Relatório IA – Takuzu (21/22 P4)

José Cutileiro 99097 & Gonçalo Silva 96925 (tp16)

Parte 0: O nosso código

O nosso código tem como primeiro objetivo tratar da parte da concretização e análise do problema, e em segundo lugar tentamos guiar os demais algoritmos durante o processo da escolha e avaliação das ações. Isto acelara bastante a busca da solução diminuindo o tempo e a memória utilizados.

```
def actions(self, state: TakuzuState):
     [...]
     # Esta função guia os algoritmos na escolha das ações

def goal_test(self, state: TakuzuState):
     [...]
     # Esta função verifica se o estado é solução, caso não seja
     # o objetivo é fazer com que esta função devolva False o mais
     # rapidamente possivel
```

Guiar os algoritmos de procura: [Actions]

Para guiar os algoritmos em primeiro lugar percebemos como jogar o jogo da maneira mais eficaz possivel. Percebemos que nem todas as restrições teem a mesma probabilidade de aparecer. Por exemplo a restrição de "metade dos números ser 1 e a outra ser 0" era a restrição mais recorrente dado que aparece em todas as linhas/colunas por preencher. Sendo assim decidimos que as restrições devem aparecer por ordem de chance de aparecimento no problema. Sendo assim

```
     Nota: Onde está 1's e 0's, o reciproco também é verdade

     # Restrição 0

     Metade 1's -> preencher o resto a 0's

     # Variação para impares (Metade+1 1's -> resto a 0's)

     # Restrições 1 e 2

      >>> Adjacentes iguais -> preencher contrário às adjacentes

      >>> Ver duplas -> preencher contrário às duplas

      Nota: Duplas -> duas em cima/baixo/esquerda/direita
```

Guiar os algoritmos de procura: [Goal test]

É importante para ajudar na eficácia do algortimo, perceber quais são as restrições que mais falham durante o processo de procura da solução e aplicar estas

restrições por ordem de modo a que o goal teste caso falhe, falhe o mais rápidamente possivel. Também devemos ter em conta a complexidade da operação. Em grupo chegámos a esta ordem:

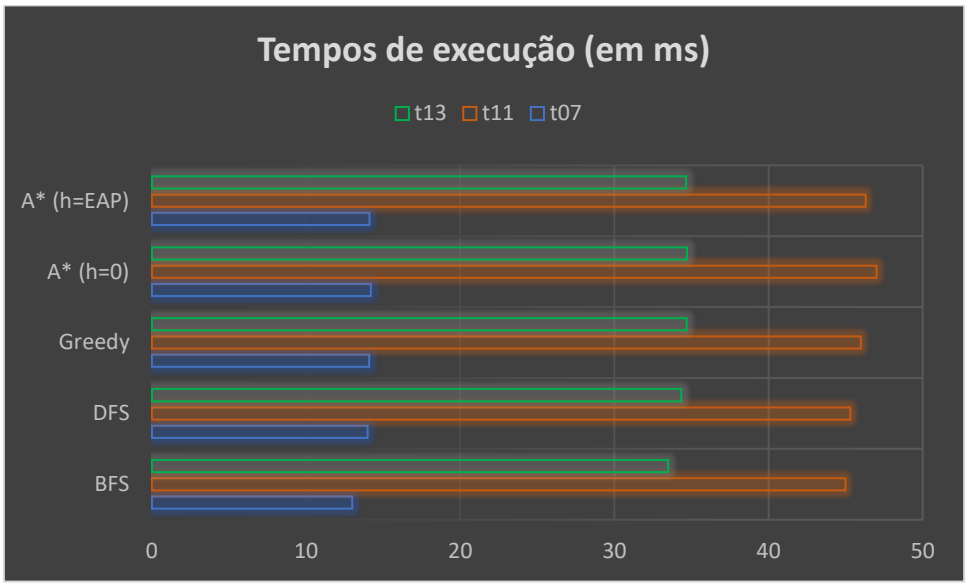
```
1. Por preencher
2. Linhas/colunas iguais e número de 1's e 0's iguais
3. Não desrespeita as condições do problema Takuzo (referidas no
   enunciado)
```

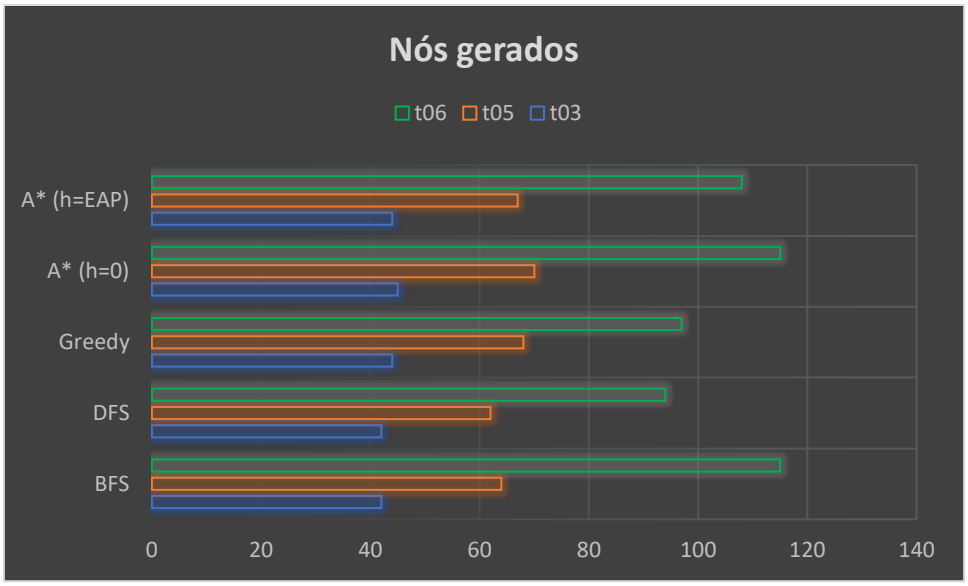
Parte 1: Dados observados (Sucessores/Estados/Tempo)

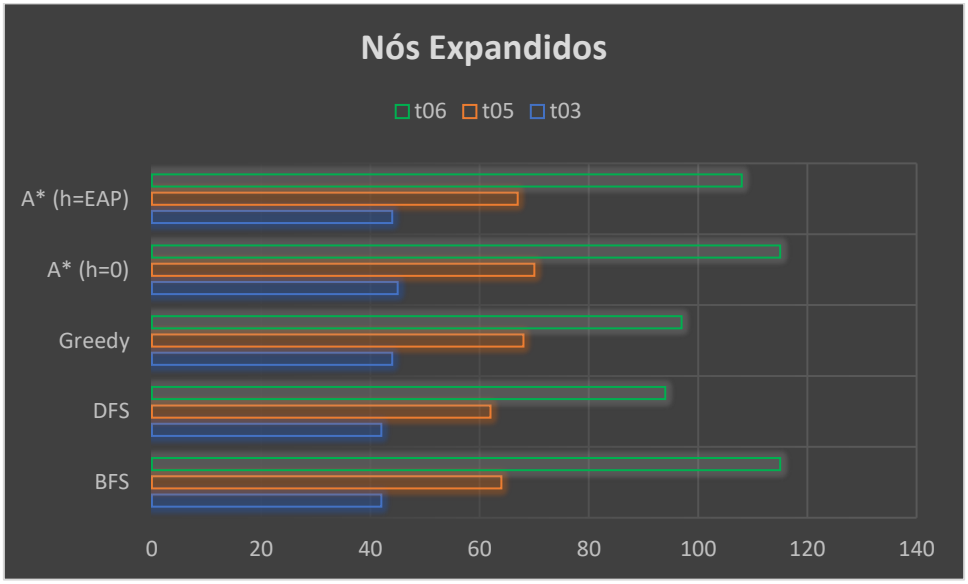
Notas: Heuristica para a avaliação do astar: *h=0* (para testes rápidos)

Como obter tempos de cada algoritmo? >>> Fizemos uma ligeira alteração no *search.py*, na função *compare_searchers* (na função do) adicionámos um temporizador.

```
start_time = time.time()
[...]
print("Time: %s seconds" % (time.time() - start_time))
```







Nota: Os testes apresentados nos gráficos são aqueles que apresentam alterações significativas. Para testes de tempos foram os testes 13, 11 e 7 e para testes de nós foram os testes 6,5 e 3.

Nota: h=EAP (Número de espaços posições livres no tabuleiro) (reparar que esta heuristica é admissivel pois nunca irá sobrestimar o custo verdadeiro)

Parte 2: Conclusões sobre resultados

As comparações entre algortimos são pouco significativas dado que o n em teste não é muito grande (sendo no máximo 31) e o branching factor também é pequeno sendo no máximo 2 (e em muitos casos será 1). Ainda assim podemos reparar em ligeiras alterações nos testes que estão nos gráficos acima. Sendo que a **nivel de tempos** o prémio vai para a BFS, seguido da DFS, Greedy, A* com heuristica dos espaços livres e A* com heuristica = 0. Já a nivel de **estados** (expandidos e gerados) o prémio vai para a DFS seguido da BFS, A* com heuristica dos espaços livres, Greedy e A* com heuristica = 0. Todos os algoritmos chegaram à solução correta do problema.