Análise de Algoritmos de Busca na Solução de um Rubik's Cube

PROTZEN, Matheus K., UCPel

Resumo

Existem diferentes algoritmos computacionais que podem ser utilizados para resolver um cubo. Neste trabalho, é feita a aplicacão dos algoritmos de Busca em Largura (BFS), Busca em Profundidade Iterativa (IDFS), Busca de Custo Uniforme (UCS) e A*, aplicados ao problema de Rubik's com a finalidade de chegar ao seu estado final. Para isso, é feita uma analise do tempo, do desempenho e do número de movimentos necessários para solucionar o cubo. Os resultados mostram que os algoritmos BFS e UCS são menos eficientes que os algoritmos IDFS e A*, se comparados em tempo e desempenho.

Palavras-Chave

Inteligência Artificial, Algoritmos de Busca, Rubik's Cube

1 Introdução

O Cubo Mágico (*Rubik's Cube*, em inglês), inventado em meados dos anos 70 pelo escultor e professor de arquitetura Ernő Rubik, é o mais famoso quebra cabeça tridimensional de todos os tempos. O cubo é listado como uma das 100 mais influentes invenções do século 20 [1] e é considerado o brinquedo mais vendido do mundo [2]. Uma grande variedade de cubos modificados foram projetados, expandindo a família com as mais diversas derivações. Geralmente, o cubo é dividido em duas categorias: o cubo de formato quadrado e o cubo de formato especial. O cubo de formato quadrado, como o próprio nome sugere, é um quadrado cuja ordem pode aumentar (NxNxN).

O menor dos cubos mágicos é chamado *Pocket Cube* (2x2x2). Seguindo a ordem, existe a versão tradicional (3x3x3) e os maiores, como o *Revenge Cube* (4x4x4) e o *Professor's Cube* (5x5x5). O recorde mundial para um cubo 3x3x3 é de 3.47s, e o 0.49s para o cubo 2x2x2. Em sua versão tradicional (3x3x3), o cubo é dotado de seis faces compostas por 9 quadrados cada. Cada face tem por característica uma cor, sendo comumente utilizadas as cores branca, amarela, vermelha, laranja, azul e verde. Cada plano do cubo, ou seja, cada quadrado interno de uma face (3x3x1) pode ser rotacionado 90, 180 ou 270 graus. Além disso, o cubo tradicional possui 8 cantos e 12 meios. Os cantos podem ser arranjados em 8! (ou 40.320) modos distintos, e existem 3^7 (ou 2.187) orientações possíveis, visto que a orientação do canto final (ou do oitavo canto) depende das suas precedentes. Já os meios podem ser arranjados em 12!/2 (ou 239.500.800) modos distintos. Sendo assim, existem 2^{11} (2.048) possibilidades apenas. Esse numero se dá ao fato de que somente 11 arestas podem ser rotacionadas de maneira independente, com a rotação da aresta final (ou da décima-segunda aresta) dependendo de suas precedentes.

A função de custo de cada movimento é sempre 1 e a notação utilizada na solução de um cubo é apresentada da seguinte maneira:

- R representa o movimento da camada direita;

- ***PROTZEN, Matheus K.****: Curso de Engenharia de Computação, Centro de Ciências Sociais e Tecnológicas. Universidade Católica de Pelotas - UCPel.*
  E-mail: matheus.protzen@sou.ucpel.edu.br

- L representa o movimento da camada esquerda;
- U representa o movimento da camada superior;
- D representa o movimento da camada inferior;
- F representa o movimento da camada frontal;
- B representa o movimento da camada traseira;
- M representa o movimento da camada central entre L e R;
- S representa o movimento da camada central entre F e B;
- E representa o movimento da camada central entre U e D;
- R, L, U, D, F, B, M, S e E representam os giros horários;
- R', L', U', D', F', B', M', S', E' representam os giros anti-horários;
- R2, L2, U2, D2, F2, B2, M2, S2, E2 representam giros duplos;
- 2R, 2L, 2U, 2D, 2F, 2B representam os giros nas camadas internas, sendo 2 o número da camada, de fora para dentro, nunca passando de $\frac{n}{2}$.

Para um cubo 3x3 em seu estado final, cada face deverá estar com todos os seus 9 quadrados internos da mesma cor, de modo que a face branca deverá sempre estar oposta a face amarela, bem como a face vermelha oposta da laranja e a face azul oposta da verde. O cubo deve ser desarranjado de maneira aleatória executando um número qualquer de movimentos, e a tarefa é restaurar o cubo ao seu estado final.

Logo, para um cubo 3x3x3, o número de possibilidades de diferentes combinações é $43 * 10^{18}$ (ou $8! * 3^7 * (12!/2) * 2^{11}$). Em um cubo de tamanho NxNxN, o número de movimentos é dado pela formula $n * 9$

2 Algoritmos

Existem diversos algoritmos desenvolvidos para as mais diversas variantes de um problema. As variantes podem incluir bordas direcionadas versus não direcionadas, por exemplo, e frequentemente fazem uso de grafos. Um grafo é um número de nós e arcos que os conectam, e um grafo rotulado tem uma ou mais assinaturas anexadas a cada nó que distingue o nó de qualquer outro nó no grafo. A pesquisa de gráficos é dividida em pesquisa de busca cega e em pesquisa heurística.

Às vezes, a pesquisa de busca às cegas é chamada de pesquisa uniforme, pois não tem conhecimento sobre seu domínio. A única opção que uma pesquisa cega é capaz de fazer é distinguir um estado sem meta de um estado de meta. A busca cega não tem preferência sobre qual estado (nó) pode ser expandido a seguir; diferentemente da pesquisa heurística. A pesquisa heurística é o estudo dos algoritmos e regras de descoberta e desenvolvimento. Heurísticas são regras práticas que podem resolver um determinado problema, mas não garantem uma solução.

Neste trabalho, serão aplicados quatro algoritmos: Busca em Largura (BFS), Busca em Profundidade Iterativa (IDFS), Busca de Custo Uniforme (UCS) e A*. Nota-se que, durante a execução dos aplicativos, se o tempo for superior a 300 segundos, a execução do algoritmo é interrompida.

2.1 Busca em Largura (BFS)

Na teoria de grafos, a busca em largura é uma das estratégias utilizadas para realizar uma busca ou travessia em um grafo ou em uma árvore. Como o próprio nome sugere, essa abordagem envolve percorrer a árvore em largura (em vez de em profundidade). O algoritmo de busca em largura começa examinando todos os nós um nível (algumas vezes chamado de uma camada) abaixo do nó raiz. Se um estado objetivo for encontrado aqui, é relatado sucesso. Caso contrário, a busca prossegue pela expansão de caminhos a partir de todos os nós do nível corrente na direção do próximo nível. Desse modo, a busca continua examinando nós em um determinado nível, relatando sucesso quando um nó objetivo for encontrado e relatando falha se todos os nós tiverem sido examinados e um nó objetivo não tiver sido encontrado.

A busca em largura é um bom método para ser utilizado em situações nas quais a árvore pode ter caminhos muitos profundos, principalmente se o nó objetivo estiver em uma parte mais rasa da árvore. Infelizmente, ela não funciona tão bem quando o fator de ramificação da árvore é muito alto, tais como ao examinar árvores de jogos para jogos como Go e Xadrez. O algoritmo não é bom em árvores onde todos os caminhos levam a um nó objetivo com caminhos de comprimentos parecidos. Em situações como esta, a busca em profundidade funciona muito melhor, pois identificaria um nó objetivo quando atingisse o final do primeiro caminho examinado.

Para encontrar a complexidade de tempo e espaço da busca em largura, devemos supor que os nossos nós não objetivos tenham b sucessores (sendo b é o fator de ramificação), e gerar cada sucessor pai. Essa tarefa tem complexidade assintótica constante. Em seguida, a raiz da arvore de busca gera b nós e consome b tempo. Assim, cada um dos nós b, com profundidade 1, gera b nós e consome b tempo, e assim por diante até que a meta de profundidade d seja atingida. A complexidade assintótica de tempo é, portanto, O(b^d).

2.2 Busca em Profundidade Iterativa (IDFS)

A busca em profundidade iterativa é uma estratégia geral, usada com frequência em combinação com a busca em profundidade, que encontra o melhor limite de profundidade. Ela faz isso aumentando gradualmente o limite – primeiro 0, depois 1, depois 2 e assim por diante – até encontrar um objetivo. A busca em profundidade iterativa é análoga à busca em largura, pelo fato de explorar uma camada completa de novos nós em casa iteração, antes de passar para a próxima camada, combinando assim os benefícios da busca em profundidade e da busca em largura.

Como na busca em largura, ele é completo quando o fator de ramificação é finito, e ótimo quando o custo de caminho é uma função não-decrescente da profundidade do nó. A busca em profundidade iterativa pode parecer um desperdício, por que os estados são gerados várias vezes. Na verdade, esse custo não é muito alto porque, em uma árvore de busca c om o mesmo (ou quase o mesmo) fator de ramificação em cada nível, a maior parte dos nós estará no nível inferior, e assim não importa muito se os níveis superiores são gerados várias vezes. Em uma busca como essa, os nós no nível inferior são geradas uma vez, os do penúltimo nível inferior são gerados duas vezes e assim por diante, até os filhos da raiz, que são gerados d vezes.

Na busca em profundidade iterativa, a complexidade assintótica de tempo do algoritmo é O(b^d), onde b é o fator de ramificação e d é a profundidade da solução. Além disso, sua complexidade em espaço é O(d).

2.3 Busca em Custo Uniforme (UCS)

A busca em custo uniforme é uma técnica que usa a função de avaliação g(nó), a qual para um dado nó avalia o custo do caminho levando aquele nó. Em outras palavras, isto é um algoritmo A* no qual h(nó) é zerado. A cada estágio, o caminho que tenha menor custo até então é expandido. Deste modo, o caminho gerado é provável que seja aquele com menor custo total, mas isto não é garantido. Para encontrar o melhor caminho, o algoritmo precisa continuar a execução após uma solução ser encontrada e, se uma solução melhor for encontrada, ela deverá ser aceita no lugar da solução anterior.

A busca de custo uniforme é completa e é ótima, desde que um custo de um caminho cresça monotonicamente. Em outras palavras, se para cada nó m que tenha um sucessor n, for verdade que g(m) $<$ g(n), então a busca de custo uniforme é ótima. Se for possível para o custo de um nó ser menor que o custo do seu pai, então a busca de custo uniforme pode não encontrar o melhor caminho.

Para encontrar a complexidade de tempo e espaço da busca de custo uniforme, precisamos do caminho de custo ao invés da profundidade d. Se C* é o custo ideal da solução, e cada passo custa pelo menos e, então a complexidade assintótica em tempo é O($b^{[1+(C*/e)]}$). Ela se mostra muito melhor, se comparada com o algoritmo BFS. Quando todos os caminhos de custo são iguais, então o custo ideal é a mesma do algoritmo BFS, exceto pela profundidade.

2.4 A*

A busca A* (chamada de A estrela) é a um dos algoritmos mais famosos na identificação de caminhos. Ele é um algoritmo de busca genérica e se baseia na análise dos nós através da combinação de g(nó), que é o custo para alcançar o nó, e h(nó), que é o custo para ir do nó ao objetivo. Sendo assim, o algoritmo usa a seguinte notação:

$$f(nó) = g(nó) + h(nó) \tag{1}$$

F(nó) é a chamada função de avaliação baseada em caminho. Ao operar o A*, a função f(nó) é avaliada para nós sucessores e caminhos expandidos utilizando os nós que tenham os menores valores de f. Se o custo h(nó) for sempre uma subestimativa da distância de um nó a um nó objetivo, então o algoritmo A* será ótimo: é garantido encontrar o caminho mais curto até um estado objetivo.

O A* é descrito como sendo otimamente eficiente, no sentido de que, para encontrar o caminho até o nó objetivo, ele expandirá o mínimo de caminhos possível. Mais uma vez, essa propriedade depende do custo h(nó) ser sempre uma subestimativa. Vale observar que, ao executar o algoritmo A*, nem sempre será garantido encontrar a solução mais curta, pois os valores estimados para h(nó) não são todos subestimativas. Em outras palavras, a heurística que está sendo utilizada pode não ser admissível. Se for utilizada uma heurística não admissível para h(nó), então o algoritmo será chamado apenas de A.

Sendo assim, podemos afirmar que o algoritmo A* só será útil quando fornecer uma subestimativa de custo verdadeiro até o objetivo, e será ótimo e completo apenas se, ao encontrar uma solução, essa seja garantidamente a melhor solução. Além disso, vale ressaltar que o algoritmo A* é análogo à busca em largura. De fato, a busca em largura pode ser considerada como um caso especial do A*, no qual o custo h(nó) é sempre 0, logo, f(nó) = g(nó), onde cada caminho direto entre um nó e o seu sucessor imediato tem custo 1.

2.4.1 Heurísticas

Heurística é o nome dado a pesquisa realizada por meio da quantificação de proximidade a um determinado objetivo. Diz-se que se tem uma boa (ou alta) heurística se o objeto de avaliação está muito próximo do objetivo; diz-se de má (ou baixa) heurística se o objeto avaliado estiver muito longe do objetivo. Etimologicamente a palavra heurística vem da palavra grega Heuriskein, que significa descobrir (e que deu origem também ao termo Eureca).

Um algoritmo aproximativo (ou algoritmo de aproximação) é heurístico, ou seja, utiliza informação e intuição a respeito da instância do problema e da sua estrutura para resolvê-lo de forma rápida.

Neste estudo, foram usadas duas heurísticas, que serão explicadas a seguir.

2.4.1.1 Contagem

Para um cubo 3x3x3, cada face pode receber uma pontuação de zero a oito, sendo zero se todos os quadrados forem da cor referente ao primeiro quadrado dentro de uma face e oito se as cores forem diferentes, repetindo esse processo para cada face. Esta avaliação pode ser de zero para um cubo solucionado e no máximo quarenta e oito para um cubo desordenado.

Nos testes efetuados, a heurística de contagem será representada por A* 0.

2.4.1.2 Cantos

Para um cubo 3x3x3, foi aplicada a ideia da distância de Manhattan. A distância de Manhattan é o número de movimentos que devemos fazer para chegar de um ponto a a um ponto b. Neste sentido, aplicando a ideia da distância, podemos ter o valor zero caso o quadrado da face esteja em sua posição correta e valor um quando o quadrado estiver a um movimento de sua solução. Qualquer outra posição pode ser resolvida com apenas dois passos. Todavia, a heurística não é aplicada a todos os cantos. Cada face pode ter somente um canto onde a heurística é fixada.

Nos testes efetuados, a heurística de contagem será representada por A* 1.

3 Resultados

Nesse estudo, comparamos o uso dos algoritmos de Busca em Largura (BFS), Busca em Profundidade Iterativa (IDFS), Busca de Custo Uniforme (UCS) e A*, aplicados a cubos de diferentes complexidades. Usaremos cubos nos seguintes tamanhos: 2x2x2, 3x3x3, 4x4x4 e 5x5x5. Para cada tamanho de cubo, foram usadas diferentes complexidades: 1, 2, 3, 4 e 5 movimentos para embaralhar.

Para um cubo de tamanho 2x2x2, os resultados obtidos foram os seguintes:

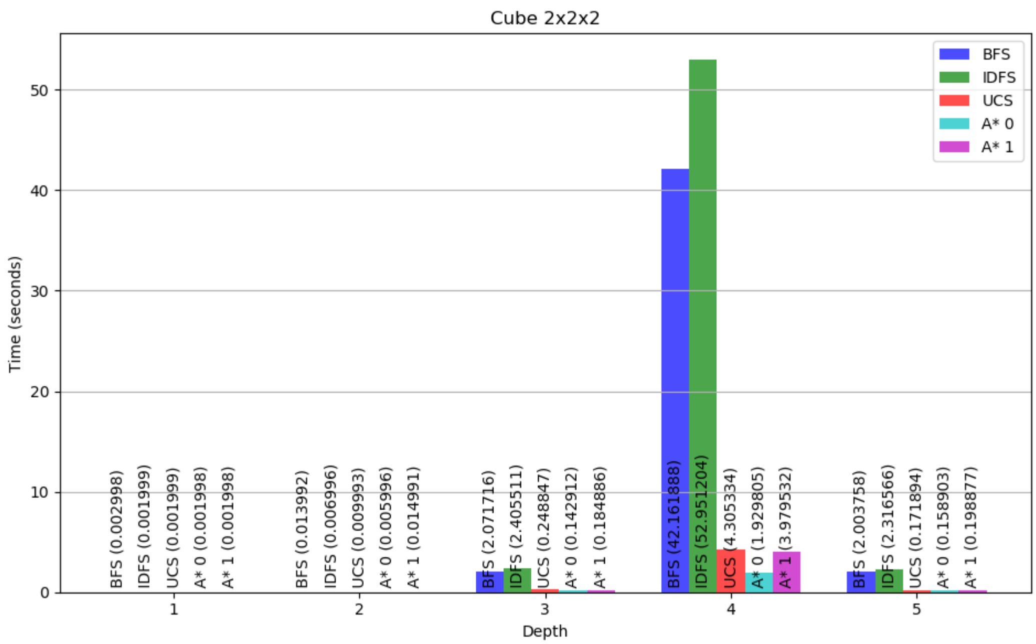

Figura 1: Tempo

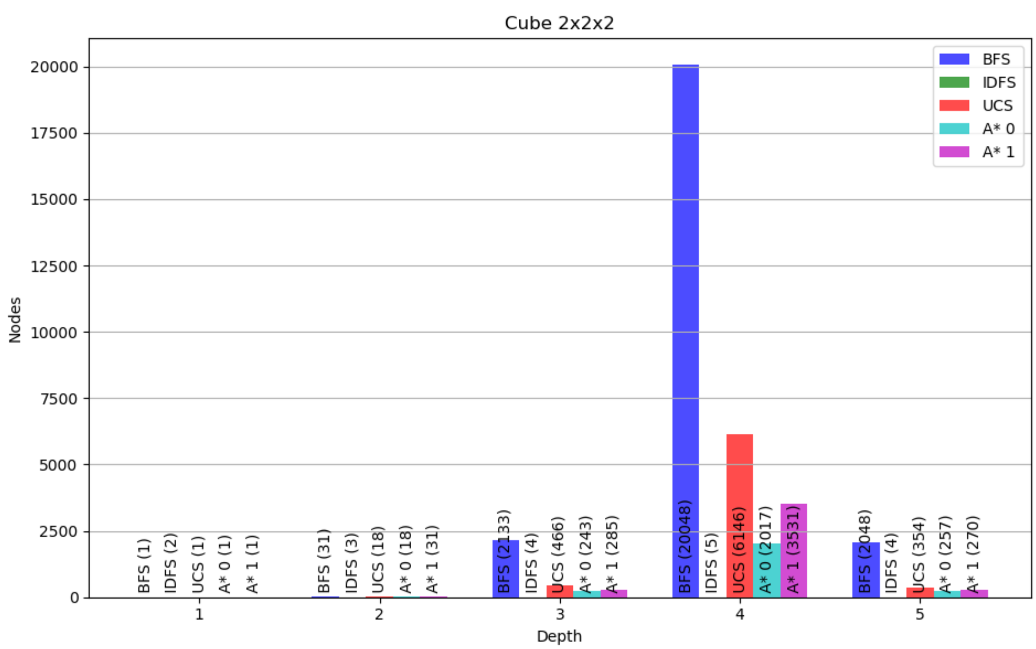

Figura 2: Nodos armazenados

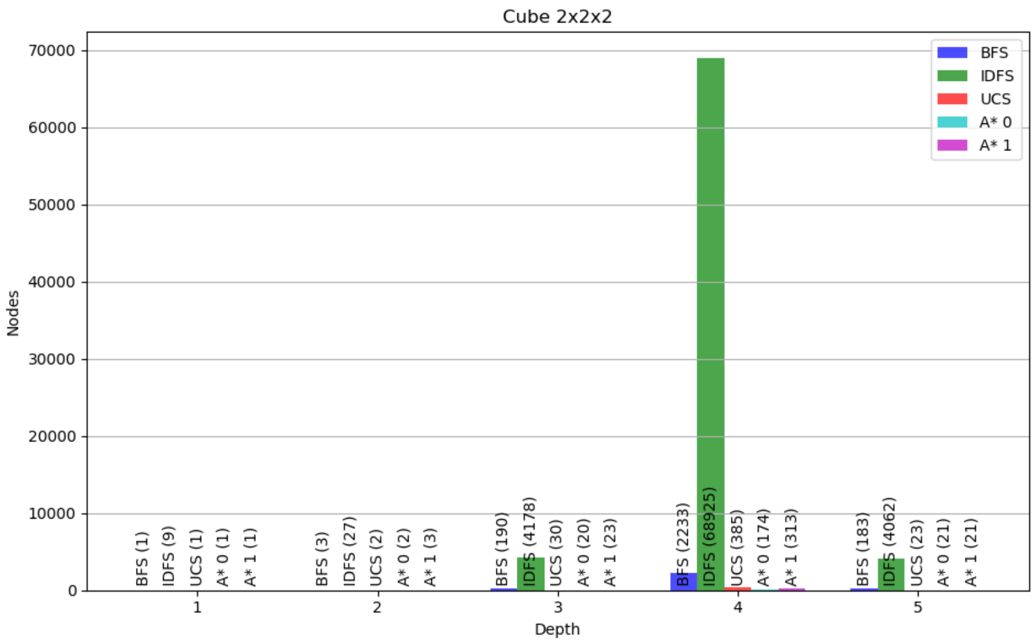

Figura 3: Nodos visitados

Esses dados demonstram que o algoritmo A* com ambas as heurísticas são mais eficiente que todos os demais.

Para um cubo de tamanho 3x3x3, os resultados obtidos foram os seguintes:

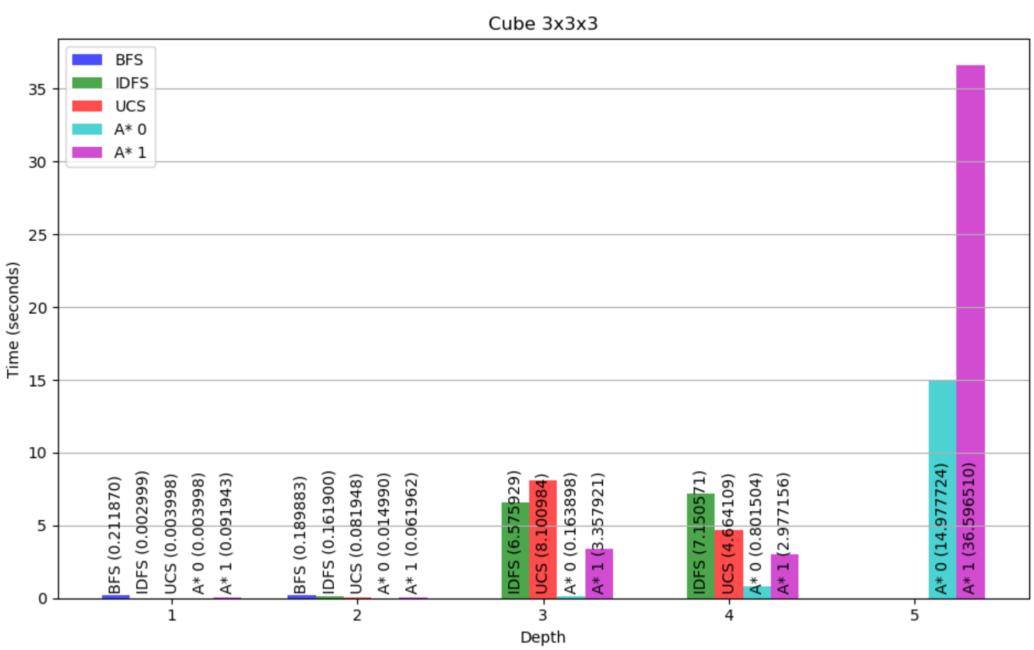

Figura 4: Tempo

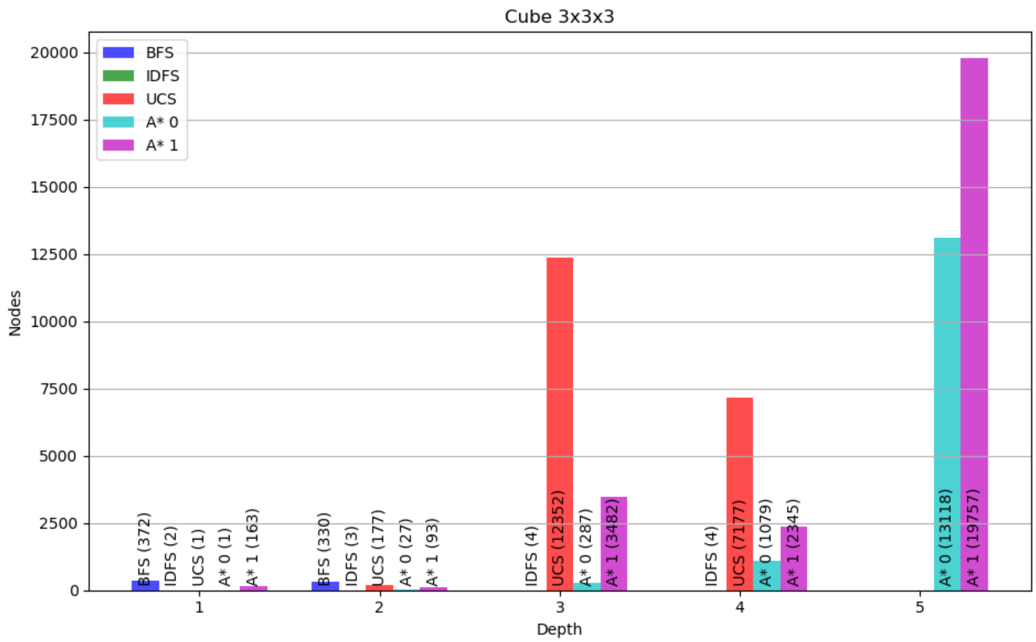

Figura 5: Nodos armazenados

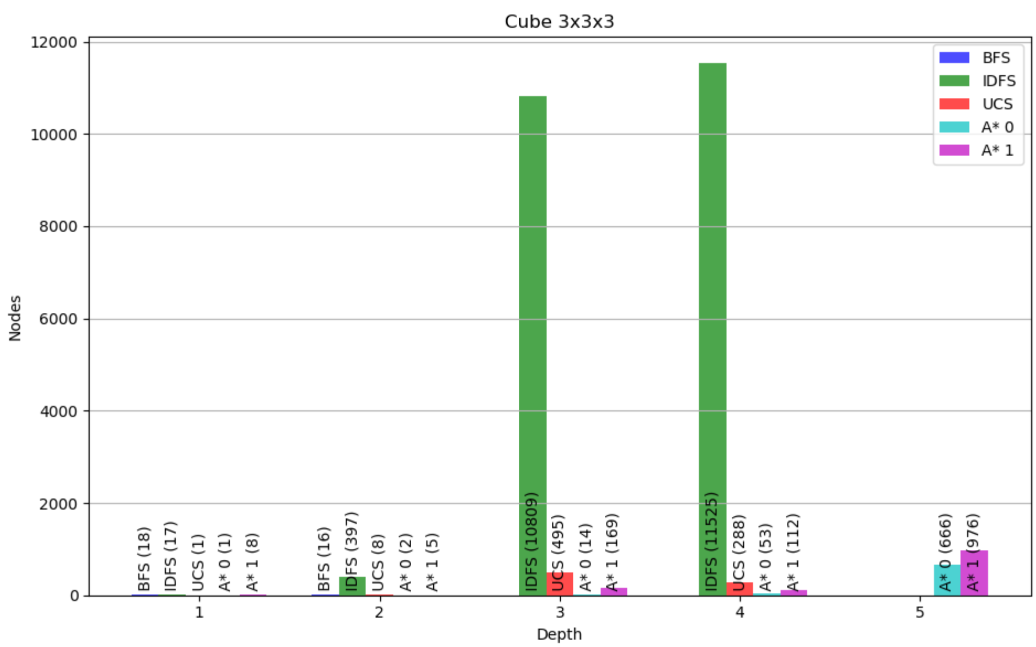

Figura 6: Nodos visitados

O algoritmo A* com a heurística 0 se mostrou mais eficiente em todas as métricas. O algoritmo BFS, partindo da terceira profundidade, não apresentou resultados no tempo limite imposto.

Para um cubo de tamanho 4x4x4, os resultados obtidos foram os seguintes:

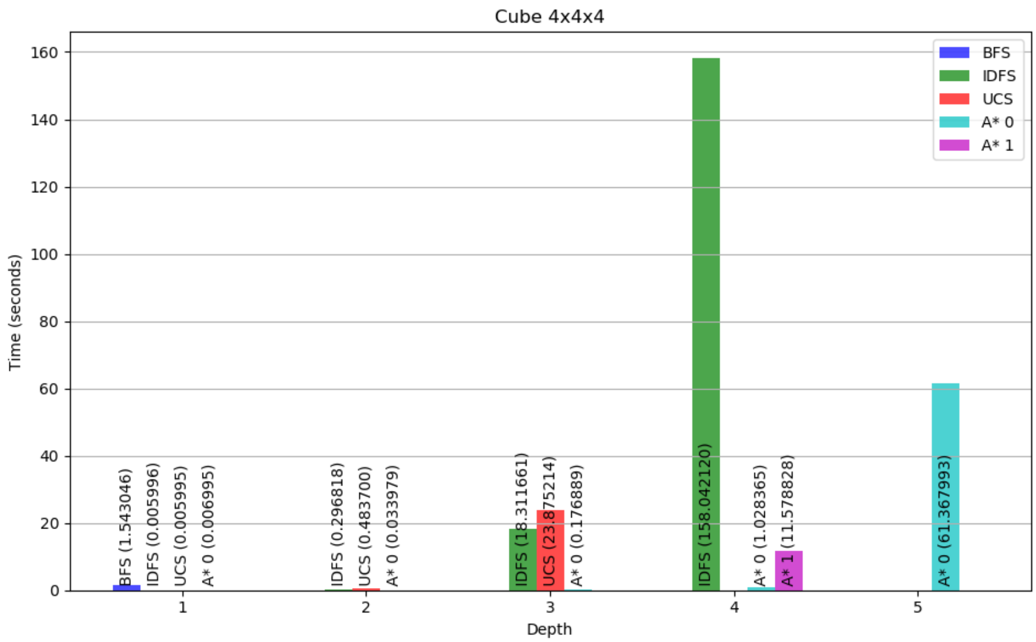

Figura 7: Tempo

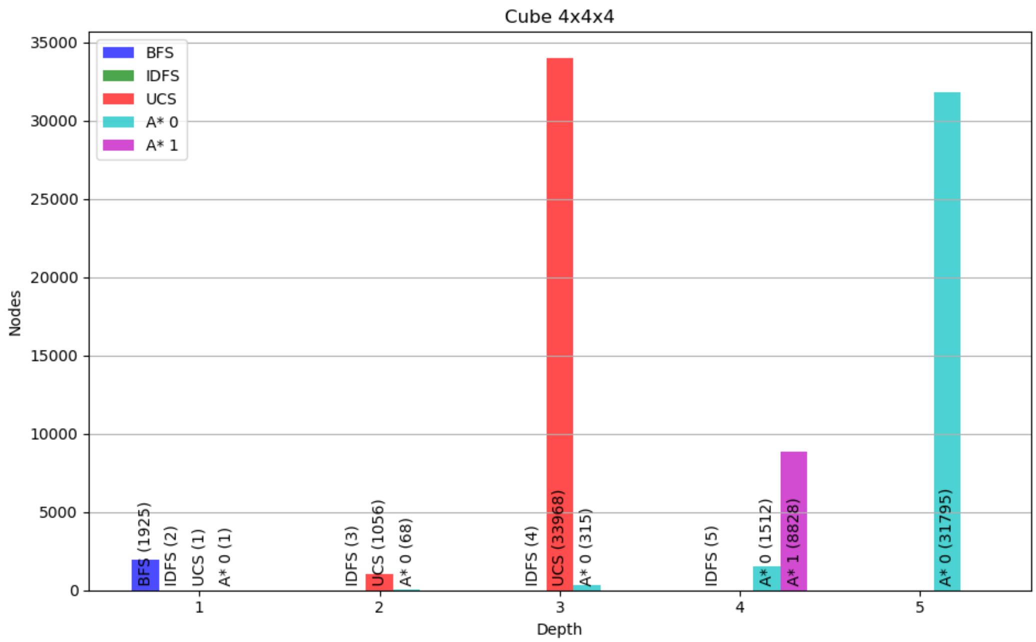

Figura 8: Nodos armazenados

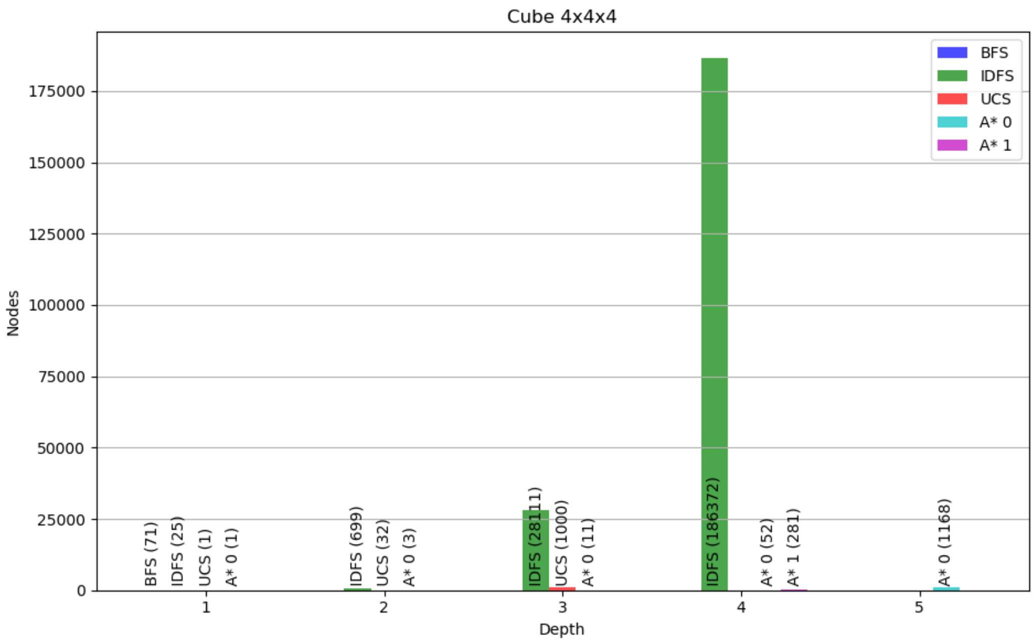

Figura 9: Nodos visitados

Os dados mostram que o A* com a heurística 1 rodou somente na quarta profundidade e o A* com a heurística 0 se mostrou mais eficiente. O algoritmo BFS só exibiu resultados com profundidade um, assim como o UCS demonstrou resultados nas profundidades um, dois e três e o IDFS nas profundidades um, dois, três e quatro.

Para um cubo de tamanho 5x5x5, os resultados obtidos foram os seguintes:

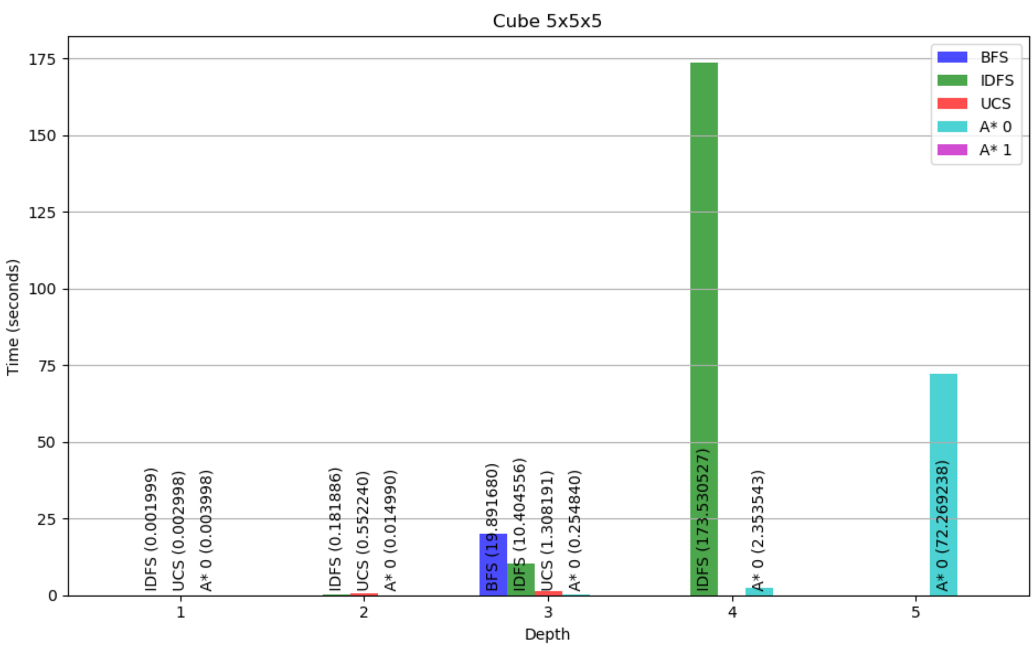

Figura 10: Tempo

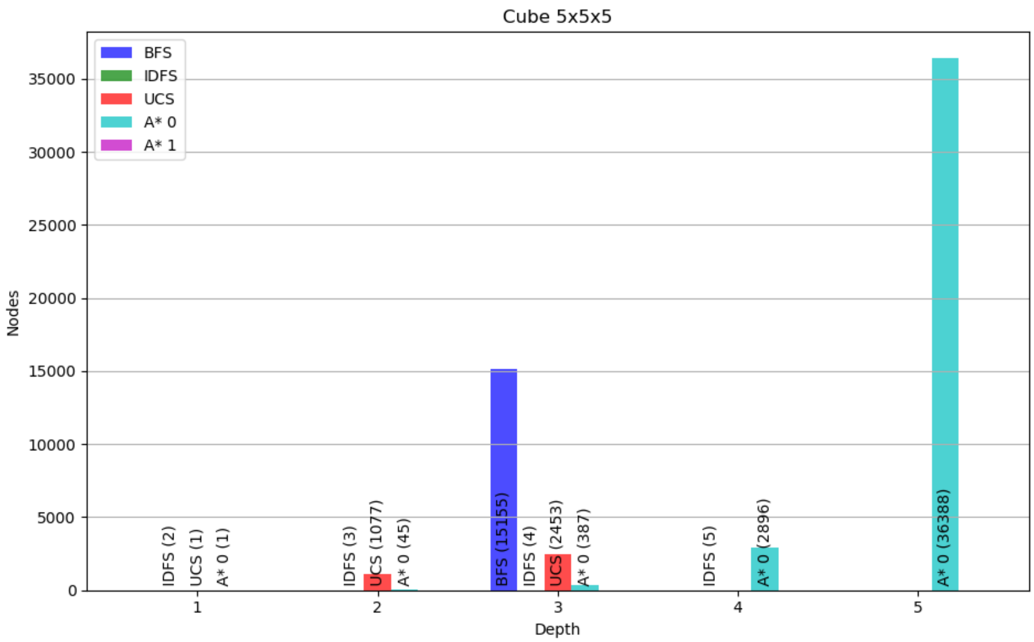

Figura 11: Nodos armazenados

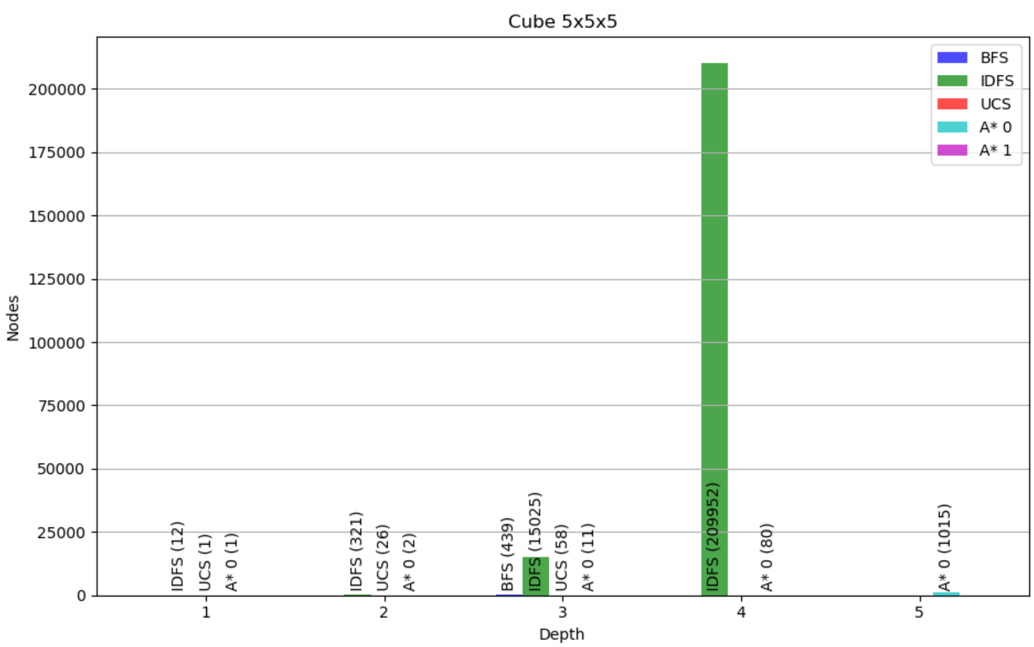

Figura 12: Nodos visitados

Em um cubo com esse tamanho, o algoritmo BFS foi incapaz de apresentar resultados. O algoritmo UCS se manteve com a mesma estatística da resolução do cubo 4x4x4, assim como o IDFS. O algoritmo A* com heurística 1 não apresentou resultados e o A* com heurística 0 apresentou os melhores resultados.

Assim, conclui-se que o desempenho do algoritmo A* com a heurística 0 foi satisfatório e apresentou os melhores resultados para todos os tamanhos de cubo testados. Entre os demais algoritmos, podemos constatar que o IDFS é o algoritmo cujo numero de nodos visitados é maior, se contrapondo com o seu uso de memória, sendo ele otimizado.

Referências

[1] V D Stephen. *Inventing the 20th Century: 100 Inventions That Shaped the World*. New York University Press, New York, 2002.

[2] Rubik's cube 25 years on: crazy toys, crazy times. https://www.independent.co.uk/news/science/rubiks-cube-25-years-on-crazy-toys-crazy-times-5334529.html, Setembro 2011.

[3] S. NORVIG, P.; RUSSEL. *Inteligência Artificial 3ª Edição*. Elsevier Editora Ltda, 2013.

[4] C. E.; RIVEST R. L. e STEIN C. CORMEN, T. H.; LEISERSON. *Algoritmos: Teoria e Prática 3ª Edição*. Elsevier Editora Ltda, Rio de Janeiro, RJ, 2012.