Sokoban – Inteligência Artificial

Osmary Camila Bortoncello Glober *Depto. Acadêmico de Informática Universidade Tecnológica Federal do Paraná - UTFPR*
Curitiba, Brazil
marycamilainfo@gmail.com

I. Introdução

Sokoban se tornou conhecido como um jogo de transporte e movimentação em um armazém onde o principal objetivo do jogo é o jogador levar os engradados (cubos ou caixas) para determinadas posições. Este jogo possui diversas configurações em que o tamanho do armazém, quantidade de caixas e disposições das paredes podem ser alteradas. Para o estudo da matéria de inteligência artificial foi utilizado o armazém com o tamanho de 7x7 e com a disposição das paredes conforme a figura 1.1.

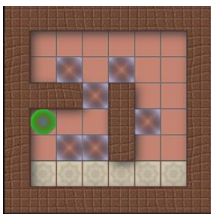

Figura 1.1 – Sokoban utilizado para implementação de algoritmo de busca.

Para a implementação do algoritmo abstraímos os elementos do jogo (armazém, caixas e agentes) em uma matriz 6x6. As caixas e as paredes foram diferenciados com números. Nesta matriz o agente pode se movimentar e movimentar as caixas, no entanto há algumas regras a serem seguidas:

- O agente inicialmente começa na posição (3,0) da matriz ilustrada com um círculo verde, como mostra a figura 1.2.
- Tanto o agente quanto as caixas podem ser movimentadas para cima, para baixo, para a direita ou para a esquerda.
- É permitida somente a movimentação de uma caixa por vez e não é permitida a sobreposição de caixas, como por exemplo pular uma caixa ou mover duas caixas com um único movimento.
- O jogador não pode ocupar o mesmo espaço de uma caixa em uma jogada
- Não é permitido ao jogador andar ou ultrapassar nos muros
- O mesmo se aplica as caixas, ou seja, as caixas não podem ultrapassar os muros ou paredes.

O jogo só é finalizado quando o agente coloca todas as caixas em suas posições predeterminadas que neste caso é a linha 5 da matriz.

O ambiente proporcionado pelo Sokoban é observável pois temos como ver o armazém a cada nova movimentação do agente e como mencionado anteriormente o Sokoban possui somente um personagem proativo e que interage com o armazém tentando alcançar o estado objetivo e com isto o ambiente se torna monoagente. O ambiente também é determinístico pois o estado é completamente definido a cada ação do agente. O ambiente do Sokoban também é sequencial, ou seja, a decisão que o agente toma afeta as decisões futuras, como por exemplo a situação em que o estado de deadlock acontece, conforme a figura 1.2. Devido ao ambiente não se alterar enquanto o agente delibera podemos descrever o Sokoban como um ambiente e estático e também conforme mencionado nas regras do jogo o número de ações que um agente pode realizar por movimentação é discreto ou seja, o ambiente possui um número finito de percepções.

Para esta implementação a busca escolhida foi a busca informada. Ela foi utilizada pois diferentemente das buscas não informadas, como a busca em largura ou busca em profundidade, a busca informada consegue armazenar informações sobre o quão importante um nó pode ser para encontrar a solução através da função de avaliação ou função heurística. Isto não ocorre na busca não informada pois esta calcula somente o custo que neste caso é o número de passos necessários para alcançar um determinado estado. Quanto a estratégia de busca informada, a escolhida foi o algoritmo A*. Este foi escolhido pois consegue avaliar os nós combinando g(n), o custo para alcançar cada nó com h(n), o custo estimado para chegar até o estado objetivo. Portanto o custo estimado total para de um nó n até o estado objetivo é de: $f(n) = g(n) + h(n)$ sendo h(n) uma função heurística admissível, ou seja, sendo otimista relaxando as restrições do problema (ela nunca deve superestimar o custo real) e consistente (respeitar o princípio da desigualdade triangular para garantir que f(n) seja não-decrescente). Utilizando a estratégia A* também assumimos que a qualidade da heurística impactará diretamente na solução do problema e na performance do algoritmo de busca, ou seja, quanto mais informada a heurística estiver menos expansões de estados teremos.

Quanto ao espaço de estados do Sokoban podemos afirmar que ele possui um espaço de estados muito amplo pois temos vários estados de deadlocks onde não atingimos o estado objetivo movendo a caixa para a linha 5 da matriz e devido as restrições do jogo não podemos mais levar a caixa até o estado objetivo, conforme ilustra a figure 1.2. Quanto a complexidade podemos descreve-lo como um problema de decisão PSPACE-Completo, devido ao seu amplo espaço de estados de deadlock, solução longa e grande fator de ramificação.

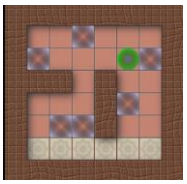

Figura 1.2 – Sokoban com as caixas nas posições 1,0 e 0,2 em estado de Deadlock.

Este estudo tem como objetivo demonstrar como a implementação do algoritmo de busca informada A* consegue fazer com que o agente encontre a solução ótima e também compara este tipo de estratégia com uma estratégia de busca custo uniforme.

II. Modelagem do problema

Permanecemos com todas as classes do algoritmo do labirinto e somente modificamos alguns métodos., Com algumas modificações no algoritmo conseguimos, conforme

mencionado na introdução , abstrair o jogo Sokoban para uma matriz de duas dimensões 6x6 onde as paredes do lado de fora do labirinto foram assumidas como índices que excedem o tamanho da matriz e com isso conseguimos validar nas iterações utilizadas se o agente ou a caixa está em um movimento permitido pelo jogo

A. Iniciando o Jogo

Para iniciar o jogo temos uma classe principal main para inicializar o armazém criando uma matriz 6x6 e colocando as paredes internas, verticais e horizontais, atribuindo o valor 1 para ambas as direções (mesmo código do labirinto). Logo depois também chamamos o método para inserir todas as caixas em suas respectivas posições na matriz atribuindo o valor 2 as posições da matriz indicadas. Ambos são utilizados para locomoção do agente futuramente. Ainda na classe main chamo um método da classe model e colocamos o agente em sua posição inicial 3,0 na matriz. No entanto para iniciar o jogo precisamos também recriar o mesmo estado inicial na classe Problem e carregar todas as características do jogo para a classe Agente, utilizada para deliberação, e com isto teremos a classe Agente consultando a classe Problem quando necessário. Na classe Problem defino exatamente as mesmas condições do estado inicial do jogo, porém em uma nova matriz 6x6 diferente da main, pois não estarei mexendo o agente ainda, apenas deliberando. Para pegar a posição inicial do meu agente na classe Agent há um método sensor em que não preciso dizer aonde o agente está, pois, esta pega o estado inicial da classe model.

B. Movimentação do agente e das caixas

Para controlar as posições válidas em que o agente pode se movimentar enquanto o agente está deliberando foi utilizado um array de inteiros com 4 posições no algoritmo. Cada índice deste array corresponde a direção que o agente pode se mover. Estas definições estão detalhadas na classe cardinal onde definimos a seguinte regra: N =0, L=1, S=2, O=3 e cada valor do array pode assumir o valor 1 ou -1, ou seja, cada posição recebe 1 quando o movimento é possível e -1 quando devido as regras do jogo o movimento não é possível.

Para mover o agente efetivamente no jogo e validar a movimentação do agente seguindo as regras do jogo, há um método checando as mesmas coisas na classe model onde a checagem de movimento e feita, porém em ambas as classes não só o agente é checado como a caixa também. São validadas todas as regras do jogo. Na classe model a validação e feita recebendo a direção em forma de inteiro e incrementando linha e coluna com a posição desejada. Abaixo há um pseudo código descrevendo a parte em que o movimento da caixa e checado na classe model:

```
funcaoMoveAgente(direcao)
    linha = linha onde o agente está
    coluna = coluna onde o agente está

    se direcao é igual a norte
        linha = linha anterior
    se direcao é igual a leste
        coluna = proxima coluna
    se direcao é igual a sul
        linha = proxima linha
    se direcao é igual a oeste
        coluna = coluna anterior

    se col ou linha for maior que a matriz do jogo
        entao
            retorne movimento nao e possivel
    se posicao na matriz do jogo for parede
        entao
            retorne movimento nao e possivel
    se posicao na matriz for caixa
        se funcaoMoveCaixa(direcao, linha, coluna) nao for possivel
            entao
                retorne movimento nao e possivel

    retorne move o agente()
```

```
funcaoMoveCaixa(direcao, linha, coluna)
    linhaCaixa = linha
    colunaCaixa = coluna
    se direcao é igual a norte
        linhaCaixa = linha anterior
    se direcao é igual a leste
        colunaCaixa = proxima coluna
    se direcao é igual a sul
        linhaCaixa = proxima linha
    se direcao é igual a oeste
        colunaCaixa = coluna anterior

    se colunaCaixa ou linhaCaixa for maior que a matriz do jogo
        entao retorne movimento nao e possivel
    se posicao na matriz do jogo for parede
        entao retorne movimento nao e possivel
    se posicao na matriz do jogo for caixa
        entao retorne movimento nao e possivel

    move a caixa
    retorne movimento possivel
```

Abaixo há um pseudo código da mesma checagem, porém com o retorno diferente da classe Problem utilizado para deliberação:

```
acoesPossiveis(estado)
    vetor acoes = [Norte válido, Leste válido, Sul válido, Oeste válido]

    linha = linha do agente na deliberacao
    coluna = coluna do agente na deliberacao

    se linha esta nao esta nos limites da matriz
        vetor acoes = [Norte inválido, Sul invalido]
    se soluna esta nao esta nos limites da matriz
        vetor acoes = [Leste invalido, Oeste invalido]

    se linha anterior da posicao do agente e coluna atual do agente igual parede
        vetor acoes = [Norte inválido]
    se linha superior da posicao do agente e coluna atual do agente igual parede
        vetor acoes = [Sul inválido]
    se linha atual do agente e coluna superior do agente igual parede
        vetor acoes = [Leste inválido]
    se linha atual do agente e coluna superior do agente igual parede
        vetor acoes = [Oeste inválido]

    se linha anterior da posicao do agente e coluna atual do agente igual caixa e caixa nao pode mover
        vetor acoes = [Norte inválido]
    se linha superior da posicao do agente e coluna atual do agente igual caixa e caixa nao pode mover
        vetor acoes = [Sul inválido]
    se linha atual do agente e coluna superior do agente igual caixa e caixa nao pode mover
        vetor acoes = [Leste inválido]
    se linha atual do agente e coluna superior do agente igual caixa e caixa nao pode mover
        vetor acoes = [Oeste inválido]

    retorne vetor todas as posições válidas e inválidas
```

C. Função sucessora

Na função sucessora dentro da classe Problem é validado o movimento de um estado do agente e se este movimento pode ser executado sem quebrar as regras. O pseudo código utilizado na movimentação do agente na model e da validação do movimento da caixa pode ser utilizado como referência na função sucessora. A única diferença é que ao invés de mover o agente na matriz principal do jogo o método sucessor retorna se o estado sucessor do agente baseado em uma direção é válido ou não, se não for válido retorna o mesmo estado, ou seja, não muda a posição do agente na deliberação.

A separação das funções do movimento do agente e caixa entre a classe Problem e a classe Agent se dá pois na classe Model temos somente o modelo utilizado e a matriz real do jogo e como mencionado nas classes Agent e Problem temos as mesma regras pois deliberamos e precisamos aplicar checar os próximos movimentos.

D. Custo das ações

Na implementação feita foi assumido que g(n) será 1 pois estamos ignorando o movimento diagonal e contabilizando somente as colunas e linhas. Conforme mencionado na introdução, para avaliação final do custo a estratégia A* define como função: f(n) = g(n) + h(n) e a cada novo estado analisado retornado da função sucessora, ou seja a cada movimento válido que meu agente poderá fazer, calculo o custo de cada direção de cada movimento e com isto posso saber qual seria a melhor direção para o agente seguir e atingir o estado objetivo. Abaixo há um pseudo código.

```
direcoes possiceis = direcoes possiveis pelo agente no estado atual

para cada direção em direcoes possiveis
    estado sucessor = calcula novo estado(direção)
    novo no.estado do agente = estado sucessor

    novo no.gn = todo o custo acumulado das outras movimentação + custo da acao atual
    novo no.hn = calcula heuristica para o novo estado

    novo no.direção = direção
```

E. Teste do estado objetivo

Para o teste do estado objetivo há dois métodos na classe Problem e ele é testado a cada iteração com um novo nó da fronteira. Primeiramente o estado objetivo individual de cada caixa e testado, se o estado do agente conseguiu mover mais uma caixa para o estado objetivo.

```
checaEstadoObjetivoCaixa()
    contaCaixas = 0
    armazem = matriz para deliberacao

    para cada coluna dentro de matriz para deliberacao
        se armazem[linha objetivo][coluna] igual a caixa
            contaCaixas = contaCaixas + 1

    se contaCaixas maior que numero de caixas
        entao
            numero de caixas = contaCaixas
            retorna mais uma caixa atingiu o objetivo
    senao
        entao
            retorna o numero de caixas no objetivo e o mesmo
```

Também há um método para checar se o estado objetivo do jogo foi alcançado, ou seja, quando todas as caixas estiverem na linha 5 da matriz do jogo.

```
checaEstadoObjetivoFinal()
    se numero de caixas == 6
        entao retorna atingiu o objetivo
    senao
        entao retorna nao atingiu o objetivo
```

A cada nó faço a checagem do objetivo e se ele for atingido paro a execução e mostro o resultado da deliberação do agente, que pode ter encontrado uma solução.

```
enquanto haver proximo no na fronteira
    no = primeiro no da fronteira para testar proximos estados

    se checa estado objetivo caixa
        se checa estado objetivo final
            retorna o no atual
```

F. Heurística utilizada

Na implementação utilizada a heurística acumulou a distância em linhas de cada caixa fora do estado objetivo que é a linha 5. Se há uma caixa na linha objetivo e a posição desta coincide com a mesma coluna da caixa que está sendo checada a distância de manhattan e medida com a diferença de linhas da caixa sendo checada até a linha e também com a distância de colunas até a próxima coluna disponível na linha objetivo. Na heurística é também checado se o agente está próximo da caixa montando uma validação de 3x3 ao redor de cada caixa para saber em qual posição o agente se encontra e se ele se encontra perto da caixa. Se o agente não se encontra perto da caixa, ou seja, se ele não se encontra dentro desta matriz então atribuo o número 8, uma média estimada de movimentos para o agente mover a caixa. Se o agente se encontra ao redor da caixa, dentro da matriz 3x3 atribuo uma média menor de movimentos necessários para movimentação da caixa. Se o agente se encontra nas diagonais da matriz 3x3 ao redor da caixa atribuo 2 como média de movimentos mínimos para o agente movimentar a caixa. Se o agente estiver em cima, dos lados ou abaixo da caixa da matriz 3x3 atribuo uma média de 1 movimento para movimentar a caixa. Agora se o agente estiver exatamente na posição da caixa então o número de movimentos é zerado.

Caso a matriz 3x3 ao redor da caixa não seja possível atribuo 16 ao número médio de movimentos pois a tendência de estar em estado de deadlock aumenta. Após todas estas validações o retorno da função e a soma das distâncias das caixas até os objetivos mais o número de movimentos necessários para mover a caixa baseado na posição do agente e da caixa. Abaixo há o pseudo código da heurística utilizada:

```
calcula heuristica(estado do agente)
    heuristica final = 0
    distancia linha das caixas = 0
    distancia coluna das caixas = 0

    para cada linha em matriz delibração
        para cada coluna em matriz delibração
            se matriz delibaracao[linha][coluna] for igual a caixa
                media movimentos = 8

                se agente esta dentro da da matriz ao redor da caixa
                    media movimentos = 4
                    se matriz ao redor da caixa não excede os limites da matriz para deliberação
                        se agente estiver nas diagonais da matriz ao redor da caixa
                            media movimentos = 2
                        se agente estiver nas direções N, S, L, O da matriz ao redor da caixa
                            media movimentos = 1
                        se agente estiver exatamente na posição da minha caixa
                            media movimentos = 0
                    senao
                        se e um estado de deadlock
                            media movimentos = 16

                distancia linha = linha da caixa - linha objetivo
                se ha alguma caixa na linha e soluna da caixa sendo testada
                    para cada coluna em matriz delibração
                        se matriz delibração[linha objetivo][coluna] for diferente de caixa
                            distancia coluna obj = coluna da caixa - coluna objetivo
                            se distancia coluna obj < distancia coluna
                                distancia coluna = distancia coluna + distancia coluna obj
                senao
                    distancia coluna das caixas = 0

                heuristica final = heuristica final + distancia linha + distancia coluna das caixas + media movimentos
    retorna heuristica final
```

III. RESULTADOS

Como teste inicial o primeiro cenário foi somente com uma única Caixa na posição (4,1) e uma matriz 6x6 já com as paredes configuradas. A heurística utilizada foi a distância de manhattan do agente até a caixa juntamente com a soma da distância da linha da caixa até o estado objetivo, ou seja, linha 5. Como um primeiro teste a heurística funcionou perfeitamente e o agente conseguiu encontrar a solução ótima com o 6 nós na árvore e com o custo 2. Passando para a segunda caixa na posição (4,2) o algoritmo não conseguia entender a ação de voltar um nó, ou seja, na hora de adicionar um novo nó na fronteira devido a comparação entre o estado do nó com os estados já explorados o nó para retornar a posição anterior não estava sendo adicionado na árvore.

Para solução deste problema tive que remover o método de comparação de cada estado do novo nó com os estados dos nós já explorados pois no caso do Sokoban, diferentemente do labirinto eu posso ter a opção de voltar para poder mover alguma outra caixa que pode ter ficado para trás.

Após remoção do método e utilizando a mesma heurística o estado objetivo foi alcançado com um custo de 5 e com 19 nós gerados. Porém ainda assim o número de nós gerados foi bastante alto então resolvi ajustar a minha heurística. Tendo como base alguns artigos citados nas referencias deste estudo comecei a utilizar outros parâmetros para o cálculo da heurística. Num primeiro tentei somente calcular a soma da quantidade de caixas fora do estado objetivo junto com a distancia de manhattan do meu agente ate as caixas. O método se provou muito raso e com isso a arvore cresceu pois o agente começou a explorar outras posições. Baseado nos estudos das referências, a distância do agente até as caixas pode variar e aumentar e com isto a heurística perde uma de suas características, ela deixa de ser consistente. Diante disso no segundo teste foi utilizando somente a distancia de manhattan das caixas ate o estado objetivo e a distância do agente até as caixas foi ignorada e com isto o meu agente consegui atingir o estado objetivo com um custo de 5 e com 14 nos gerados.

Passando a testar o algoritmo com 3 caixas sendo a terceira caixa na posição (3,4) me deparo com um problema, a eficiência do algoritmo decai muito, sendo que o tempo de execução saltou exponencialmente. Com isto resolvo ajustar a heurística fornecendo mais parâmetros para calculá-la. Começo incrementando a distância de manhattan das caixas até a linha objetivo, porém neste caso específico coloco também a distância da coluna. Utilizando esta heurística a precisão dos objetivos é aumentada, porém testando eles

individualmente e aplicando um teste final percebo que a diferença não foi tanta em comparação com calcular a distancia das linhas das caixas com a distâncias da linha objetivo. Analisando algumas estratégias para cálculo de heurística para o sokoban como *Minmatchin, Rolling Stone*, dentre outras comecei a adicionar uma quantidade movimentos necessários para fazer com que a caixa se mova. Conforme mencionado anteriormente faço uma comparação ao redor da caixa em uma matriz 3x3 e dependendo da posição do meu agente é atribuído um determinado número de movimentos para que o agente consigo mover a caixa. Como tenho somente 3 caixas e que para movê-las até a posição objetivo meu agente precisará movê-las do norte para o Sul inicialmente atribuo um media de movimentos maior (4 movimentos) para quando o agente está posicionado embaixo da caixa e menor (1 movimento) quando o meu agente está na posições superior da caixa. Quando meu agente está ao lado da caixa atribuo 2 como média de movimento. Somando esta média de movimentos com a distância das linhas das caixas até a linha objetivo consigo a solução ótima a um custo de 13 com 49 nós na árvore e com 34 nós descartados.

Inserindo a quarta caixa percebo que o peso anteriormente dado a posição do agente precisaria ser repensado e com isto defino que quando o agente esta nas diagonais a media de movimentos e 2, se ele esta em cima, embaixo ou do lado da caixa média é 1 e se o meu agente chegou na posição na caixa minha média é 0. Com isto o algoritmo chegou ao estado objetivo com um custo de 22 com 61 nos na árvore e 33 descartados.

A solução encontrada foi:
L > S > N > L > S > N > N > N > L > L > S > S > S > N > N > N > N > L > S > S > S > S > FIM

Quando adicionei a quinta caixa na posição (2,2) o algoritmo não conseguiu chegar a uma solução pois devido aos imensos espaços de estados com deadlocks a performance ficou muito ruim e não consegui adicionar mais caixas.

A. Comparação com o algoritmo custo uniforme

Podemos ver melhor nas tabelas abaixo a comparação tempo e espaço em nós da árvore de busca entre a estratégia de busca A* com a estratégia de busca custo uniforme.

1 Caixa	Custo Uniforme	A*
Nos na Árvore	6	6
Descartados	0	0
Total	6	6

2 Caixas	Custo Uniforme	A*
Nos na Árvore	10	11
Descartados	2	1
Total	12	12

3 Caixas	Custo Uniforme	A*
Nos na Árvore	67	54
Descartados	75	42
Total	142	96

4 Caixas	Custo Uniforme	A*
Nos na Árvore	100	61
Descartados	124	33
Total	224	94

IV. Conclusão

Como mencionei anteriormente devido à alta complexidade do espaço de estados os testes com a quinta e com a sexta caixa não foram realizados, então um futuro passo desta implementação a criação de cenários mais detalhados nas heurísticas para atribuir a média de movimento correta em cenários de deadlock se prova muito relevante, calculando assim uma heurística mais consistente. Outro fator a ser considerado seria na heurística analisar ao redor das caixas quantos obstáculos tem e atribuir um peso maior para a média de movimentos necessários para mover a caixa.

Referências.

Jacqueline Rodrigues, “MAC0499 - Trabalho de Formatura Supervisionado” in https://bcc.ime.usp.br/tccs/2008/rec/jacqueline/jacquelineMonografia.pdf.

Dr. André Grahl Pereira, “Learning Deadlocks in Sokoban” https://www.lume.ufrgs.br/bitstream/handle/10183/190174/001088740.pdf?sequence=1.

Timo Virkkala, “Solving Sokoban” http://weetu.net/Timo-Virkkala-Solving-Sokoban-Masters-Thesis.pdf.

Dorit Dor 1, Uri Zwick, “SOKOBAN and other motion planning problems”.

Peter Norvig e Stuart Russell, “Inteligência Artificial”, 2020.