UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO (PPGCC)
CAMPUS PONTA GROSSA

Projeto nº 1 – Algoritmos de Busca

Andressa Oliveira; Gilberto Júnior; Paulo Costa; Paulo Duarte
andressasouza@alunos.utfpr.edu.br; gilbertoluis@alunos.utfpr.edu.br;paulocosta@alunos.utfpr.edu.br;
pkduarte@gmail.com;

1. Introdução

Os algoritmos de busca são utilizados para diversos fins, como por exemplo em aplicações que implementam o GPS (do inglês, *Global Positioning System*), onde o usuário insere endereço de origem e o endereço de destino. Após isso, o software localiza a melhor rota, baseado em critérios como distância, congestionamento de veículos, pedágios, entre outros.

No campo da inteligência artificial, os algoritmos de busca possuem grande importância, pois segundo Russell (2013, p.99), "Um algoritmo de busca recebe um problema como entrada e devolve uma solução sob a forma de uma sequência de ações."

Neste sentido, após encontrar a solução do problema, passa-se à fase de execução, que é o momento onde as ações podem ser executadas. O procedimento de busca será acionado após a formulação do objetivo e do problema a ser resolvido. A definição de um problema, que é feita antes mesmo do procedimento de busca, é composta por cinco componentes [1]:

a) Estado Inicial;
b) Ações;
c) Modelo de transição;
d) Teste de objetivo;
e) Custo de caminho;

Além destes componentes, é importante elencar o espaço de estados possíveis para o mundo do agente envolvido.

Diante disso, o objetivo deste trabalho é apresentar um relatório contendo a abordagem, resolução por meio de diversos algoritmos de busca e uma avaliação comparativa dos resultados para dois problemas distintos: o quebra-cabeça de 8 peças e o tabuleiro de xadrez com 8 rainhas. Em seguida, serão apresentados os capítulos de descrição do problema, métodos e algoritmos com uma sucinta explicação teórica de cada algoritmo implementado e seus respectivos resultados obtidos e por fim, a conclusão e referências utilizadas no desenvolvimento deste projeto.

2. Descrição dos Problemas

Os problemas abordados neste projeto são relativos a jogos de tabuleiro, porém cada qual com dimensões, regras de movimentação e posicionamento, e objetivos distintos.

2.1 Quebra-Cabeça das 8 peças

Consiste em um tabuleiro quadrado com 9 posições, onde são dispostas 8 peças numeradas de forma sequencialmente (1 a 8), ficando uma das posições deste tabuleiro vazia, conforme a Figura 1:

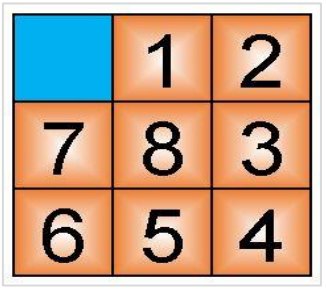

Figura 1 - Estado inicial Puzzle
(Fonte: Autoria própria)

O movimento permitido é deslizar as peças para este espaço vazio, tornando o espaço antes por ela ocupado um espaço vazio. Esses movimentos podem ser realizados para cima e para baixo, para a direita e para a esquerda. Serão realizados sucessivos movimentos serão para atingir a disposição final das peças no tabuleiro, ilustrada na Figura 2:

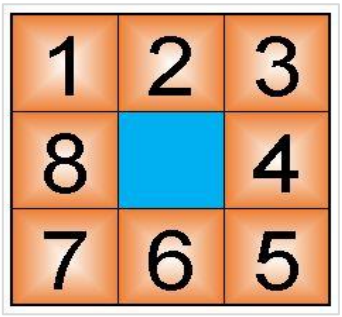

Figura 2 - Estado objetivo Puzzle
(Fonte: Autoria própria)

Neste problema, é importante o registro das movimentações realizadas para atingir o estado final, assim não foram empregados algoritmos de busca local, visto que estes não mantêm registro das ações realizadas e dos estados intermediários durante o processo de execução. Os algoritmos empregados foram:

a) BFS;
b) DFS;
c) Busca Gulosa;
d) A* com heurísticas: Manhattan e Peças Faltantes.

2.2 As 8 Rainhas

Trata-se de um tabuleiro de jogo de xadrez, no qual 8 rainhas devem ser dispostas de tal forma que não seja possível nenhum tipo de ataque entre elas, ou seja, não podem ocupar a mesma linha, coluna ou diagonal. Na Figura 3 observa-se o caminho de posições que cada rainha ataca de acordo com sua posição. Neste exemplo ambas as rainhas estão atacando suas diagonais (positivas e negativas) e direcionais (cima, baixo, esquerda e direita).

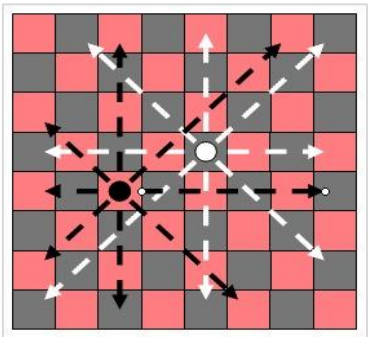

Figura 3 - Figura ilustrando as áreas de ataque de cada uma das duas rainhas do exemplo
(Fonte: Serrano [2])

Para este caso, foi possível o emprego de um algoritmo de busca local, visto que para a solução do problema o registro dos passos não é relevante, pois as rainhas são inseridas diretamente na posição final, conforme o estado final fornecido pelo algoritmo.

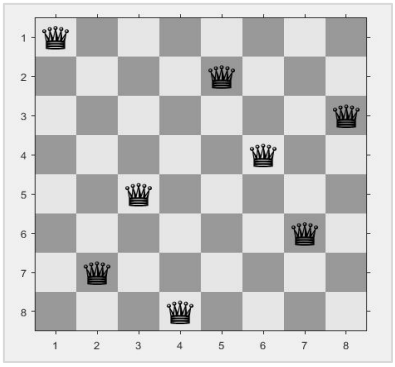

Figura 4 - Representação de uma possível solução para o problema das rainhas
(Fonte: Dehghani [3])

Os algoritmos utilizados para este problema forma:

a) Busca Heurística A*;

b) Hill Climbing.

Os resultados destes algoritmos são apresentados na seção de resultados.

3. Métodos e Algoritmos

A formulação de objetivos é o primeiro passo para a resolução de problemas. O objetivo pode ser determinado por um estado final desejado explícito, como no caso dos quebra cabeças, ou por um conjunto de estados que satisfaz uma condição, como no caso das 8 rainhas. Uma função de verificação é utilizada para testar cada estado, se este é um estado final desejado. Esta função é chamada “teste de objetivo” [1]

Definido os objetivos, o problema pode ser formulado e a busca pela solução pode ser executada. A resolução dos problemas neste projeto foi realizada através de alguns algoritmos de busca.

> “Em que vemos como um agente pode encontrar uma sequência de ações que alcança seus objetivos quando nenhuma ação isolada é capaz de fazê-lo.” Russel [1]

Portanto, pode-se considerar que os algoritmos de busca, cada qual em sua metodologia, busca chegar até a solução desejada, ao objetivo, podendo registrar ou não a sequência de ações por ele executadas. Algumas ações estão disponíveis para serem executadas no estado atual, e cada ação executada neste estado gera um novo estado, sendo a escolha da ação dependente do algoritmo de busca utilizado. Segundo Russel [1], "Um caminho no espaço de estados é uma sequência de estados conectados por uma sequência de ações"

3.1 Algoritmos de Busca Cega

São algoritmos que percorrem os estados possíveis em busca do estado final (objetivo), porém sem utilizar informações que possam indicar uma direção que possa reduzir o custo para chegar ao estado final.

3.1.1 DFS (Depth-First Search)

O algoritmo DFS (do inglês, Depth-First Search) realiza a busca em profundidade, expandindo até o nó mais profundo na posição atual da árvore de busca até não haver mais nós sucessores. À medida que os nós são expandidos, eles são removidos da árvore e a busca prossegue no nó vizinho que ainda não foi explorado.

A implementação da busca pode ser em grafos ou em árvores, sendo a busca em grafos livres de estados repetidos e caminhos redundantes, e completa em espaços de estado finitos, expandindo cada nó. Em árvores, não é completa, pois está sujeita a laços e caminhos redundantes, podendo ser modificada para evitar os laços, mas não os caminhos redundantes. [1].

Não é uma busca ótima, porém é utilizada como carro-chefe básico em muitas áreas da Inteligência Artificial devido a sua complexidade espacial, pois para um estado de ramificação *b* e profundidade de máxima *m*, a complexidade espacial é *O(bm)*.

3.1.2 BFS (Depth-First Search)

O algoritmo BFS (do inglês, Breadth-First Search) expande inicialmente o nó raiz, expandindo na sequência todos os nós sucessores deste. Assim, todos os nós de um determinado nível da árvore são expandidos antes que qualquer outro do próximo nível.

Neste contexto de resolução de problemas, o algoritmo é ligeiramente diferente do algoritmo de busca em grafos [1], pois a cada nó gerado é realizado o teste de objetivo, verificando assim se este nó representa o estado objetivo, e caso seja, retorna a função de busca.

Esta técnica de busca não garante que o nó encontrado seja o melhor resultado (ótimo), visto que estados mais próximos.

3.1.3 Comparativo entre BFS e DFS

Com o intuito de comparar a eficiência e complexidade dos algoritmos descritos acima, a Tabela 1 mostra os algoritmos de busca cega DFS e BFS e algumas de suas características principais:

Critério	BFS	DFS
Completa?	Sim, se b finito	Não
Tempo	$O(b^d)$	$O(b^m)$
Espaço	$O(b^d)$	$O(b^m)$
Ótima?	Sim, se custos dos passos idênticos	Não

Tabela 1 - Comparativo entre BFS e DFS
(Fonte: Adaptado de Russel [1])

3.2 Algoritmos de Busca Heurística

Também conhecidos como algoritmos de busca informada, utilizam conhecimento sobre o problema, em vez de apenas sua definição, podendo assim encontrar soluções de forma mais eficiente que algoritmos de busca cega. [1]

A informação é uma função de avaliação, que fornece ao algoritmo uma estimativa de custo para os possíveis nós a serem expandidos, sendo escolhido para a expansão o nó que apresente o menor custo estimado.

3.2.1 Busca Gulosa

Considerado como um algoritmo "ambicioso", realiza a expansão a partir do nó com menor custo estimado ao objetivo, usando como métrica apenas a função heurística:

$$f(n) = h(n)$$

Não é um algoritmo ótimo, sua complexidade no pior caso é *O(b^m)*, sendo *m* a profundidade máxima do espaço de busca. A complexidade pode ser substancialmente reduzida com uma função heurística de boa qualidade. [1]

3.2.1 Algoritmo A *

Como na busca gulosa, utiliza uma função heurística, porém combinada com a função e do custo para alcançar o nó:

$$f(n) = g(n) + h(n)$$

Trata-se de uma função de busca completa e ótima, tanto para a versão de busca em árvore quanto busca em grafos, desde que *h(n)* satisfaça algumas condições:

a) Busca em Árvore: *h(n)* deve ser admissível, ou seja, nunca superestimar o custo de atingir o objetivo. "Heurísticas admissíveis são otimistas por natureza..."[1]
b) Busca em Grafos: *h(n)* deve ser consistente, ou seja, "*h(n)* será consistente se, para cada nó *n* e para cada sucessor n' de n gerado por uma ação a, o custo estimado para alcançar o objetivo de n não for maior que o custo do passo de chegar a *n'* mais ou custo de alcançar o objetivo de *n*'. " [1]

A Figura 5 representa essa condição, como uma forma genérica da desigualdade triangular, onde um dos lados nunca pode ser maior que a soma dos outros dois.

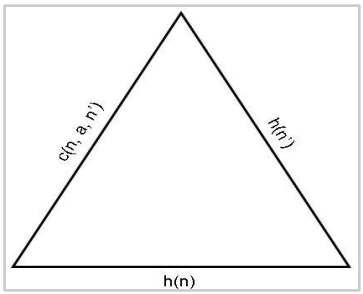

Figura 5 - Representação das medidas para a condição de consistência de uma função heurística
(Fonte: Autoria própria)

A consistência é uma exigência mais rigorosa que a admissibilidade, pois toda heurística consistente é também admissível segundo Russel [1]

3.3 Algoritmos de Busca Local

Os algoritmos de busca local são utilizados quando não há necessidade de registro do caminho percorrido, ou seja, os estados e as ações realizadas para chegar ao estado objetivo. Desta forma, esta classe de algoritmos precisa manter apenas o estado atual e o estado objetivo armazenados, sem a necessidade de registrar os estados anteriores.

3.3.1 Hill Climbing

Dado um estado inicial, o algoritmo de subida de encosta busca dentre os estados vizinhos aquele que possui o maior valor e se move para este estado, tornando-o o estado atual. Faz isso sucessivamente até encontrar um estado onde não existam vizinhos com valor mais alto. Neste ponto, o estado selecionado é aquele do topo da encosta.

Pela sua natureza de operação, o algoritmo sempre encontrará um valor máximo local, o qual não é necessariamente o valor máximo global, pois ao atingir um pico, retorna o valor encontrado, sem descer a encosta para buscar outra mais alta. Pode também chegar a uma planície, onde não há valores mais altos em seus estados vizinhos, mas sim valores iguais, esses locais são chamados platô.

Normalmente estes algoritmos utilizam uma formulação de estados completos de acordo com Russell [1], como no problema das 8 rainhas, onde todas as rainhas são posicionadas aleatoriamente no tabuleiro, e o algoritmo realiza movimentos no sentido de diminuir os ataques possíveis entre as rainhas, até chegar no resultado desejado. Um espaço de estados é ilustrado através da Figura 6.

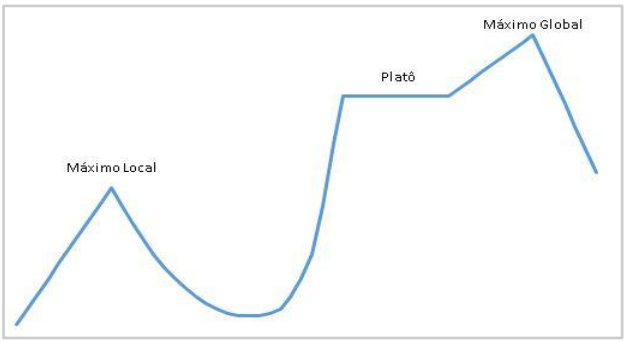

Figura 6 - Espaços de estado Hill Climbing
(Fonte: Autoria própria)

A próxima subseção aborda as especificidades do projeto desenvolvido tal como ferramentas utilizadas e estratégias.

3.3 Implementação

Para o desenvolvimento dos algoritmos deste projeto utilizamos a linguagem de programação Python 3, a escolha desta linguagem se deu pelo poder que a mesma possui por conta de suas bibliotecas robustas e atualmente ser a linguagem mais utilizada quando tratamos de projetos de Inteligência Artificial e áreas afim.

Para o gerenciamento de pacotes da linguagem utilizamos o Anaconda [5] acompanhada da ferramenta Jupyter [6] para codificação e compartilhamento dos algoritmos. Esta ferramenta também possibilita o desenvolvimento dos algoritmos em forma de blocos e transformando-os em relatórios de fácil entendimento.

Os algoritmos finalizados foram hospedados na plataforma Github [4] e as análises aqui apresentadas foram feitas através de bibliotecas Python e do Excel. Para finalizar, a equipe contava com as seguintes configurações de equipamentos:

OS	Processador	RAM
macOS BigSur	i5-dualCore 1.8 GHz	8gb
Ubuntu 14.04	AMD Athlon 2.1GHz	8gb
Windows 10	i7-7500U 2.90 GHz	8gb
Windows 10	Ryzen 7 1700	16gb

Tabela 2 - Configurações de equipamentos da equipe
(Fonte: Autoria própria)

4. Resultados & Discussões

Esta seção aborda os resultados obtidos através da implementação dos algoritmos selecionados e previamente exemplificados através de algumas métricas de eficiência com os casos bases disponibilizadas e outros analisados. Os testes aqui realizados foram analisados através das bibliotecas Matplotlib [7] e Seaborn[8] da linguagem Python 3.

4.1 Quebra-Cabeça das 8 peças

Os algoritmos de solução do Quebra-Cabeças das 8 peças podem receber *n* Entradas Iniciais e *n* Entradas Objetivo, para facilitar a análise padronizamos o uso dos casos solicitados para o projeto e nos algoritmos mais eficientes testamos outros casos também. As entradas solicitadas do projeto foram para Estado Inicial com a seguinte composição:

	1	2
7	8	3
6	5	4

Figura 7 – Caso Base
(Fonte: Autoria Própria)

Para o Estado Objetivo, o projeto solicita que o Quebra-Cabeças alcance como meta a composição ilustrada na Figura 8.

1	2	3
8		4
7	6	5

Figura 8 – Estado Objetivo Base
(Fonte: Adaptado Autoria Própria)

4.1 Algoritmos de Busca Local

Utilizamos a função time para armazenar o tempo decorrido dos algoritmos até chegar no Estado Objetivo. Esta função pode ser utilizada através da importação via *pip install time* através do Anaconda e importada no código através da linha *import time.* Com isto, chegamos a seguinte análise demonstrada através da Figura 9.

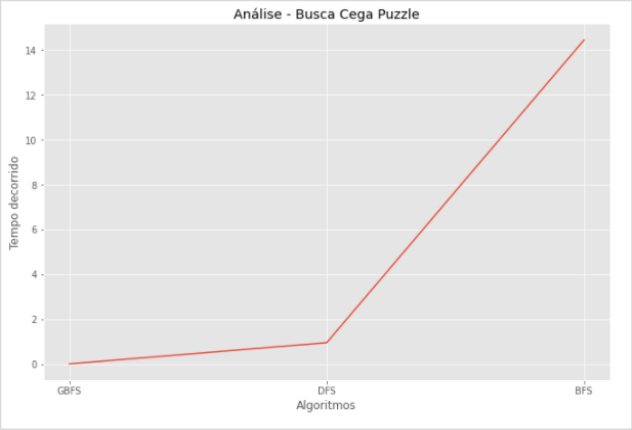

Figura 9 – Buscas Locais x Tempo Decorrido
(Fonte: Autoria própria)

Ao observarmos que os algoritmos de GBFS levaria poderia levar tempos extremos de execução, adotamos o uso da heurística de Estado Objetivo para auxiliar no desempenho e compararmos um algoritmo híbrido com dois algoritmos de buscas locais. O tempo de execução da Busca Gulosa foi menor, no entanto o algoritmo de DFS ainda ocupa a melhor colocação em eficiência quando falamos do critério de quantidade de interações, as informações plotadas no gráfico da Figura estão resumidas na Tabela 3.

Algoritmo	Interações	Tempo
BFS	124727	14.423
DFS	8	0.9832
GBFS	21	0.0523

Tabela 3 - Comparativo entre algoritmos de Busca Local
(Fonte: Adaptado Autoria Própria)

Com o algoritmo GBFS ainda testamos nosso caso 2 com as com função objetivo do caso base.

Entrada inserida: 1 5 4 3 7 2 6 8 0
Quebra-Cabeças Gerado:

1	5	4
3	7	2
6	8	0

Figura 10 – Caso Base
(Fonte: Autoria Própria)

```
Este algoritmo retornou o objetivo
com um tempo decorrido de 0.0732 ap
ós 21 peças movimentadas com o segu
inte caminho:
['ESQUERDA', 'CIMA', 'CIMA', 'DIR
EITA', 'BAIXO', 'ESQUERDA', 'CIMA
', 'ESQUERDA']
```

Trecho de código 1 – GBFS – Caso base 2
(Fonte: Autoria própria)

A subseção de busca heurística comprova ainda mais o poder do uso desta técnica na solução de problemas.

4.1 Algoritmos de Busca Heurística

Para as soluções utilizando heurística adotamos os critérios de avaliação:

a) Quantidade de árvores geradas pela heurística aplicada;

b) Quantidade de árvores processadas;

c) Quantidade de movimentos necessários para atingir o estado objetivo;

d) Tempo gasto no processamento do caso.

Foram utilizados o total de três casos testes para comparar o desempenho do algoritmo. Sendo assim, ilustramos os casos e seus resultados a seguir:

Entrada inserida: 0 1 2 3 4 5 6 7 8
Quebra-Cabeças Gerado:

	1	2
7	8	3
6	5	4

Figura 11 – Caso Base
(Fonte: Autoria Própria)

Chamamos a heurística de Peças Faltantes de h1 e Distância de Manhattan de h2. Com isto, analisamos primeiramente a quantidade de árvores geradas e expandidas através de ambas as técnicas e chegamos aos seguintes resultados ilustrados na Figura 12.

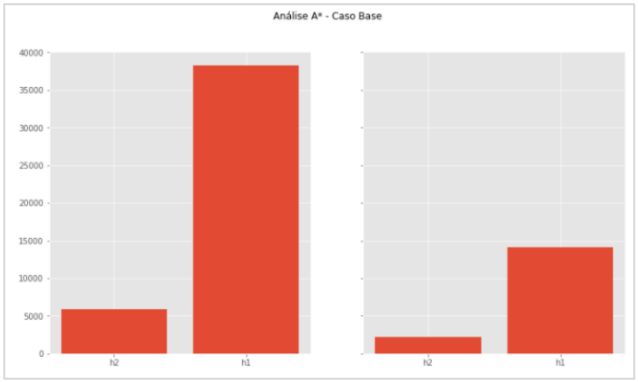

Figura 12 – Buscas Locais x Tempo Decorrido
(Fonte: Autoria própria)

Com esta análise inicial já é possível sugerir a conclusão de que a heurística h2 de Manhattan possui o menor número de nós de árvores gerado e expandidos em relação a heurística h1 de Peças Faltantes. Porém, curiosamente ambas as estratégias retornaram o mesmo número de passos para a solução de Estado Objetivo com o total de 22 movimentos. A Tabela 4 representa os valores apresentados na Figura 8.

Algoritmo	Gerados	Expandidos
A* H1	38208	14067
A* H2	5887	2194

Tabela 4 - Comparativo entre algoritmos de A* - Caso Base
(Fonte: Autoria Própria)

Em um segundo caso de testes utilizamos outro caso de entrada inicial com o mesmo estado de objetivo e obtivemos os seguintes resultados.

Entrada inserida: 1 5 4 3 7 2 6 8 0
Quebra-Cabeças Gerado:

1	5	4
3	7	2
6	8	0

Figura 13 – Caso Base
(Fonte: Autoria Própria)

Chamamos novamente a heurística de Peças Faltantes de h1 e Distância de Manhattan de h2. Com isto, analisamos primeiramente a quantidade de árvores geradas e expandidas através de ambas as técnicas e chegamos aos seguintes resultados

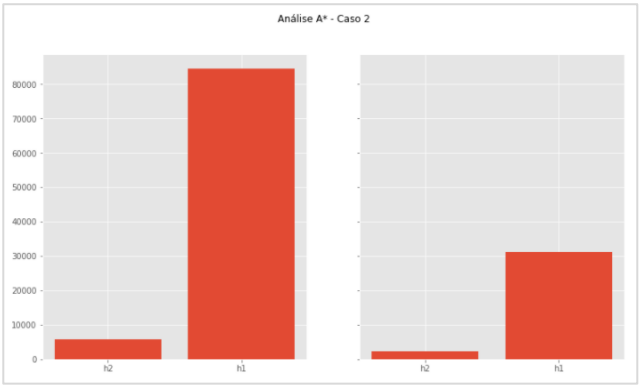

Figura 14 – Buscas Locais x Tempo Decorrido
(Fonte: Autoria própria)

Os itens plotados na Figura estão dispostos na Tabela 5. Observamos novamente a prevalência da heurística de Manhattan sobre a de peças faltantes.

Algoritmo	Gerados	Expandidos
A* H1	84416	31086
A* H2	5887	2194

Tabela 5 - Comparativo entre algoritmos de A* - Caso 2
(Fonte: Autoria Própria)

Não foi possível realizar outros casos testes nos demais algoritmos por conta do tempo decorrido gasto com eles. Entendemos dessa forma que os algoritmos que melhores se sobressaíram foram os algoritmos dos quais utilizamos estratégias de heurísticas, GBFS, A* com Manhattan e A* com Peças Faltantes.

4.2 As 8 Rainhas

Esta seção aborda os resultados obtidos através do desenvolvimento de duas estratégias de algoritmos na solução do problemas das 8 Rainhas.

4.2.1 Busca Gulosa

Para diversificamos os resultados obtidos e implementarmos outra estratégia de solução para o problema das Rainhas desenvolvemos uma Busca Gulosa para encontrar possíveis soluções e contabilizar quantos tabuleiros conseguimos encontrar através do algoritmo.

Sabemos que em um problema onde N seja o número de rainhas e este N seja igual a 8 teremos

um tabuleiro de dimensão 8x8, logo teremos 64 posições disponíveis. Pensando matematicamente, para a primeira rainha teremos o total de 64 possibilidades, para a segunda 63 e assim sucessivamente. Dessa forma, conseguimos abstrair que a combinação possível de possibilidades se dá por:

$$\frac{64 * 63 * 62 * 61 * 62 * 59 * 67}{8!}$$

Isso nos retorna a combinação C64,8, ou seja, o número de combinações de 64 elementos tomados de 8 a 8 o que nos retorna um total de 4426165368 o que é um espaço de busca extenso demais. Para reduzir estes espaços de busca precisamos utilizar as regras de ataque da rainha, dessa forma sabemos que as rainhas ocuparão diferentes colunas (numeradas de 1 a 8)

- A rainha **R1** terá 8 possibilidades de linhas na coluna 0,
- A rainha R2 terá 8 possibilidades de linhas na coluna 1,
- {...)
- A rainha R8 terá 8 possibilidades de linhas na coluna 7.

Com essas reduções alcançamos o total de 8! = 40320 o que torna a solução computacionalmente possível. A partir desta análise teórica e matemática retornarmos que nosso algoritmo obteve sucesso encontrando o total de 92 soluções possíveis e os tabuleiros podem ser encontrados em nosso repositório do GitHub no seguinte formato conforme ilustra a Figura 15.

```
0000000[R]
000[R]0000
[R]0000000
00[R]00000
00000[R]00
0[R]000000
000000[R]0
0000[R]000
```

Figura 15 – Busca Gulosa 8 Rainhas
(Fonte: Autoria Própria)

Onde o 0 representa posições livres do tabuleiro e [R] representa cada uma das 8 rainhas posicionadas de acordo com o problema.

4.2.1 Hill Climbing

Com o caso teste de base com N-Rainhas sendo 8 obtivemos a seguinte composição de tabuleiro através do algoritmo:

```
Melhor Vizinho # 6
[[0 0 0 0 1 0 0 0]
 [1 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0]
 [0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 0 1]
 [0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0]
 [0 0 1 0 0 0 0 0]]
```

Figura 14 – Hill-Climbing 8 Rainhas
(Fonte: Autoria Própria)

Nesta implementação optamos por utilizar a métrica de *Wall Time* inserindo um *%time* antes da chamada do algoritmo:

```
%time hillClimbing(numRainhas=9)
```

Trecho de código 2 – Wall-Time
(Fonte: Autoria própria)

O tempo de Wall Time obtido para posicionar as 8 rainhas foi de 2.83 com uso de 34 movimentos. O tempo de CPU foi de 2.19 segundos e de memória de sistema 54.4 ms. Aproveitamos o algoritmo e utilizamos mais 2 casos testes e os resultados obtidos são analisados e compõe a Tabela 6.

N-Rainhas	Movimentos	Wall-Time
8	34	2.83
9	13	2.33
10	12	3.01
15	38	33.1

Tabela 5 - Comparativo entre N-Rainhas para HillClimbing
(Fonte: Autoria Própria)

Observamos que para rainhas 9 e 10 obtivemos menores quantidades de movimento do que para o caso base 8. Porém, quando tratamos de 15 rainhas a quantidade de movimentos voltou a aumentar e o tempo de Wall-Time disparou. Plotamos um gráfico de movimento comparativo da métrica de total de movimentos realizados na Figura 15.

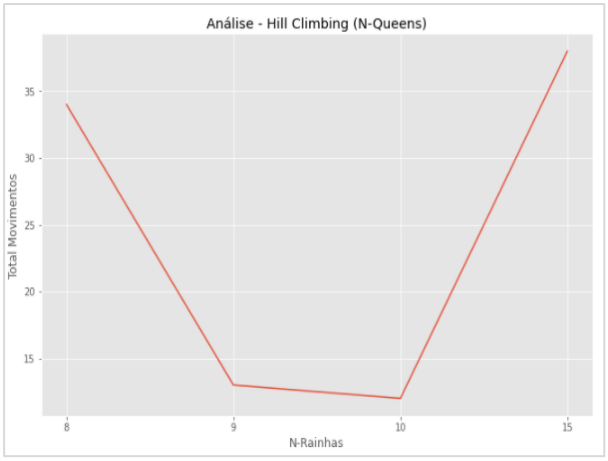

Figura 16 – Hill-Climbing N-Rainhas x Movimentos
(Fonte: Autoria Própria)

Complementando e finalizando nossa análise os melhores vizinhos encontrados para o caso mais custoso aqui testado com 15 rainhas sugere a disposição do tabuleiro da forma apresentada na Figura 16.

```
Melhor Vizinho # 38
[[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
 [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
 [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]]
```

Figura 16 – Hill-Climbing 15-Rainhas
(Fonte: Autoria Própria)

Concluímos que o algoritmo de Hill-Climbing se torna uma ótima estratégia na otimização de problemas que envolvam análises de custos para inserção ou ajustes de locais. Sabemos que o algoritmo busca encontrar pontos de máximo e mínimo afim de encontrar melhores soluções para o problema. Um problema que muito provavelmente possa trazer bons resultados com este algoritmo seria o de Caixeiro Viajante.

5. Conclusão

Com o desenvolvimento dos algoritmos aqui apresentados e nas breves análises que foram realizadas pudemos concluir que nem sempre o melhor algoritmos para um problema será a melhor solução para um outro. Como tudo na computação a resposta “depende” se torna essencial quando se projeta soluções de problemas.

Dependemos dos objetivos e metas que um problema gostaria de alcançar, dependemos da possibilidade de uso de heurísticas para otimização e de recursos computacionais para a aplicação da solução. Este projeto contemplou alguns dos problemas mais clássicos da computação sendo solucionados com alguns dos algoritmos também clássicos da teoria de Inteligência Artificial e através deste relatório resumimos a jornada realizada para alcance das metas solicitadas.

Referências

[1] RUSSELL, S. J., NORVIG, P. **Inteligência Artificial**. Tradução da terceira edição por Regina Célia Simile. Rio de Janeiro: Elsevier, 2013. 1324 p.

[2] SERRANO, A. I. **Resolucion del problema de N reinas mediante recocido simulado.** Sistemas Inteligentes: Reportes Finales Ene-May 2014, p. 14

[3] DEHGHANI, M. **N Queens Puzzle with Solution.** MATLAB Central File Exchange. Disponível em: <https://www.mathworks.com/matlabcentral/fileexchange/69461-n-queens-puzzle-with-solution> Acesso em 29 de março de 2021.

[4] OLIVEIRA, A. GitHub – Projeto 1. Disponível em: <https://github.com/andressalamarca/MastersProjects > Criado em 19 de março de 2021

[5] ANACONDA. Disponível em: <https://www.anaconda.com/> Acesso em 17 de março de 2021.

[6] JUPYTER. Disponível em: <https://jupyter.org/> Acesso em 19 de março de 2021.

[7] MATPLOLIB. Disponível em: <https://matplotlib.org/> Acesso em 28 de março de 2021.

[8] SEABORN: Statistical Data Visualization. Disponível em: <https://seaborn.pydata.org/> Acesso em 28 de março de 2021.