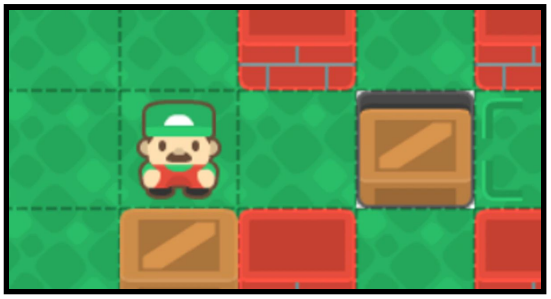
Desenvolvimento de um Agente autónomo para o jogo

Sokoban

Alexandre Rodrigues nmec: 92951

Orlando Macedo nmec: 94521

2020/2021



Sr.(s) Professores: Diogo Gomes e Luís Seabra Lopes

Índice

Introdução

Proposto a trabalho de grupo prático na cadeira de Inteligência Artificial (IA), este tem como objetivo desenvolver um agente capaz de jogar de forma inteligente o jogo do Sokoban.

O objetivo do agente do Sokoban é arrumar caixas em locais pré-definidos no mapa, sinalizados por pequenos diamantes. Para movimentar as caixas o agente terá que as empurrar. Não existem adversários nem fatores aleatórios.

Todos os níveis têm um tempo limite para serem concluídos.

A implementação de uma solução para resolver parte dos problemas do Sokoban passou pelo uso de STRIPS.

Descrição do Algoritmo

STRIPS

Script *strips.py*

Aconselhado pelos professores o uso de STRIPS como fonte para a resolução dos problemas do Sokoban, o nosso objetivo foi sempre seguir essa metodologia. O principal desafio ao utilizarmos esta metodologia foi particularmente a escolha do(s) predicados que iríamos usar e não a escolha dos operadores visto que os únicos possíveis são os movimentos/direções (A,S,W,D).

O nosso projeto foi sofrendo alterações consideráveis ao longo do seu desenvolvimento. Inicialmente o objetivo foi mover o *keeper* de forma aleatória seguindo-se de imediato a escolha dos predicados. Foram utilizados predicados tais como o uso de todas as *tiles* disponíveis sendo que cada operador seria uma lista de listas com todos os movimentos válidos utilizando os predicados. De imediato reparamos que o tempo total para a resolução do nível 1 ultrapassava os 30 segundos.

De seguida reduzimos os predicados para ***Box*** e ***Availables***. Os Availables eram uma lista com todas as coordenadas disponíveis no mapa. Com o uso de uma pesquisa A* que verificava todas as permutações possíveis até que o *goal* (coordenadas das boxes igua10l às coordenadas dos diamantes) fosse atingido. O movimento do *keeper* é descrito mais à frente.

Por fim, apenas foi utilizado o predicado ***Box*** pois a informação relativa às posições disponíveis para o *keeper* se movimentar mantêm-se constantes ao longo da evolução dos estados, excetuando as posições para onde as caixas se movem. Portanto as coordenadas disponíveis passaram a ser guardadas numa lista inicializada no construtor do strips.

Para o cálculo das *actions* foi utilizada a função *get_subP_goals* que mediante o estado que lhe é passado como parâmetro, verifica quais são as coordenadas para onde o keeper se pode mover (sempre tendo em atenção futuros *deadlocks*). Depois de retornadas todas as coordenadas possíveis, faz-se uma sub-pesquisa com o intuito de encontrar o caminho ótimo entre o local onde o *keeper* se encontra no estado atual e para onde o keeper terá de ir para empurrar uma determinada caixa no estado seguinte.

Keeper

Script *moveKeeper.py*

O keeper move-se utilizando para tal uma árvore de pesquisa distinta da árvore usada para a pesquisa do problema principal.
Foi utilizada uma implementação A* cujo custo é dado pelo deslocamento do agente de uma coordenada para a outra de forma lateral (custo 1) e a heurística como sendo a distância de manhattan.

O objetivo do agente é mover-se para uma posição lateral disponível mais próxima de uma dada caixa sempre em coordenadas livres.

O percurso percorrido pelo Agente é traduzido em *keys* pela função *evolution2keys(evo).*

Pesquisa principal

Script *tree_search.py*

Para a pesquisa relacionada com o movimento de caixas, foi também utilizada uma pesquisa A* em que os nós eram ordenados pela sua heurística.

A heurística utilizada mapeia uma determinada caixa ao objetivo que estiver mais próximo de si, sendo a distância calculada a partir do teorema de pitágoras. Cada objetivo fica associado a apenas uma caixa.
O valor final da heurística é a soma de todas as distâncias de caixa a objetivo.

Para não abrir estados repetidos foi criado o set *visited,* no qual se guarda as hash do estado aberto bem como a posição onde o *keeper* se encontra.
Do mesmo modo, para não repetir estados que se encontrem na fila em espera para serem abertos, foi criado o set *in_queue* que guarda as *hash* de todos os estados que se encontram no momento à espera para serem expandidos.

Estes dois *set's* são usados quando há a formação de um novo estado. Esse estado só formará um novo nó da árvore se não se encontrar em nenhum dos *set's.*

Prevenção de deadlocks

Simple deadlock: A implementação desta detenção de deadlock consiste em verificar se o movimento de uma caixa para uma determinada coordenada lateral é possível. Isto é, o deslocamento de uma caixa apenas poderá ser realizado para uma coordenada livre e não para cima de uma parede.
Para além disso, também permite verificar se a linha/coluna próxima duma determinada parede faz sentido ser adicionada à lista de coordenadas disponíveis (fará se houver um objetivo nessa linha/coluna, caso contrário irá ocorrer num *deadlock*).
Por último retira todos os cantos do mapa (coordenadas onde existe uma parede a bloquear o movimento póstumo da caixa tanto na vertical como na horizontal), exceto os casos em que essa coordenada é um objetivo.

```python
def wall_deadlock(free_coord, walls, coords_of_goals, coords_available):
    # se a coordenada é um goal entao vamos supor que n é um wall deadlock
    if free_coord in coords_of_goals:
        return False
    x, y = free_coord

    # se todas as coordenadas à volta estiverem livres n é deadlock
    if all(c in coords_available for c in [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)]):
        return False

    # verificar para o topo esquerdo
    if check_wall_conditions((x, y - 1), (x - 1, y), walls,
                             lambda : sum(1 for c in coords_of_goals if y >= c[1]),
                             lambda : sum(1 for c in coords_available if y-1 >= c[1]) > 3,
                             lambda : sum(1 for c in coords_of_goals if x >= c[0]),
                             lambda : sum(1 for c in coords_available if x-1 >= c[0]) > 3):
        return True
    # para o topo direito
    if check_wall_conditions((x, y - 1), (x + 1, y), walls,
                             lambda : sum(1 for c in coords_of_goals if y >= c[1]),
                             lambda : sum(1 for c in coords_available if y-1 >= c[1]) > 3,
                             lambda : sum(1 for c in coords_of_goals if x <= c[0]),
                             lambda : sum(1 for c in coords_available if x+1 <= c[0]) > 3):
        return True
    # para o canto inferior esquerdo
    if check_wall_conditions((x, y + 1), (x - 1, y), walls,
                             lambda : sum(1 for c in coords_of_goals if y <= c[1]),
                             lambda : sum(1 for c in coords_available if y+1 <= c[1]) > 3,
                             lambda : sum(1 for c in coords_of_goals if x >= c[0]),
                             lambda : sum(1 for c in coords_available if x-1 >= c[0]) > 3):
        return True
    # para o canto inferior direito
    if check_wall_conditions((x, y + 1), (x + 1, y), walls,
                             lambda : sum(1 for c in coords_of_goals if y <= c[1]),
                             lambda : sum(1 for c in coords_available if y+1 <= c[1]) > 3,
                             lambda : sum(1 for c in coords_of_goals if x <= c[0]),
                             lambda : sum(1 for c in coords_available if x+1 <= c[0]) > 3):
        return True

    return False
```

img1. wall deadlock

Freeze Deadlock

Relativamente ao *freeze deadlock*, o mesmo permite verificar se o movimento de uma caixa vai provocar um *deadlock* ou não, essa verificação é feita recursivamente tirando proveito do facto de se saber que havendo uma caixa y no caminho de x, a caixa x não se poderá mover se também não for possível mover a caixa y.

Em primeira instância, verifica-se se a caixa a tratar está bloqueada verticalmente por caixas ou paredes.

Caso não esteja, não existe deadlock. Caso contrário, teremos de verificar horizontalmente.

Na horizontal fazemos a mesma verificação, portanto se à esquerda ou direita da caixa se encontra uma *wall* ou caixa.

Se a caixa estiver bloqueada horizontalmente, há um *deadlock* e essa coordenada não é adicionada às coordenadas possíveis para onde uma dada caixa se pode movimentar.

Uma ressalva ao facto de no caso de uma caixa estar a ser bloqueada por uma outra caixa, na ocorrência desta situação é necessário verificar se a dada caixa está também ela bloqueada. Para isso, a caixa inicial é adicionada à lista de paredes, passando a coordenada da caixa a comportar-se como se fosse uma parede.

```python
def vertical_check(coord_box, coord_boxes, walls, coords_goals):
    x, y = coord_box
    up = (x, y - 1)
    down = (x, y + 1)
    if up in walls or down in walls:
        return coord_box

    if up in coord_boxes:
        return up
    if down in coord_boxes:
        return down

    return None

def horizontal_check(coord_box, coord_boxes, walls, coords_goals):
    x, y = coord_box
    left = (x - 1, y)
    right = (x + 1, y)

    if left in walls or right in walls:
        return coord_box

    if left in coord_boxes:
        return left
    if right in coord_boxes:
        return right

    return None
```

img2. Verificações verticais e horizontais de movimento da caixa

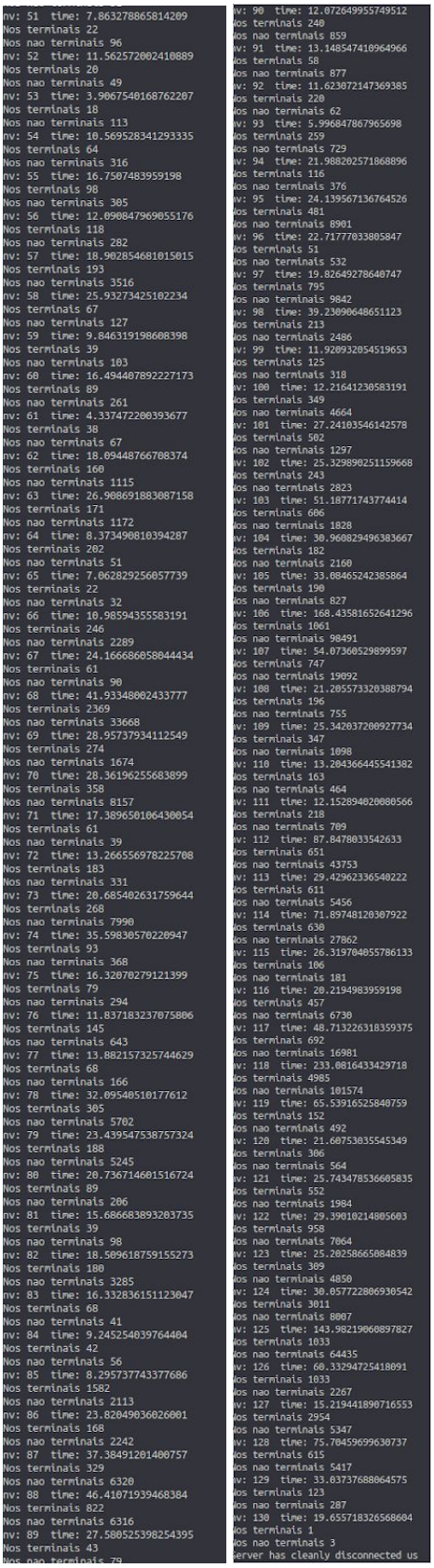
```python
def freeze_deadlock(coord_box, coord_boxes, walls, coords_goals, current_coord_box):
    coord_boxes.remove(current_coord_box)
    coord_boxes.append(coord_box)
    ret = False
    box2check = []
    while True:
        if coord_box in coords_goals:
            break

        vertical = vertical_check(coord_box, coord_boxes, walls, coords_goals)

        if vertical is None:     # no caso de n haver deadlock na vertical
            break

        horizontal = horizontal_check(coord_box, coord_boxes, walls, coords_goals)

        #no caso de haver deadlock na vertical mas na horizontal n
        if horizontal is None:
            break

        # se há uma wall na vertical e tmb na horizontal
        if vertical == coord_box and horizontal == coord_box:
            ret = True
            break

        if vertical != coord_box:
            box2check.append(vertical)

        if horizontal != coord_box:
            box2check.append(horizontal)

        walls.add(coord_box)
        if len(box2check) == 0:
            break

        coord_box = box2check.pop(0)

    return ret
```

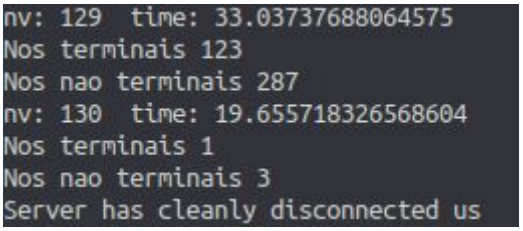
img3. Freeze Deadlock

Corral Deadlock

Basicamente esta verificação de deadlock consiste em verificar se há uma área para o qual o *keeper* não consegue alcançar e consequentemente não consegue puxar a caixa.

Resultados Obtidos


nv: 51 time: 7.8632788658142O9
Nos terminais 22
Nos nao terminais 96
nv: 52 time: 11.562572002410889
Nos terminais 20
Nos nao terminais 49
nv: 53 time: 3.9067540168762207
Nos terminais 18
Nos nao terminais 113
nv: 54 time: 10.569528341293335
Nos terminais 64
Nos nao terminais 316
nv: 55 time: 16.7507483959198
Nos terminais 98
Nos nao terminais 305
nv: 56 time: 12.090847969055176
Nos terminais 118
Nos nao terminais 282
nv: 57 time: 18.902854681015015
Nos terminais 193
Nos nao terminais 3516
nv: 58 time: 25.93273425102234
Nos terminais 67
Nos nao terminais 127
nv: 59 time: 9.846319198608398
Nos terminais 39
Nos nao terminais 103
nv: 60 time: 16.494407892227173
Nos terminais 89
Nos nao terminais 261
nv: 61 time: 4.337472200393677
Nos terminais 38
Nos nao terminais 67
nv: 62 time: 18.09448766708374
Nos terminais 160
Nos nao terminais 1115
nv: 63 time: 26.908691883087158
Nos terminais 171
Nos nao terminais 1172
nv: 64 time: 8.373490810394287
Nos terminais 202
Nos nao terminais 51
nv: 65 time: 7.662829256057739
Nos terminais 22
Nos nao terminais 32
nv: 66 time: 10.98594355583191
Nos terminais 246
Nos nao terminais 2289
nv: 67 time: 24.166686058044434
Nos terminais 61
Nos nao terminais 90
nv: 68 time: 41.93348002433777
Nos terminais 2369
Nos nao terminais 33668
nv: 69 time: 28.95737934112549
Nos terminais 274
Nos nao terminais 1674
nv: 70 time: 28.36196255683899
Nos terminais 358
Nos nao terminais 8157
nv: 71 time: 17.389650106430054
Nos terminais 61
Nos nao terminais 39
nv: 72 time: 13.266556978225708
Nos terminais 183
Nos nao terminais 331
nv: 73 time: 20.685402631759644
Nos terminais 268
Nos nao terminais 7990
nv: 74 time: 35.59830570220947
Nos terminais 93
Nos nao terminais 368
nv: 75 time: 16.32070279121399
Nos terminais 79
Nos nao terminais 294
nv: 76 time: 11.837183237075806
Nos terminais 145
Nos nao terminais 643
nv: 77 time: 13.882157325744629
Nos terminais 68
Nos nao terminais 166
nv: 78 time: 32.09540510177612
Nos terminais 305
Nos nao terminais 5702
nv: 79 time: 23.439547538757324
Nos terminais 188
Nos nao terminais 5245
nv: 80 time: 20.736714601516724
Nos terminais 89
Nos nao terminais 206
nv: 81 time: 15.686683893203735
Nos terminais 39
Nos nao terminais 98
nv: 82 time: 18.509618759155273
Nos terminais 180
Nos nao terminais 3285
nv: 83 time: 16.332836151123047
Nos terminais 68
Nos nao terminais 41
nv: 84 time: 9.245254039764404
Nos terminais 42
Nos nao terminais 56
nv: 85 time: 8.295737743377686
Nos terminais 1582
Nos nao terminais 2113
nv: 86 time: 23.82049036026001
Nos terminais 168
Nos nao terminais 2242
nv: 87 time: 37.38491201400757
Nos terminais 329
Nos nao terminais 6320
nv: 88 time: 46.41071939468384
Nos terminais 822
Nos nao terminais 6316
nv: 89 time: 27.580525398254395
Nos terminais 43
Nos nao terminais 79
nv: 90 time: 12.072649955749512
Nos terminais 240
Nos nao terminais 859
nv: 91 time: 13.148547410964966
Nos terminais 58
Nos nao terminais 877
nv: 92 time: 11.623072147369385
Nos terminais 220
Nos nao terminais 62
nv: 93 time: 5.996847867965698
Nos terminais 259
Nos nao terminais 729
nv: 94 time: 21.988202571868896
Nos terminais 116
Nos nao terminais 376
nv: 95 time: 24.139567136764526
Nos terminais 481
Nos nao terminais 8901
nv: 96 time: 22.71777033805847
Nos terminais 51
Nos nao terminais 532
nv: 97 time: 19.82649278640747
Nos terminais 795
Nos nao terminais 9842
nv: 98 time: 39.23090648651123
Nos terminais 213
Nos nao terminais 2486
nv: 99 time: 11.928932054519653
Nos terminais 125
Nos nao terminais 318
nv: 100 time: 12.21641230583191
Nos terminais 349
Nos nao terminais 4664
nv: 101 time: 27.24103546142578
Nos terminais 502
Nos nao terminais 1297
nv: 102 time: 25.329890251159668
Nos terminais 243
Nos nao terminais 2823
nv: 103 time: 51.18771743774414
Nos terminais 606
Nos nao terminais 1828
nv: 104 time: 30.960829496383667
Nos terminais 182
Nos nao terminais 2160
nv: 105 time: 33.08465242385864
Nos terminais 190
Nos nao terminais 827
nv: 106 time: 168.43581652641296
Nos terminais 1061
Nos nao terminais 98491
nv: 107 time: 54.07360529899597
Nos terminais 747
Nos nao terminais 19092
nv: 108 time: 21.205573320388794
Nos terminais 196
Nos nao terminais 755
nv: 109 time: 25.342037200927734
Nos terminais 347
Nos nao terminais 1098
nv: 110 time: 13.204366445541382
Nos terminais 163
Nos nao terminais 464
nv: 111 time: 12.152894020080566
Nos terminais 218
Nos nao terminais 709
nv: 112 time: 87.8478033542633
Nos terminais 651
Nos nao terminais 43753
nv: 113 time: 29.42962336540222
Nos terminais 611
Nos nao terminais 5456
nv: 114 time: 71.89748120307922
Nos terminais 638
Nos nao terminais 27862
nv: 115 time: 26.319704055786133
Nos terminais 106
Nos nao terminais 181
nv: 116 time: 20.2194983959198
Nos terminais 457
Nos nao terminais 6730
nv: 117 time: 48.713226318359375
Nos terminais 692
Nos nao terminais 16981
nv: 118 time: 233.0816433429718
Nos terminais 4985
Nos nao terminais 101574
nv: 119 time: 65.53916525840759
Nos terminais 152
Nos nao terminais 492
nv: 120 time: 21.60753035545349
Nos terminais 306
Nos nao terminais 564
nv: 121 time: 25.743478536605835
Nos terminais 552
Nos nao terminais 1984
nv: 122 time: 29.3901021480503
Nos terminais 958
Nos nao terminais 7064
nv: 123 time: 25.20258665084839
Nos terminais 309
Nos nao terminais 4850
nv: 124 time: 30.057722806930542
Nos terminais 3011
Nos nao terminais 8007
nv: 125 time: 143.98219060897827
Nos terminais 1033
Nos nao terminais 64435
nv: 126 time: 60.33294725418091
Nos terminais 1033
Nos nao terminais 2267
nv: 127 time: 15.219441890716553
Nos terminais 2954
Nos nao terminais 5347
nv: 128 time: 75.70459699630737
Nos terminais 615
Nos nao terminais 5417
nv: 129 time: 33.03737688064575
Nos terminais 123
Nos nao terminais 287
nv: 130 time: 19.655718326568604
Nos terminais 1
Nos nao terminais 3
Server has cleanly disconnected us



img4. Server disconnected us at level 131

Depois de todos os testes feitos, utilizando diferentes algoritmos para calcular distâncias entre caixas e objetivos, utilizando diferentes heurísticas, diferentes custos, depois de todas essas experiências foi consensual que a melhor solução tinha sido a atual. Usando A* com *set's* a monitorar ocorrências de estados repetidos, usando o teorema de pitágoras para calcular distâncias entre caixas e objetivos, e usando um método de mapeamento de objetivos a caixas para a heurística.
Na reta final deparamo-nos com um cenário em que se teve de escolher entre completar mais níveis e ter mais pushes de caixas ou então completar menos níveis mas com uma solução mais ótima. Optou-se por completar mais níveis ao custo de soluções com mais pushes de caixas.
Por último, o agente criado foi capaz de resolver 131 níveis nos computadores utilizados pelos alunos que efetuaram este projeto. De seguida encontram-se evidências da resolução desses 131 níveis.

Conclusão

A realização do trabalho de grupo prático permitiu que fossem aprofundados os conhecimentos sobre como se deve comportar um **agente inteligente** que é aquele que adota a melhor ação possível diante de uma situação. Ao longo da realização do TPG foram discutidas diversas linhas de pensamento sobre como o agente deveria ou não se comportar.

Por outro lado, também foram melhorados os nossos conhecimentos sobre a pesquisa em árvore, os diferentes prós e contras que o uso de um determinado tipo de árvore de pesquisa implicaria na performance do problema e para além do mais o custo e a heurística da implementação usada (A*) também foi uma verdadeira fonte de troca de ideias.

Foi feita muita pesquisa sobre diferentes tipos de estruturas que ajudariam na performance do problema e foi colocado em prática o uso de funções lambda, recursivas e list comprehensions.

Relativamente a trabalho futuro, gostaríamos de ter implementado uma heurística que usasse o método Húngaro de forma a verificar se o mesmo seria uma mais valia para a resolução dos níveis do sokoban. Também seria interessante ver o resultado duma

pesquisa CBFS em vez de A*, e por último, para o cálculo de distâncias, fica a pluma de não termos implementado o algoritmo de goal pull.

Assim concluímos que o presente trabalho enriqueceu o nosso leque de conhecimentos sobre como implementar um agente inteligente.

Referências

http://sokobano.de/wiki/index.php?title=How_to_detect_deadlocks&fbclid=IwAR0rpY0djteccnAvq2Ln-uig6SgdwsWIFks3SNuR8beImDlnxsQh3NoVPUQ

https://towardsdatascience.com/dont-use-recursion-in-python-any-more-918aad95094c

https://towardsdatascience.com/dynamic-programming-for-data-scientists-bb7154b4298b

http://www.sokobano.de/wiki/index.php?title=Solver

http://www.sokobano.de/wiki/index.php?title=Sokoban_solver_%22scribbles%22_by_Florent_Diedler_about_the_Sokolution_solver

http://www.sokobano.de/wiki/index.php?title=Sokoban_solver_%22scribbles%22_by_Brian_Damgaard_about_the_YASS_solver

https://www.sciencedirect.com/science/article/pii/S0004370215000867#br0090

https://www.sciencedirect.com/science/article/pii/S0004370201001096

https://algorithmsinsight.wordpress.com/graph-theory-2/ida-star-algorithm-in-general/

http://codeanalysis.fr/sokoban/

http://sokobano.de/wiki/index.php?title=Sokoban_solver_%22scribbles%22_by_Florent_Diedler_about_the_Sokolution_solver

https://baldur.iti.kit.edu/theses/SokobanPortfolio.pdf