Inteligência Artificial - Projeto

Grupo 41

1 Introdução

Este projeto da unidade curricular de Inteligência Artificial tem como objetivo desenvolver um programa em *Python* que resolva o problema Takuzu utilizando técnicas de procura de IA.

2 Resolução de Problemas

2.1 Problema de Satisfação de Restrições

Dada a natureza restritiva do *Takuzu*, decidimos modelar o jogo como um problema de satisfação de restrições, com o objetivo de reduzir o *branching factor* da árvore de procura.

Primeiramente, definimos as variáveis como cada uma das posições do tabuleiro. O domínio de cada uma das variáveis está contido no conjunto $\{0, 1\}$. As variáveis que representam as posições livres (denominadas **variáveis livres**), apresentam esse mesmo domínio, enquanto que as variáveis que representam uma posição já ocupada têm um domínio unitário, correspondente ao valor dessa posição.

Durante a procura, são utilizadas as restrições do jogo, listadas no enunciado, de forma a reduzir os domínios da variáveis livres. Sempre que uma variável livre vê o tamanho do seu domínio reduzido a 1, a ação correspondente é retornada, na função `actions()`. Ao selecionarmos a variável com o menor domínio, estamos a aplicar a heurística dos **Valores Remanescentes Mínimos**.

Quando, a partir das restrições, não podemos realizar nenhuma inferência, de modo a reduzir o domínio das variáveis livres, vê-mo-nos obrigados a *explorar* as duas possibilidades para os valores daquela posição. Assim, retornamos o conjunto das duas ações, correspondentes aos valores `0` e `1`, fazendo com que a procura *explore* as duas possibilidades.

É importante notar que, a cada nível de profundidade da árvore de procura, apenas atribuímos valores a uma única variável, visto que a ordem de atribuição neste problema

é irrelevante. Mais explicitamente, atribuir o valor `1` à variável `A` e depois o valor `0` à variável `B` é equivalente a atribuir o valor `0` à variável `B` e depois atribuir o valor `1` à variável `A`.

Todos os procedimentos listados acima contribuíram para reduzir o número de nós gerados, durante a procura.

2.2 Heurísticas

A função heurística por nós implementada corresponde ao número de posições livres no tabuleiro. Esta é uma heurística admissível, pois nunca sobrestima o custo de chegar ao objetivo. É de notar que o tempo de computação da heurística é constante, pois é guardado o número de posições livres do tabuleiro respetivo a cada estado.

3 Comparação de Procuras

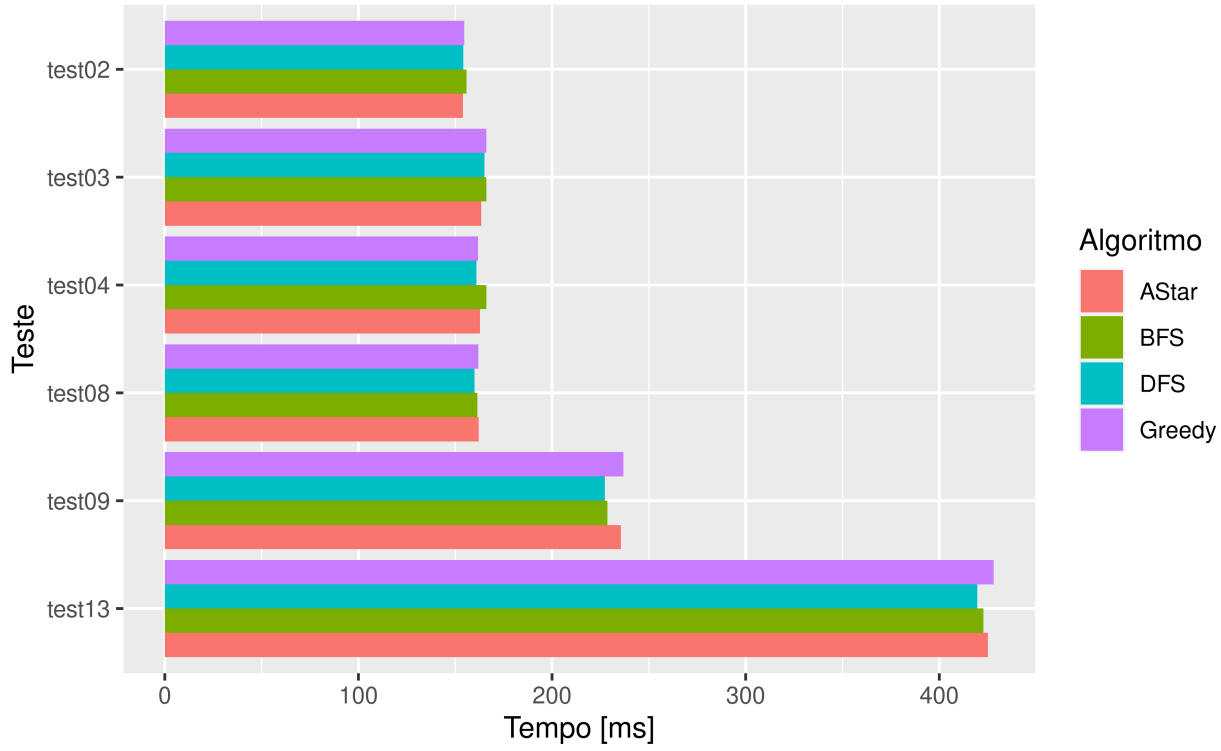
Comparando as procuras, com recurso aos gráficos seguintes, observamos que a `DFS` é o algoritmo que melhor se comporta. Isto seria de esperar porque, priorizando a procura em profundidade, a procura alcança primeiro o nó solução, em relação a um procura em largura por exemplo, pois não tem de explorar todos os nós de profundidades inferiores à profundidade do nó solução. Embora a `DFS` não seja nem completa nem ótima, estes fatores não são relevantes para o problema pois, respetivamente, assume-se que existe sempre solução e apenas se quer a solução do problema, e não o *caminho* até à solução.

Uma procura em largura seria teoricamente mais demorada pois, dada a natureza do problema, sabemos que todas as soluções se encontram à mesma profundidade da árvore de procura. Logo, expandir a árvore em largura gera todos os nós a profundidade menor do que a profundidade do nó solução. O facto de a `BFS` ser completa e ótima em nada é relevante para o problema, como referido acima.

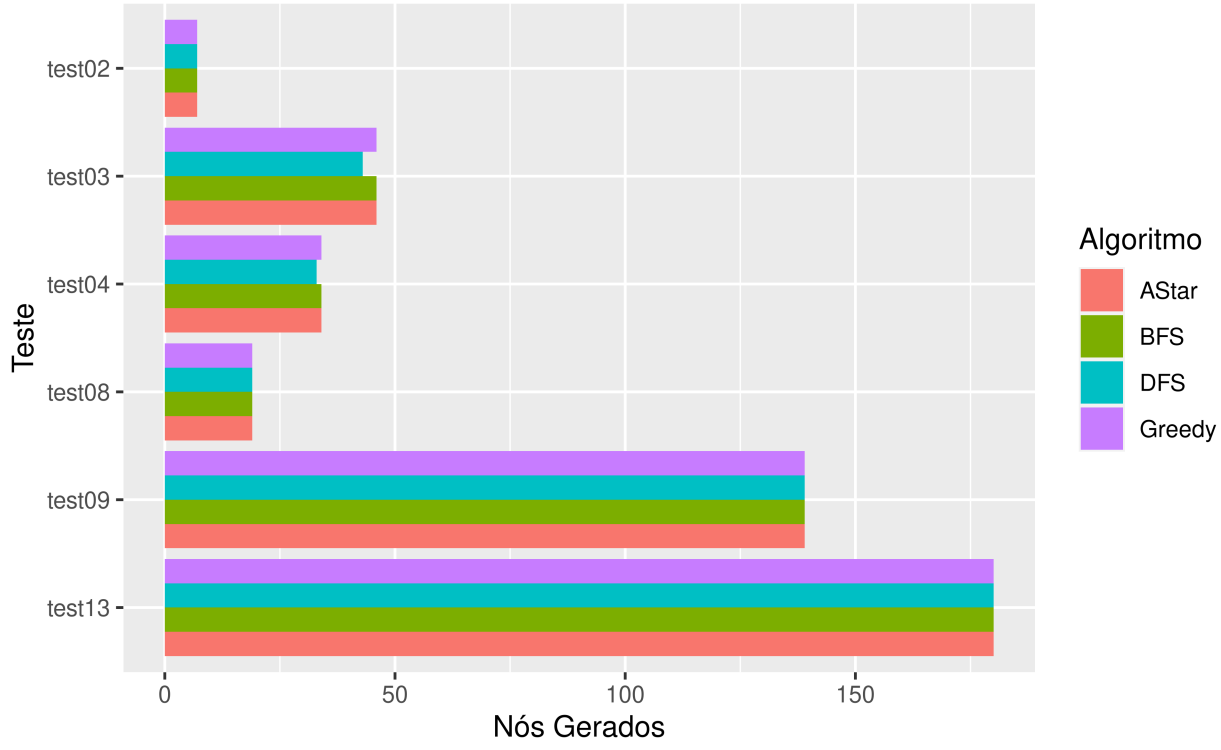
Conclusões acerca dos algoritmos `A*` e `Greedy Search` são difíceis de inferir pois dependem fortemente da heurística implementada. Observa-se que as duas procuras têm resultados semelhantes.

De qualquer forma, todas as procuras parecem ser boas opções, sobretudo porque a função `actions()` reduz bastante a árvore de procura, tornando-a muito pequena, o que faz com que os algoritmos se comportem de forma semelhante.

3.1 Tempo de Execução



3.2 Nós Gerados



3.3 Nós Expandidos