking para resolver."""
        if len(pieces) == 5:
            return pieces
        for (piece1, domain) in self.select_pieces(pieces):
            for piece2 in domain:
                new_pieces = copy.deepcopy(pieces)
                self.connect_pieces(new_pieces, piece1, piece2)
                result = self.backtracking(new_pieces)
                if result is not None:
                    return result
        return None
```

4. RESULTADOS

Nesta seção apresento os resultados encontrados decorrentes da execução dos algoritmos, avaliando a consistência dos outputs gerados.

4.1 Resultado das N-Rainhas

Para determinar o número médio de passos da solução do problema das N-Rainhas, cujo número é aleatório, devido ao elemento de perturbação inserido na solução, foi criado um método "monte_carlo" para solucionar o problema um número grande de vezes e depois determinar a média:

```
def monte_carlo(n_squares, depth):
    u"""Run algorithm several times."""
    total_steps = 0
    start = timeit.default_timer()

    for i in range(depth):
        board = [0] * n_squares
        total_steps += hill_climbing(board, 1000)[1]

    stop = timeit.default_timer()
    print "Avarage time: " + str(((stop - start) * 1.0) / depth)
    print "Avarage steps: " + str((total_steps * 1.0) / depth)
```

Para N = 8, por exemplo, temos o seguinte resultado da execução de Monte Carlo para 1000 iterações. Note que o tempo é dado em segundos e o número de steps é uma média (por isso não é inteiro):

```
************************************
*                                  *
*       Problema das N-Rainhas     *
*                                  *
************************************
Avarage time: 0.00925021745285
Avarage steps: 100.731
[Finished in 9.7s]
```

Com base na execução deste algorimo para vários valores de N, foi montada a seguinte tabela de execução:

N	Steps	Tempo (s)
5	11.7	0.00051
8	98.7	0.00916
10	294.3	0.0412
15	551.6	0.187
20	989.1	0.584
25	2245.3	2.04
30	4124.7	5.63

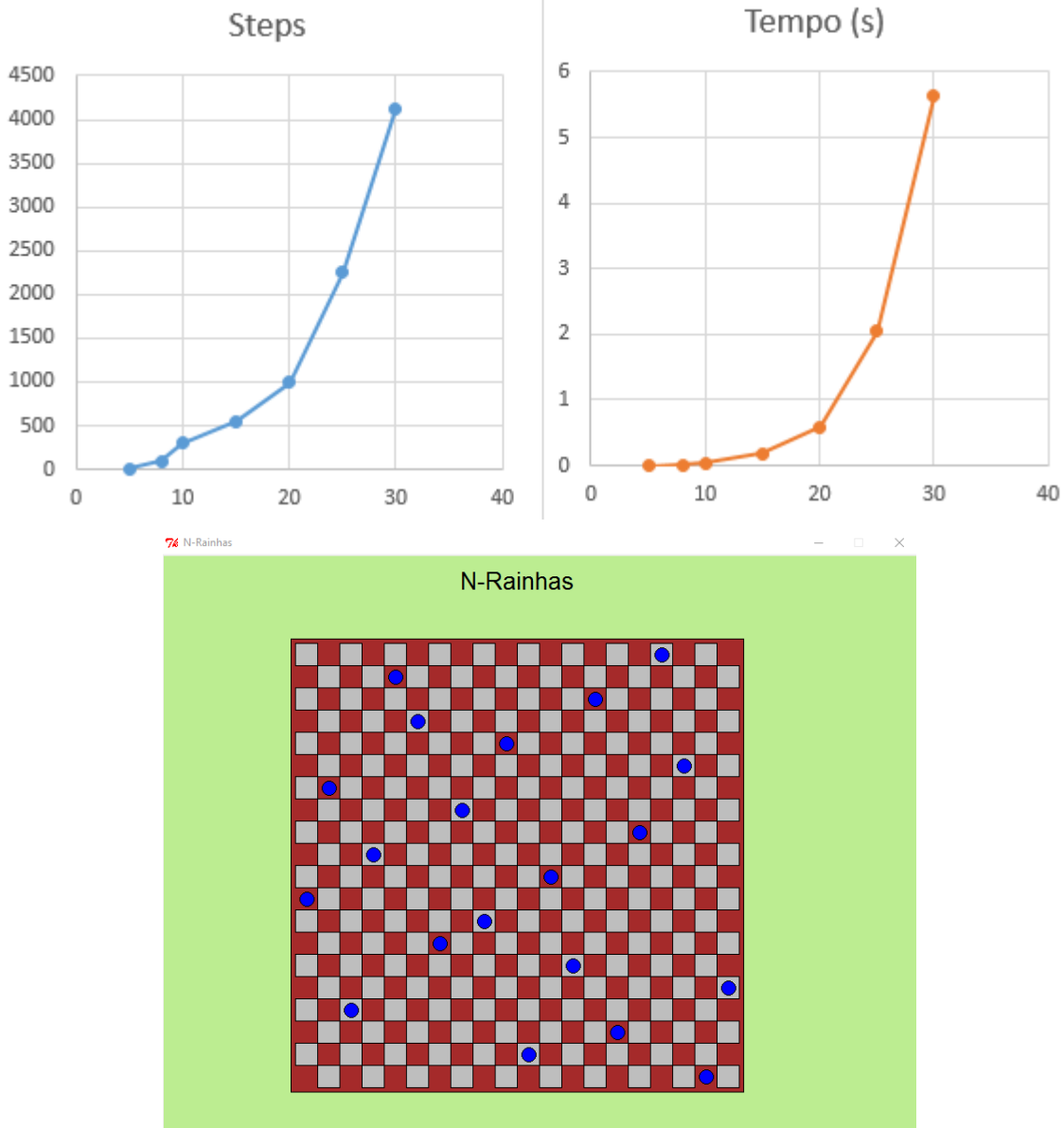

Figura – Solução das N-Rainhas para N = 20.

4.2 Resultado do máximo de função

A solução para encontrar o máximo de função tem resultados diferentes conforme os parâmetros aplicados a ela. Para determinar o máximo global com alguma confiança, seria necessário aplica-la a diversos pontos iniciais, além de considerar variações de temperatura e step específicos para solução com variações bruscas.

Abaixo apresento algumas soluções típicas encontradas. Escolhendo um ponto incial próximo ao zero, o máximo global é encontrado sem dificuldades:

solution = tempera_simulada(1.0, -2.0, 10.0, 1.0) (máximo global)

```
************************************
*                                  *
*     Problema do Máximo de f(x)   *
*                                  *
************************************
(0.04565439936781335, -0.020421554457098923)
3.99000704694
[Finished in 0.5s]
```

Tomando um ponto inicial intermediário entre o zero e os máximos locais, temos uma condição de disputa com alguma probabilidade de achar o máximo local/ máximo global, que depende da execução do programa. Uma melhoria de confiabilidade seria rodar o programa várias vezes, até achar o máximo global:

solution = tempera_simulada(4.0, -3.0, 1.0, 0.5) : (máximo local)

```
************************************
*                                  *
*     Problema do Máximo de f(x)   *
*                                  *
************************************
(5.007037198947561, -5.012074236059261)
0.999804709726
[Finished in 0.4s]
```

solution = tempera_simulada(4.0, -3.0, 1.0, 0.5) (executando novamente)

```
************************************
*                                  *
*     Problema do Máximo de f(x)   *
*                                  *
************************************
(-0.0810891743287172, 0.06260309519997953)
3.9582410979
[Finished in 0.4s]
```

Ou então simplesmente aumentar a exploração do algoritmo, aumentando a temperatura inicial da têmpera e também o seu step de exploração:

solution = tempera_simulada(5.0, 3.0, 10.0, 1.0) : (outro máximo local)

```
************************************
*                                  *
*     Problema do Máximo de f(x)   *
*                                  *
************************************
(0.02176643204711204, 0.036298829182994874)
3.99284088402
[Finished in 0.5s]
```

Quando o ponto é muito distante dos picos, todas as funções exponenciais que compõem a função tendem a zero. As variações são praticamente nulas e o algoritmo da têmpera simulada se prova ineficaz neste caso.

solution = tempera_simulada(20.0, 20.0, 1.0, 0.5) : (indiferente)

```
************************************
*                                  *
*     Problema do Máximo de f(x)   *
*                                  *
************************************
(19.140351469450135, 19.614324612083344)
2.55306140995e-180
[Finished in 0.5s]
```

Consegui contornar esse problema ao calcular o delta percentual no algoritmo da têmpera simulada: "`delta = (fxx - fx) / fx`". Em seguida, aumentei a temperatura inicial da têmpora e consegui ao menos achar um máximo local (a têmpora esfria após chegar ao máximo local):

solution = tempera_simulada(20.0, 20.0, 5.0, 0.5)

```
************************************
*                                  *
*     Problema do Máximo de f(x)   *
*                                  *
************************************
(5.0279199920251205, 5.040518237934931)
0.99758167531
[Finished in 0.4s]
```

Para aumentar o domínio de sua exploração, aumentei ainda mais a temperatura e também aumentei o seu step, de modo que a busca seja mais exploratório através do plano xy. Como resultado, consegui encontrar o máximo global da função:

solution = tempera_simulada(20.0, 20.0, 20.0, 1.0)

```
************************************
*                                  *
*     Problema do Máximo de f(x)   *
*                                  *
************************************
(0.05074563706574553, -0.013676867723382946)
3.98896653963
[Finished in 0.4s]
```

4.3 Resultado dos vizinhos de Einstein

Conferindo manualmente as soluções geradas, confirma-se o sucesso dos algoritmos desenvolvidos. Segue uma breve análise dos outputs e desempenhos das duas abordagens:

4.3.1 Resultado do método procedural

O código final ficou bem direto e fácil de se entender (150 linhas de código), com seu funcionamento centralizado no método "backtracking". A execução também foi razoavelmente rápida:

```
[('amarela', 0),
 ('kool', 0),
 ('agua', 0),
 ('raposa', 0),
 ('noruegues', 0),
 ('cavalo', 1),
 ('cha', 1),
 ('ucraniano', 1),
 ('azul', 1),
 ('chesterfield', 1),
 ('winston', 2),
 ('vermelha', 2),
 ('caramujos', 2),
 ('leite', 2),
 ('ingles', 2),
 ('suco_laranja', 3),
 ('cachorro', 3),
 ('lucky_strike', 3),
 ('espanhol', 3),
 ('marfim', 3),
 ('cafe', 4),
 ('japones', 4),
 ('parliament', 4),
 ('verde', 4),
 ('zebra', 4)]
**********************************
*                                *
* Problema dos vizinhos de Einstein *
*                                *
**********************************
[Finished in 0.4s]
```

Figura – Execução dos vizinhos de Einstein usando algoritmo procedural.

4.3.2 Resultado do método orientado a objeto

A ideia de solução é bem mais simplificada que a solução anterior, sendo bem fácil de explica-la para leigos. A ideia de orientação a objetos reduz a carga de um único método para distribuí-la entre objetos, dentro de um contexto de encargos (aumentando coesão e diminuindo acoplamento).

Porém o código ficou maior (200 linhas de código) e encontrei também problemas graves, como comparar se dois objetos são iguais (eles podem ter mesmo conteúdo, mas serem instâncias diferentes). Observei também uma degradação de desempenho, a chamada de métodos de instância tem sempre um argumento a mais (self), além das variáveis de instância ocuparem maior espaço em memória.

Abaixo temos o output de solução deste problema:

```
**********************************
*                                *
* Problema dos vizinhos de Einstein *
*                                *
**********************************

Puzzle:
{'animal': 'raposa',
 'bebida': 'agua',
 'casa': 'amarela',
 'cigarro': 'kool',
 'numero': 1,
 'pessoa': 'noruegues'}
{'animal': 'caramujos',
 'bebida': 'leite',
 'casa': 'vermelha',
 'cigarro': 'winston',
 'numero': 3,
 'pessoa': 'ingles'}
{'animal': 'cachorro',
 'bebida': 'suco_laranja',
 'casa': 'marfim',
 'cigarro': 'lucky_strike',
 'numero': 4,
 'pessoa': 'espanhol'}
{'animal': 'zebra',
 'bebida': 'cafe',
 'casa': 'verde',
 'cigarro': 'parliament',
 'numero': 5,
 'pessoa': 'japones'}
{'animal': 'cavalo',
 'bebida': 'cha',
 'casa': 'azul',
 'cigarro': 'chesterfield',
 'numero': 2,
 'pessoa': 'ucraniano'}
[Finished in 1.4s]
```

Figura – Execução dos vizinhos de Einstein usando orientação a objeto.

5. CONCLUSÃO

A realização deste trabalho permitiu aprender na prática o funcionamento de buscas de melhoria iterativa e PSRs, compreender as dificuldades de superar os obstáculos de máximos locais (em busca iterativa) e a complexidade de se implementar algoritmos backtracking, especialmente sem uma boa heurística para atacar o problema.

Cada problema tem suas peculiaridades e maneiras de se otimizar a busca de solução, sendo casos de estudo que podemos aprofundar bastante e fazer uma análise completa sobre o caso.

No caso das N-Rainhas, por exemplo, é possível tornar a complexidade linear desde que se tenha cuidado de como abordar o problema, já que o número de estados

Para o máximo de função, observa-se que os parâmetros da têmpera simulada têm impacto direto e significativo no resultado, de modo que cada caso de função é único.

Já o problema de PSR tem como principal desafio a escolha da heurística certa para determinar qual variável analisar. Tomar as melhores decisões significa expandir menos a árvore de busca do backtracking e otimizar o tempo de busca total, que se provou inviável tratar neste caso (conectar 14 peças entre si demanda grande esforço computacional).

6. REFERÊNCIAS BIBLIOGRÁFICAS

[1] Slides de aula: http://www.comp.ita.br/~pauloac/

[2] Stackoverflow (sempre): http://stackoverflow.com/

[3] Wikipedia: https://www.**wikipedia**.org