Trabalho Prático 1

Resolvendo Sudoku com Busca em Espaço de Estados

ALUNO: GUILHERME DRUMOND ROSA
MATRÍCULA: 2020041531

Modelagem do Problema

Este trabalho prático foi desenvolvido com o intuito de fazer um comparativo entre os diferentes algoritmos de busca com e sem informação para resolução do quebra-cabeça Sudoku. Dessa maneira, o problema foi modelado a partir da construção de uma árvore de busca (a ser percorrida pelos algoritmos) em que cada nó representa um estado do quebra-cabeça, ou seja, uma configuração específica da matriz, e cada transição é responsável por preencher uma célula vazia com algum número válido.

Estruturas de Dados e Componentes de Busca

O trabalho todo faz uso de uma grande classe Node que é instanciada a cada nó criado. Nesse momento, o atributo matrix é inicializado com a configuração passada por parâmetro, o atributo children é inicializado como uma lista vazia e o atributo g (referente ao valor acumulado de custo para a heurística utilizada em A*) é inicializado com 0. Essa classe conta com uma série de funções que chamaremos aqui como componentes de busca, são elas:

- get_zero - retorna os índices do primeiro espaço vazio encontrado na configuração do Nó em questão;
- get_actions - retorna um array com os possíveis valores a serem preenchidos no primeiro espaço vazio encontrado na configuração do Nó em questão (função sucessora);
- solution - retorna verdadeiro se o estado atual configurar uma solução ou falso em caso contrário (função verificadora);
- get_dist_to_goal - inicializa a variável f referente ao número de espaços vazios restantes (de acordo com a heurística utilizada)
- get_less_actions - retorna as coordenadas do espaço vazio com o menor número de opções disponíveis e um array com os possíveis valores a serem preenchidos neste espaço encontrado na configuração do Nó em questão (de acordo com a heurística utilizada);

Funcionamento dos Algoritmos

BFS: Cada nó encontrado é adicionado a uma fila (frontier) e no momento de sua expansão (por meio da função get_less_actions) é removido. A cada iteração verifica se o estado configura uma solução para o problema e quando isso ocorrer, retorna a matriz final e a quantidade de nós visitados. Caso a fronteira esteja vazia e não tenha mais nós a serem expandidos, retorna erro. Dessa forma, garantimos que o nó a ser expandido seja sempre o mais raso ainda não expandido.

IDS: Cada nó encontrado é adicionado a uma pilha (next_frontier) que só será decrementada quando todos os nós da frontier atual forem expandidos (por meio da função get_less_actions), após isso o limite de profundidade aumenta. A cada iteração verifica se o estado configura uma solução para o problema e quando isso ocorrer, retorna a matriz final e a quantidade de nós visitados. Dessa forma, garantimos que o nó a ser expandido seja sempre o mais profundo ainda não expandido em uma profundidade menor que a limite.

UCS: Cada nó encontrado é adicionado a uma fila de prioridades (frontier, como o custo de expansão de um nó é o mesmo para toda expansão, a busca se degenera num bfs) e no momento de sua expansão (por meio da função get_less_actions) é removido. A cada iteração verifica se o estado configura uma solução para o problema e quando isso ocorrer, retorna a matriz final e a quantidade de nós visitados. Caso a fronteira esteja vazia e não tenha mais nós a serem expandidos, retorna erro. Dessa forma, garantimos que o nó a ser expandido seja sempre o mais raso ainda não expandido.

ASS: Cada nó encontrado é adicionado a um array (open) e no momento de sua expansão (por meio da função get_less_actions) é removido. O nó a ser expandido (e consequentemente removido de open) é aquele de menor valor de f, ou seja, aquele que possuir o menor número de espaços vazios restantes. A cada iteração verifica se o estado configura uma solução para o problema e quando isso ocorrer, retorna a matriz final e a quantidade de nós visitados. Caso a fronteira esteja vazia e não tenha mais nós a serem expandidos, retorna erro. Dessa forma, garantimos que o nó a ser expandido siga a heurística determinada graças à informação.

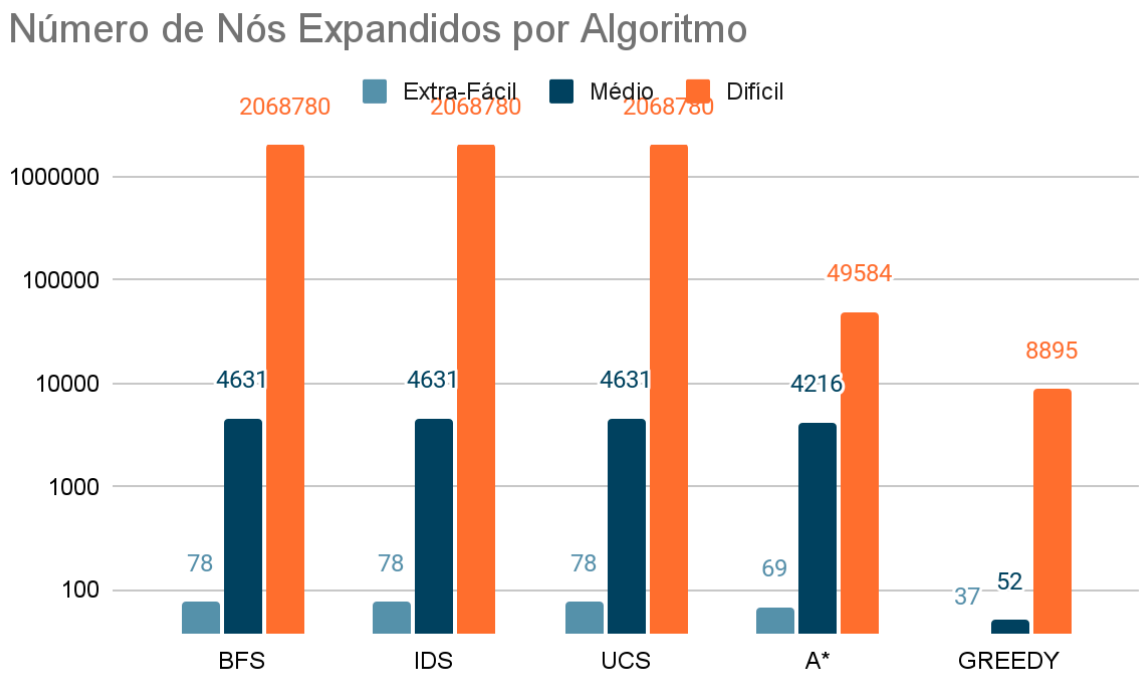
GBFS: Cada nó encontrado é adicionado a uma pilha (frontier) e no momento de sua expansão (por meio da função get_less_actions) é removido. A cada iteração verifica se o estado configura uma solução para o problema e quando isso ocorrer, retorna a matriz final e a quantidade de nós visitados. Além disso, a cada expansão, a escolha da célula a ser preenchida é a de menor opções disponíveis. Dessa forma, garantimos que o nó a ser expandido seja sempre o mais profundo e que a célula a ser preenchida siga a heurística determinada graças à informação.

Heurísticas Escolhidas

Neste trabalho foram utilizadas duas heurísticas, uma para cada algoritmo com informação, são elas:

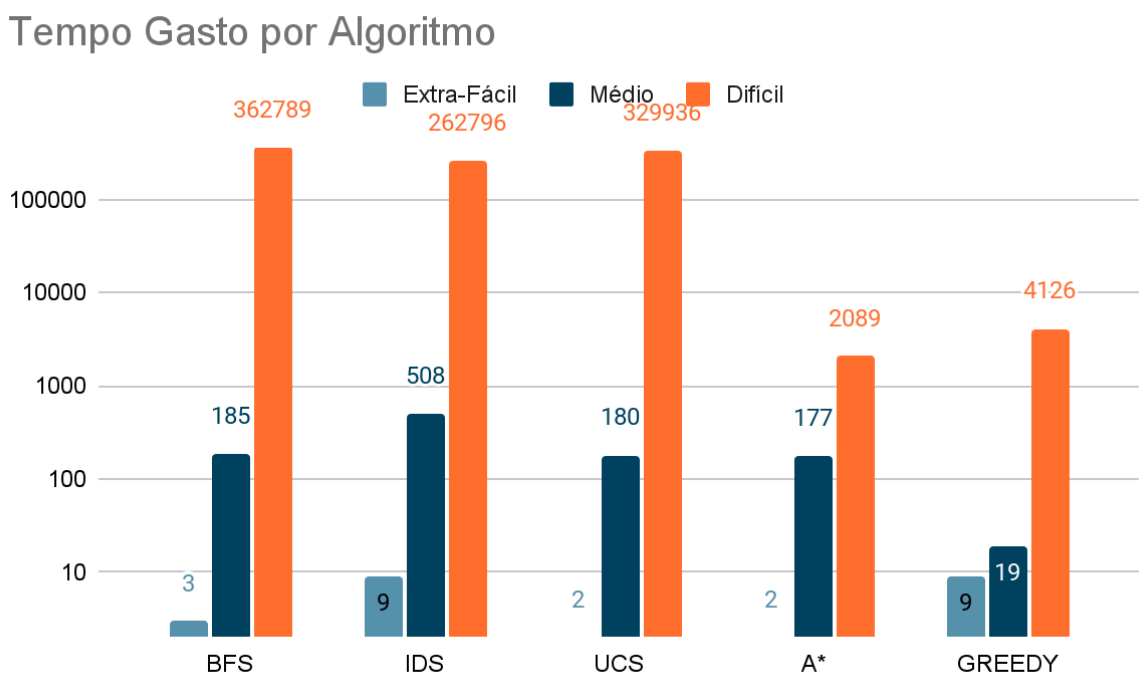
1. Expandimos o nó de menor valor de f, ou seja, aquele que possuir o menor número de espaços vazios restantes (utilizada no algoritmo A*). Essa estratégia foi escolhida pois assim conseguimos otimizar o tempo e espaço gastos na busca, afim de alcançarmos o objetivo mais rapidamente, expandindo sempre os nós que teoricamente estão mais próximos da solução;
2. Preenchemos a célula de menor opções disponíveis, ou seja, aquela que possuir menos números válidos (utilizada no algoritmo guloso). Essa estratégia foi escolhida pois conseguimos minimizar a quantidade de caminhos a serem podados e, consequentemente, chegar à solução expandindo o menor número de nós possível.

Análise Quantitativa



A partir do gráfico acima, podemos notar como os algoritmos de busca sem informação tiveram o mesmo número de nós expandidos em todos os algoritmos. Isso ocorre pois tanto o IDS quanto o UCS se degeneram em um BFS no problema do Sudoku. Para evitar o elevado número de caminhos percorridos repetidamente pelo IDS, foi escolhido que os nós já visitados não fossem visitados novamente, resultando assim em uma espécie de busca em largura, já que a profundidade limite sempre era incrementada em uma unidade. No caso do UCS, o custo de se mover de um estado para outro é sempre o mesmo e, portanto, não apresenta distinção do BFS. Por outro lado, a melhora obtida nos algoritmos com informação é facilmente perceptível. O A* conseguiu reduzir drasticamente o número de nós visitados (em comparação aos anteriores) à medida que o nível de dificuldade aumentava. O guloso, por incrível que pareça, foi o algoritmo de melhor

desempenho, isso graças à heurística utilizada. Como ele sempre buscava nós mais “seguros”, a redução de nós visitados foi drástica.



Analisando agora os tempos gastos por cada um, novamente os sem informação se mantiveram bem parecidos pelos motivos citados anteriormente. Porém, podemos perceber como o IDS se mostrou pior nas dificuldades mais baixas e melhor à medida que o problema se tornava difícil. Os algoritmos com informação foram mais uma vez superiores em relação aos não informados e o A* o melhor de todos para o problema mais complexo.

Conclusão

O problema do Sudoku definitivamente é mais efetivamente resolvido se usarmos heurísticas adequadas que se apoiem nas informações disponíveis. Neste trabalho não foi possível explorar a fundo as individualidades de cada um dos algoritmos sem informação, que no final se assemelhavam a um BFS, isso porque o IDS começou a gastar horas ao percorrer o mesmo caminho repetidas vez (por isso os nós já visitados pararam de ser revisitados) e o UCS se deparava com custo = 1 para cada expansão. De qualquer forma, todos os algoritmos chegaram ao mesmo resultado final e se mostraram apropriados para uso dependendo da dificuldade. Mesmo que contraditório, o algoritmo Guloso se mostrou superior que o A* em alguns momentos, mas isso é explicado de acordo com as diferentes heurísticas usadas por eles.