Uma solução ótima para o jogo Sokoban

Leonardo D. L. Gargioni

I. INTRODUÇÃO

Sokoban é um jogo de quebra-cabeça de computador no qual o jogador (ou agente) só pode empurrar uma caixa por vez por um labirinto para colocá-las em posições destino. Este artigo descreve a implementação de uma Inteligência Artificial que que tem por objetivo resolver uma fase específica deste jogo, cujo estado inicial é visualizado na Figura 1, através de um algoritmo de busca clássica e de forma ótima, ou seja, com o menor número possível de movimentos.

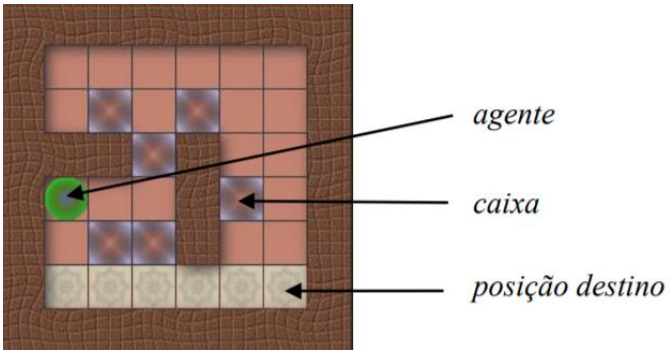


Figura 1 – O Jogo Sokoban

De acordo com a classificação de Russel e Norvig [2], descreve-se o ambiente deste problema como completamente observável, tendo somente um agente, determinístico, sequencial, estático e discreto.

Um novo estado nesse jogo é observado a cada movimentação do agente, tendo ele movido ou não uma caixa. Observa-se que na faze do jogo escolhida existem 6 linhas x 6 colunas e 5 posições em que existem paredes, ou seja, existem 31 posições que podem ser ocupadas por uma caixa ou pelo agente. Como há 6 caixas e 1 agente que podem ocupar uma posição por vez, conclui-se que existem $\frac{31!}{7!(31-7)!} = \frac{31.30.29.28.27.26.25.24!}{7!24!} = 2.629.575$ possíveis estados.

Adicionalmente, um mesmo estado pode ser alcançado de diferentes maneiras, característica de uma busca em Grafo de Estados. Dentre as principais estratégias de busca clássica que fazem busca em Grafo há somente três algoritmos que permitem encontrar uma solução ótima: Largura, Custo Uniforme e A*. Os dois primeiros são buscas cegas (usam somente a informação disponível na formulação do problema) sendo que neste caso o primeiro seria mais apropriado pois contabiliza-se um custo único para cada ação do agente, enquanto o algoritmo de Custo Uniforme é apropriado para problemas em que o custo das ações varia. Já o algoritmo A* é mais eficiente, pois além de levar em conta o custo para chegar a um determinado nó da arvore de busca, ele também é guiado por uma heurística que o conduz à seleção para expansão dos nós com maior probabilidade de estar mais próximo da solução, porém necessita de uma heurística, que nesse caso é viável.

Tendo em vista o tamanho do espaço de estados e que as possíveis soluções se encontram profundamente na arvore de busca, foi decidido implementar o algoritmo A*.

Adicionalmente, foi também implementada a estratégia Gulosa a fim de comparação de performance. Esta estratégia, diferentemente da A* leva em consideração somente a heurística para guiar a busca, por isso não é ótima, porém gera menos nós para alcançar uma solução.

II. ABORDAGEM

Descreve-se nesta sessão alguns dos principais componentes utilizados na implementação.

Para diminuir o espaço de busca, foram implementadas algumas verificações que checam por alguns possíveis deadlocks (estados a partir dos quais não é possível alcançar uma solução). Alguns exemplos de deadlock podem ser visualizados na Figura 2. As explanações sobre as verificações por deadlock implementadas no código serão feitas junto com o detalhamento dos componentes que as implementaram.

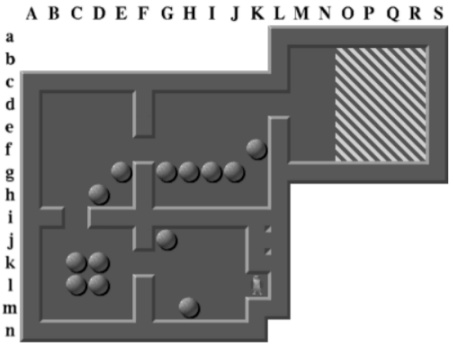


Figura 2 – Deadlocks (adaptado de [1])

A fim de aumentar a performance do programa, foram também implementados dicionários, que por serem

indexados garantem uma velocidade de pesquisa maior principalmente quando a quantidade de itens é alta.

Os estados foram implementados em uma forma fatorada [2] tendo o agente guardado em duas variáveis que representam a linha e coluna em que ele se encontra. Já uma matriz chamada de *"map"*, contém a posição das caixas, espaços em branco, paredes, objetivos e objetivos preenchidos com uma caixa. Adicionalmente foram criados outros objetos usados na mesma matriz para representar um deadlock estático, no caso os espaços próximos à uma parede que não tem uma das posições destino, ou que esta posição já está preenchida.

Durante a exploração de um determinado estado, as ações possíveis são verificadas através do método *"possibleActionsWithoutCollaterals"* que recebe o estado em questão e então primeiramente verifica se existem deadlocks estáticos naquele estado. Após esta verificação ele checará se existe uma parede em alguma das direções, e caso contrário, ele ainda verifica se há uma caixa, e caso positivo, se a caixa pode ser movimentada na direção desejada (se não há outra caixa, parede ou deadlock que impeça o movimento). Por fim este método retorna uma lista que indica quais movimentos são possíveis a partir da posição atual.

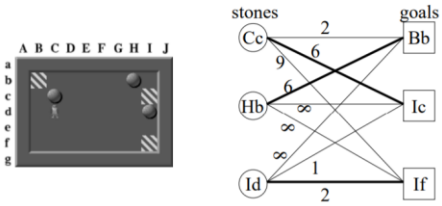
Para cada uma das ações possíveis a função sucessora *"suc"* é chamada tendo como parâmetros o estado atual e ação que o agente está tomando (a direção que ele está indo). Então ela cria um novo estado a partir de uma cópia do estado atual e modifica este novo estado com a nova posição do agente e da caixa caso aplicável. Então ela chama um método do novo estado que irá atualizar uma *string* única que o identifica nos dicionários usados para listar os estados já explorados e de estados que estão na fronteira. Por fim, a função retorna o novo estado para o agente.

O custo para cada ação é de 1 em todos casos, tendo movido ou não uma caixa. Este custo é somado ao *"gn"* do estado pai e salvo no estado filho como seu *gn*.

Como teste de objetivo, é chamado um método do estado em questão que verifica se existe alguma caixa que não esteja em um objetivo. Caso existam caixas fora do objetivo, seu retorno é falso, e caso todas as caixas estejam preenchendo objetivos o retorno é verdadeiro indicando que um caminho foi encontrado para a solução do problema.

Foram implementadas duas heurísticas, a primeira delas (*"hn1"*) é usada pelo algoritmo A* para encontrar o caminho ótimo, por isso ela é admissível e consistente. Esta heurística essencialmente soma as distancias Manhattan das caixas que ainda não estão em uma posição destino para a posição destino que está mais próxima, mesmo que esta posição já esteja ocupada ou que seja a mais próxima de duas ou mais caixas. Foi também implementada uma verificação de deadlock formada por quatro objetos (paredes ou caixas) que estejam juntos formando um quadrado como visualizado no canto inferior esquerdo da Figura 2. Neste caso a heurística retornará o valor máximo, fazendo com que seja evitada sua expansão por ficar no final da fila.

Já a segunda heurística *"hn2"* é uma implementação adaptada da heurística "*Minmatching*" de Junghanns [1], como visualizada na Figura 3. Esta heurística busca primeiramente a distância Manhattan entre cada uma das caixas e as posições destino. Depois ela descobre qual a melhor combinação (em negrito na tabela da Figura 3) e finalmente soma as distâncias. Foram também utilizadas nesta heurística a mesma verificação de deadlock usada na *hn1*, uma verificação adicional de deadlock para caixas que se encontram próximas a uma parede e não podem se mover para uma posição destino que não esteja próxima a mesma parede (identificado como infinito na Figura 3), um adicional de movimentos que são necessários para o agente chegar na caixa caso ele esteja em uma posição que não favorece, e um adicional de movimentos caso haja um obstáculo entre a caixa e a posição destino, e finalmente um custo adicional para o agente chegar até a caixa mais próxima.



stones on	goals on *Bb*	*Ic*	*If*
Cc	2	**6**	9
Hb	**6**	∞	∞
Id	∞	1	**2**

Figura 3 – heurística Minmatching (adaptado de [1])

Como se pode notar, a segunda heurística é muito mais refinada que a primeira e por isso pode guiar a busca com uma maior facilidade por ser mais granular, entretanto, apesar de ter sido tomado o cuidado para que ela seja admissível (ou otimista), ela não é consistente, pois em alguns momentos o nó sucessor tem f(n) menor do que o de seu pai, implicando assim à não obtenção de uma solução ótima para o problema.

III. METODOLOGIA

Serão analisadas três execuções: Uma com o algoritmo A* & heurística *hn1* outra com o algoritmo Guloso usando a mesma heurística e finalmente uma com o algoritmo Guloso e com a heurística *hn2*.

Os seguintes indicadores serão analisados em cada uma das estratégias:

- O Custo - Quantos passos para se atingir o objetivo. Para o algoritmo A* usando a heurística *hn1* é esperado o menor número de passos, por ser ótimo como já analisado anteriormente.
- O tamanho da Fronteira em sua última execução

- O número de nós explorados
- A Quantidade de nós na arvore
- A Quantidade de nós descartados já na fronteira
- A Quantidade de nós descartados por terem sido já explorados
- O total de nós gerados

Adicionalmente será comparada a quantidade de nós gerados para cada um dos algoritmos dependendo do número de caixas na mesma faze do jogo.

IV. RESULTADO

Na execução com todas as 6 caixas, os valores obtidos foram:

	A* & hn1	**Gulosa & hn1**	**Gulosa & hn2**
Custo	124	149	197
Tamanho da Fronteira em sua última execução	40.821	10.205	4.582
Número de nós explorados	767.871	261.225	108.759
Quantidade de nós na arvore	808.691	271.429	113.340
Quantidade de nós descartados já na fronteira	232.920	68.101	28.533
Quantidade de nós descartados por terem sido já explorados	940.872	334.593	137.656
Total de nós gerados	1.982.483	674.123	279.529

Observa-se na tabela que mesmo com a implementação de detecção de deadlocks para evitar a geração de nós ainda o número de nós gerados é próximo de dois milhões, implicando em um alto tempo de execução. Em termos de espaço, observa-se a quantidade de nós na arvore, que também traz preocupações já que ele guarda todos os nós em memória.

O algoritmo guloso se mostra claramente mais eficiente em termos de tempo e espaço como se observa pela quantidade de nós na arvore de busca e pelo número de nós gerados, entretanto como sabido, é sacrificando a otimalidade.

Como pode-se observar na Figura 4, o algoritmo A* é exponencial no número de nós gerados dependendo do número de caixas que são colocadas no jogo, pois quanto mais caixas, maior o número de possíveis estados.

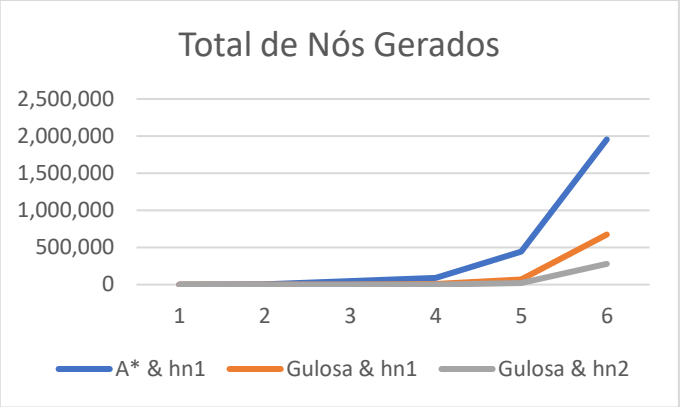


Figura 4 – Total de nós gerados por algorítmo

Solução obtida com A & hn1*

Abaixo a solução ótima (movimentos do agente) obtidos pelo algoritmo A* e heurística *hn1*:

S > S > L > L > L > L > N > L > N > N > O > S > N > N > N > O > O > S > L > N > L > S > N > O > O > O > O > S > L > L > L > N > L > L > S > S > S > S > S > O > O > O > O > O > N > N > L > L > N > S > O > O > S > S > L > L > L > L > L > N > N > O > S > L > S > O > O > O > L > L > N > N > L > N > N > N > O > O > S > L > S > S > S > L > S > O > N > N > N > N > N > O > O > O > S > L > S > S > S > O > N > O > S > N > L > L > N > N > L > N > L > S > S > S > S > N > N > N > N > L > S > S > S > S > FIM

V. CONCLUSÃO

Como visto, a implementação de um algoritmo A* e utilizando-se de uma heurística admissível e consistente foi possível resolver de forma ótima a faze específica do jogo apresentada como problema. Entretanto, apesar de possível a utilização da mesma estratégia em outras fazes, pode-se tornar inviável temporalmente e espacialmente dependendo no número de possíveis estados. Já a estratégia gulosa provou ser mais rápida e útil quando otimalidade não é um requisito.

Como melhorias em trabalhos futuros, deve-se considerar a implementação de artifícios explorados em outros trabalhos acadêmicos, como o uso de macros [1] e o aperfeiçoamento de detecção de deadlocks, para a resolução de fazes do jogo com número ainda maior de possíveis estados.

VI. REFERÊNCIAS

[1] Junghanns, A., Pushing the Limits: New Developments in Single-Agent Search, PhD Thesis, University of Alberta, 1999.

[2] Russell, S. and Norvig, P., Artificial Intelligence: A Modern Approach (3rd Edition), volume 3. Prentice Hall, 2009.