Lista 9 - Implementação de Puzzle
Inteligência Artificial

Caroline Freitas Alvernaz

[1]Instituto de Ciências Exatas e Informática – PUC Minas

1. Proposta

Desenvolvimento de um quebra-cabeça de 8 peças dispostas em um tabuleiro 3x3 em que deve-se movimentar uma peça por vez a fim de reorganizar os números de 1 a 8. Para isso, devem ser aplicados três algoritmos de busca para que o puzzle realize a solução automática e apresente a quantidade de movimentos usados por cada método e o tempo necessário para executar cada um.

2. Métodos utilizados

Para este trabalho, foram escolhidos os algoritmos Busca Gulosa, Busca Uniforme e A*.

2.1. Busca Gulosa

A busca gulosa, ou best-first greedy search, expande os nós de menor custo, avaliando-os de acordo com a função heurística apenas e, portanto, não garante a solução ótima e não é completa, porém é um algoritmo rápido se a heurística for boa. A heurística utilizada para a construção do puzzle foi a de Manhattan, ou seja, calcula a diferença entre a posição (considerando o tabuleiro como uma matriz 3x3) correta da peça e a posição original, garantindo sempre uma heurística admissível, pois o algoritmo de Manhattan dá a melhor solução independente de como as peças do tabuleiro estão dispostas, portanto a heurística será sempre menor ou igual ao custo real, ou seja, não superestima o custo. A função do algoritmo é descrita por: **f(n) = h(n)**, sendo n o número de vértices e h(n) a heurística, neste caso de Manhattan, atribuída a cada nó.

Descrição do Funcionamento da Busca Gulosa

1. Inicia-se a busca registrando o tempo inicial e inicializando:
    - a fila de prioridade vazia; e
    - o conjunto de estados visitados.
2. Define-se uma função interna chamada push, que recebe:
    - um estado;
    - o caminho percorrido até aquele estado; e
    - o custo acumulado (não utilizado na prioridade).

   Essa função insere na fila de prioridade uma tupla contendo:
    - a prioridade, dada exclusivamente pela heurística de Manhattan;
    - o custo acumulado;
    - o estado atual; e
    - o caminho até esse estado.
3. O estado inicial é inserido na fila com custo igual a zero.
4. Em um laço principal, o nó com menor heurística (menor valor estimado até o objetivo) é removido da fila.
5. Se o estado atual corresponde ao estado objetivo, a função retorna:

    - o caminho completo até a solução;
    - o número de passos (comprimento do caminho); e
    - o tempo de execução, em milissegundos.
6. Caso contrário, se o estado ainda não tiver sido visitado:
    - ele é adicionado ao conjunto de visitados;
    - todos os estados vizinhos são gerados;
    - cada vizinho não visitado é inserido na fila com custo incrementado de 1 e caminho atualizado.
7. Se a fila esvaziar sem encontrar a solução, a função retorna:
    - uma lista vazia;
    - zero passos; e
    - o tempo total de execução.

2.2. Busca Uniforme

A busca uniforme utiliza-se do algoritmo de Dijkstra para encontrar a solução: utiliza uma fila de prioridade que organiza os custos acumulados do caminho desde o início do percurso até o vértice alcançado. Assim, analisa-se apenas os pesos das arestas g(n), tendo sua função definida por **f(n) = g(n)**. Assim, diferente da busca gulosa, é garantida a busca ótima, se existir, e completa, pois todos os custos reais são analisados e, por isso, em contrapartida, se torna um algoritmo mais lento que a busca gulosa.

Descrição do Funcionamento da Busca Uniforme

1. Inicia-se a busca registrando o tempo inicial e inicializando:
    - a fila de prioridade vazia; e
    - o conjunto de estados visitados.
2. Define-se uma função interna chamada push, que recebe:
    - um estado;
    - o caminho percorrido até aquele estado; e
    - o custo acumulado.

   Essa função insere na fila de prioridade uma tupla contendo:
    - a prioridade (igual ao custo, no caso da busca uniforme);
    - o custo acumulado;
    - o estado atual; e
    - o caminho até esse estado.
3. O estado inicial é inserido na fila com custo igual a zero.
4. Em um laço principal, o nó de menor custo (ou seja, a menor prioridade) é removido da fila.
5. Se o estado atual corresponde ao estado objetivo, a função retorna:
    - o caminho completo até a solução;
    - o número de passos (comprimento do caminho); e
    - o tempo de execução, em milissegundos.
6. Caso contrário, se o estado ainda não tiver sido visitado:
    - ele é adicionado ao conjunto de visitados;
    - todos os estados vizinhos são gerados;
    - cada vizinho não visitado é inserido na fila com custo incrementado de 1 e caminho atualizado.
7. Se a fila esvaziar sem encontrar a solução, a função retorna:
    - uma lista vazia;
    - zero passos; e
    - o tempo total de execução.

2.3. Busca A*

Por fim, o algoritmo A* utiliza-se das duas métricas: custo real e estimativa de custo e, portanto, sua função é descrita por **f(n) = h(n) + g(n)**. O A* também garante a solução ótima, caso exista, e a busca completa, quando a heurística é admissível. Para este algoritmo, também foi utilizada a distância de Manhattan para cálculo da heurística.

Descrição do Funcionamento da Busca A*

1. Inicia-se a busca registrando o tempo inicial e inicializando:
    - a fila de prioridade vazia; e
    - o conjunto de estados visitados.
2. Define-se uma função interna chamada push, que recebe:
    - um estado;
    - o caminho percorrido até aquele estado; e
    - o custo acumulado.

   Essa função insere na fila de prioridade uma tupla contendo:
    - a prioridade, dada pela soma do custo acumulado e da heurística de Manhattan;
    - o custo acumulado;
    - o estado atual; e
    - o caminho até esse estado.
3. O estado inicial é inserido na fila com custo igual a zero.
4. Em um laço principal, o nó com menor valor de prioridade (custo + heurística) é removido da fila.
5. Se o estado atual corresponde ao estado objetivo, a função retorna:
    - o caminho completo até a solução;
    - o número de passos (comprimento do caminho); e
    - o tempo de execução, em milissegundos.
6. Caso contrário, se o estado ainda não tiver sido visitado:
    - ele é adicionado ao conjunto de visitados;
    - todos os estados vizinhos são gerados;
    - cada vizinho não visitado é inserido na fila com custo incrementado de 1 e caminho atualizado.
7. Se a fila esvaziar sem encontrar a solução, a função retorna:
    - uma lista vazia;
    - zero passos; e
    - o tempo total de execução.

3. Puzzle

O puzzle foi desenvolvido em linguagem Python e possui como interface inicial a figura abaixo que é composta pelo tabuleiro embaralhado inicialmente de maneira aleatória. Link para acesso ao código: Puzzle-GitHub. https://github.com/carolalvernaz/Inteligencia-Artificial/tree/main/Puzzle

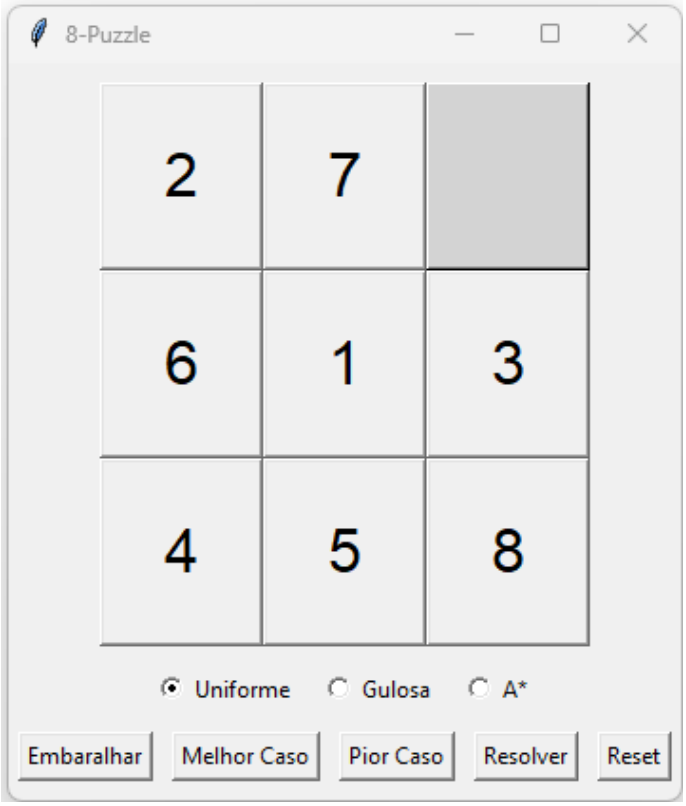

Figura 1. Interface inicial do puzzle.

Para iniciar o puzzle, deve-se selecionar o algoritmo a ser utilizado para chegar na solução e clicar na opção ”Resolver”para iniciar a ordenção, ou embaralhar o puzzle novamente de maneira aleatória, ou escolher o pior caso (tabuleiro ordenado de forma decrescente), ou o melhor caso (tabuleiro ordenado de forma crescente) para realizar o teste e, após, fazer a seleção do algoritmo e depois clicar em ”Resolver”. Após, o puzzle começa a realizar sua resolução de acordo com o algoritmo escolhido e, após, aparecem um ”pop-up”indicando quantos movimentos foram necessários e o tempo gasto de processamento, conforme mostra a imagem abaixo.

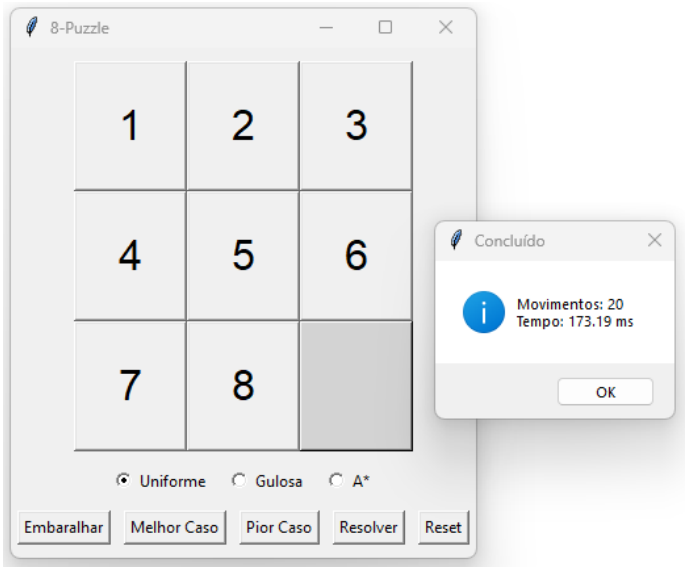

Figura 2. Interface após a resolução do puzzle via Busca Uniforme.

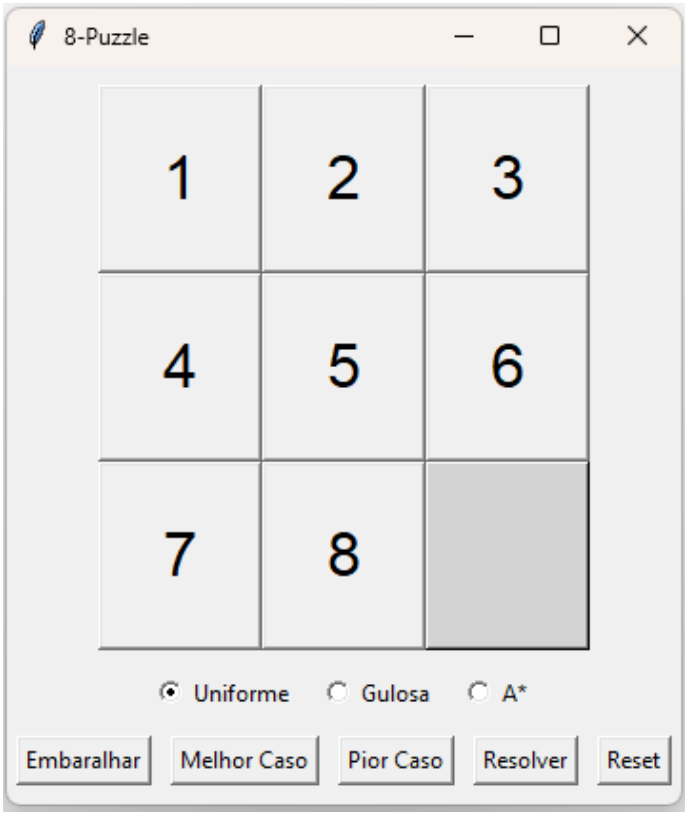

Figura 3. Interface no melhor caso.

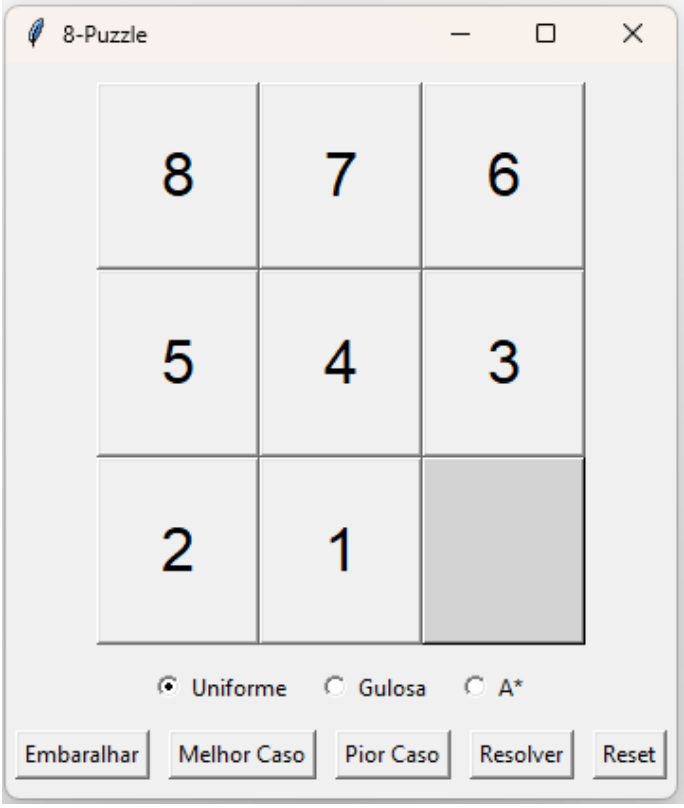

Figura 4. Interface no pior caso.

Ainda, para que se possa ser feita a comparação entre os métodos, deve-se clicar no botão ”Restart”para que o puzzle volte a ficar embaralhado como antes de iniciar a ordenação com o algoritmo escolhido. Após, é possível escolher um novo método a ser aplicado com o puzzle embaralhado da mesma forma de antes.

4. Resultados

Após a utilização do puzzle e realização de alguns testes, foi possível notar que o algoritmo de busca gulosa possui uma alta eficiência, uma vez que é, geralmente, o mais rápido entre os três analisados, porém tem uma baixa eficácia, já que não garante solução ótima nem busca completa. Já o algoritmo de busca uniforme tem a menor eficiência dos três, mas garante a solução ótima e busca completa. Por fim, o algoritmo A* tem uma média eficiência, contudo uma alta eficácia, pois, sendo a heurística admissível, garante solução ótima e busca completa.

Foram realizados três testes: um com o puzzle desordenado aleatoriamente, um com o puzzle ordenado e outro ordenado de maneira decrescente, representando o pior caso. As interfaces com os estados iniciais estão apresentadas abaixo.

Na tabela a seguir, estão descritos os resultados obtidos para cada situação apresentada anteriormente.

Tabela 1. Comparação entre os desempenhos dos algoritmos de busca

Tipo de ordenação	Algoritmo	Movimentos necessários para chegar à solução	Tempo de execução (em ms)
Desordenado aleatoriamente	Busca Gulosa	30	3,46
	Busca Uniforme	24	731,01
	Busca A*	24	15,43
Ordenado de maneira descrescente (pior caso)	Busca Gulosa	58	3,23
	Busca Uniforme	30	1.261,22
	Busca A*	30	144,04
Ordenado (melhor caso)	Busca Gulosa	0	0,04
	Busca Uniforme	0	0,04
	Busca A*	0	0,04

Conforme mostra a tabela, o algoritmo de Busca Gulosa se mostrou o mais eficiente, pois apresentou o menor tempo de compilação, entretanto não apresentou a solução ótima, ou seja, ele não realiza o menor número de movimentações possível. Por outro lado, a Busca Uniforme fornece a solução ótima, mostrando grande eficácia, porém é a que mais demora para exibir o resultado, ou seja, possui baixa eficiência. Por fim, a Busca A* apresenta a solução ótima e, por mais que seja um algoritmo mais lento quando comparado à Busca Gulosa, ainda apresenta um valor satisfatório, mostrando alta eficácia e uma eficiência considerável.

5. Conclusão

O algoritmo A*, para os testes realizados, se mostrou com o melhor custo-benefício, uma vez que ele fornece a solução ótima, caso haja solução, e a busca completa em um tempo razoável que, por mais que seja mais demorado que a busca gulosa, não apresenta um tempo médio de execução tão longo quanto a busca uniforme que também apresenta a solução ótima, ou seja, a mesa solução do A*. Por fim, é importante salientar que os tempos médios de execução dependem do estado inicial do puzzle, podendo os algoritmos apresentar tempos maiores ou menores que os apresentados e ter seus tempos mais aproximados, ou mais afastados uns dos outros, como é o caso da solução para o melhor caso em que nenhuma movimentação é feita, tendo o tempo de execução próximo de zero para os três algoritmos, porém, para os demais casos, a solução ótima, se existir e dada uma heurística admissível, será a mesma.