Escola Politénica - USP

Departamento de Engenharia de Computação e Sistemas Digitais

PCS3438: Inteligência Artificial

Resolução do Exercício Programa 01

Autores:
Douglas Ramos[1]
NUSP: 9853300
Rafael Higa[2]
NUSP: 9836878

Professor:
Anna Reali

Turma: 01

20 de setembro de 2019

[1]douglas.ramos@usp.br
[1]rafael.seiji.higa@usp.br

Sumário

Lista de Figuras

Lista de Tabelas

1 Introducao

Este relatório tem por objetivo mostrar os resultados obtidos na elaboração do exercício programa 1, proposto pela disciplina "PCS3438: Inteligência Artificial", ministrada no semestre 8 (2019) do curso de Engenharia Elétrica com Ênfase em computação da Escola Politécnica, como avaliação parcial da disciplina. O código completo da solução do exercício programa está presente no seguinte repositório: https://github.com/douglasramos/pacman-ia-search.

2 A* Search Algorithm

Como se sabe através da literatura relacionada a técnicas de busca, o algoritmo A* se assemelha muito ao algoritmo de busca com custo uniforme, diferenciando-se pelo fato de que além do custo intrínseco aos ramos que ligam os nós (estados), insere-se ainda um custo relacionado à heurística do problema.

De maneira resumida, a busca A* inicia através do estado inicial do problema e expande esse estado a fim de encontrar seus sucessores. Todo nó filho encontrado é adicionado à lista de fronteira em ordem de prioridade, na qual o nó de menor custo vem primeiro, e o de maior, por último. A cada iteração, o algoritmo escolhe o primeiro nó presente na fronteira e o expande. Esse processo se repete até que se encontre o nó meta. Mais uma vez, é importante salientar que o diferencial deste algoritmo se dá na ordem de expansão dos nós, onde os seus custos são o fator principal e este peso recebe forte influência da função heurística associada ao problema.

A seguir, mostram-se snippets de código da solução implementada para o algoritmos A*.

Neste primeiro snippet, exibe-se a inicialização da lista de nós visitados (closedList) e a lista de nós na fronteira/borada (openList). Note que a estrutura de dados para representar a closedList é, com efeito, uma Lista simples. Porém, para o caso da fronteira, utilizamos uma fila com prioridade. Esta estrutura de dados foi aproveitada do arquivo **util.py**. A sua peculiaridade é que ela retorna, no contexto de um *pop*, o item da fila como menor número de prioridade. Este número no presente contexto será o custo total associado ao estado.

Finalmente, adiciona-se o estado inicial à fronteira e depois se faz um *pop* da openList. Como nesse momento a lista possui apenas o estado inicial, ele mesmo que é retornado da fila. Este parece, à primeira vista, um passo desnecessário, mas assim está rigorosamente conforme o rigor da teoria relacionada ao algoritmo A*. Além disso, é importante para fim estatístico, pois a openList possui um estado interno (count) que armazena a quantidade

de estados que já passaram pela lista.

```
# initialize the closedList [list that contains the expanded/visited nodes]
closedList = []

# initialize the openList [list (implemented as a PriorityQueue)
# that contains the border nodes (states of the tree)]
openList = util.PriorityQueue()

# Add the problem initial state to the openList with priority number 0
# and then, pop out the same state.
openList.push((problem.getStartState(), [], 0), 0 +
               heuristic(problem.getStartState(), problem))  # add

(currentState, toCurrentStateDirection,
 toCurrentStateCost) = openList.pop()  # pop out
```

No snippet, mostra-se o restante da implementação que está envolta de um grande loop implementado com um *while*. Esta iteração é interrompida quando o *currentState* (estado que está sendo expandido) é o estado meta (*goalState*). A iteração começa adicionando o estado atual à lista de estados visitados. Após isso, obtemos os sucessores do estado. Com isso, realizamos um iteração sobre a lista de sucessores obtidas pelo passo anterior onde para cada sucessor avaliamos se este já é um estado presente na lista de nós visitados. Se não está presente em tal lista, adicionamos tal estado sucessor na lista de fronteira, com um número de prioridade da fila sendo o custo total associado a este nó, isto é, o custo do caminho até o nó somado com o valor de sua heurística.

Pode-se perceber que no segundo laço *for* do código, onde confrontamos o estado do sucessor com a lista de nós visitados, mesmo que o nó já tenha sido expandido, se ele, na corrente iteração, possui um custo associado menor do que aquele presente na lista de nós visitados, a variável *isStateVisited* permanece como false, e então o sucessor será adicionado novamente à openList, porém com custo menor.

Esse é um passo essencial, pois é um cenário que ocorre recorrentemente dentro de um problema de busca, e que se não for avaliado corretamente, não contemplará caminhos de resolução mais otimizados.

Outro ponto que gostaríamos de chamar atenção, visando o pleno entendimento do código, é que as variáveis *toCurrentStateCost* e *toCurrentStateDirection* correspondem respectivamente ao custo do caminho (sem contar a heurística) para chegar ao estado atual a partir do estado inicial e a sequência de direções que o *Agent* precisa tomar para do estado

inicial chegar ao estado atual. Dessa forma, fica claro que o retorno da função deve ser *toCurrentStateDirection*, que será a sequência de direções que devem ser tomadas para que o Agent chegue ao estado meta.

```
# iterate through the tree until get the goal state
while not problem.isGoalState(currentState):
    # add the currentState to the closedList
    closedList.append((currentState, toCurrentStateCost))

    # get the state successors
    successors = problem.getSuccessors(currentState)

    # successor = (state, direcion, cost)
    # iterate through each successor
    for successor in successors:

        successorState = successor[0]
        successorDirection = successor[1]
        successorCost = successor[2]

        isStateVisited = False
        cost = toCurrentStateCost + successorCost
        direction = toCurrentStateDirection + [successorDirection]

        for (visitedState, visitedToCost) in closedList:
            # if the successor has already been visited and still
            # has a new cost greater then the previous one,
            # check isStateVisited as true
            if (successorState == visitedState) and (cost >= visitedToCost):
                isStateVisited = True
                break

        if not isStateVisited:
            # add the successor State to the openList with its totalCost as
            # the priority Queue number
            totalCost = cost + heuristic(successorState, problem)
            openList.push((successorState, direction, cost), totalCost)

    # get the new current State which will be
    # the state with smaller total cost
    (currentState, toCurrentStateDirection,
    toCurrentStateCost) = openList.pop()

return toCurrentStateDirection
```

3 Heurísticas

Foram testadas as seguintes heurísticas:

- Distância de Manhattan;
- Distância Euclidiana;
- Quadrado da Distância Euclidiana;
- Distância entre o Pacman (agente) e a comida em relação às coordenadas do eixo X do tabuleiro;
- Distância entre o Pacman (agente) e a comida em relação às coordenadas do eixo Y do tabuleiro;
- Máximo entre as distâncias entre o agente e a comida em relação, respectivamente, aos eixos X e Y do tabuleiro.

As heurísticas foram implementadas em um arquivo separado - nomeado "heuristics.py". Para utilizá-las, basta que sejam chamadas no terminal conforme o nome das respectivas def's.

Acerca das heurísticas testadas, nota-se que o próprio código já fora previamente elaborado de forma a calcular o número de nós gerados e o tempo demandado para encontrar a solução, para cada jogo executado. Tais métricas foram utilizadas como critérios de avaliação de desempenho das diferentes heurísticas testadas.

O tabuleiro de referência adotado para os testes fora o "Big Maze", retratado na figura 1.

É possível constatar que a solução encontrada através do algoritmo A* para todas as heurísticas testadas é a mesma. Assim, o parâmetro "d", que corresponde à profundidade da solução na árvore de busca, é o mesmo para todos os exemplos.

Dessa forma, o fator de expansão efetivo b* pode ser estimado exclusivamente pelo número de nós gerados N. Quanto menor o valor de N, menor o valor de b*, e mais próximo de 1 o parâmetro b* será (pois a situação limite do problema teria N o menor possível, com $N + 1 = d + 1$ e b* = 1).

É interessante observar as diferenças em termos de nós expandidos conforme as heurísticas de forma gráfica. No tabuleiro de jogo do Pacman, os nós expandidos e cuja função $f(n) = g(n) + h(n)$ foram calculadas se encontram coloridos, com escala de cor variando de cinza a vermelho, conforme o valor de $f(n)$. Os tabuleiros das respectivas heurísticas

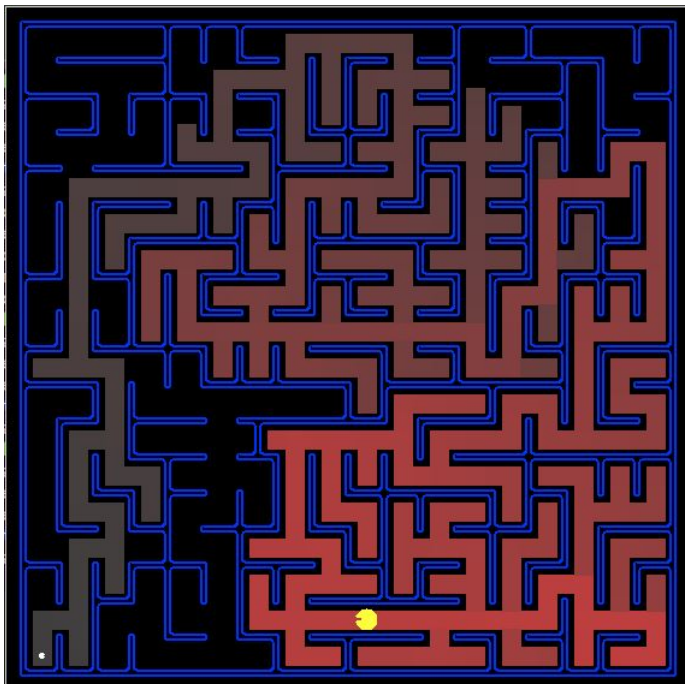

Figura 1: Tabuleiro Big Maze

constam nas figuras 2, 3, 4, 5, 6 e 7. É interessante se observar as diferentes colorações de tabuleiro conforme as diferentes heurísticas adotadas.

Assim, a tabela 1 exibe o desempenho do algoritmo A* para as distintas heurísticas testadas.

Heurística	Tempo de Execução	Quantidade de nós expandidos (N)
Distância de Manhattan	0.2 s	550
Distância Euclidiana	0.2 s	558
Quadrado da distância euclidiana	0.1 s	474
Distância no eixo X	0.2 s	607
Distância no eixo Y	0.2 s	582
máx(Dist. eixo X, Dist. eixo Y)	0.2 s	579

Tabela 1: Desempenho das heurísticas testadas

Diante dos resultados medidos, tem-se o **quadrado da distância euclidiana** como a melhor heurística a ser adotada para o problema. Tanto o tempo de execução é menor em relação às demais heurísticas como a quantidade de nós expandidos é significativamente

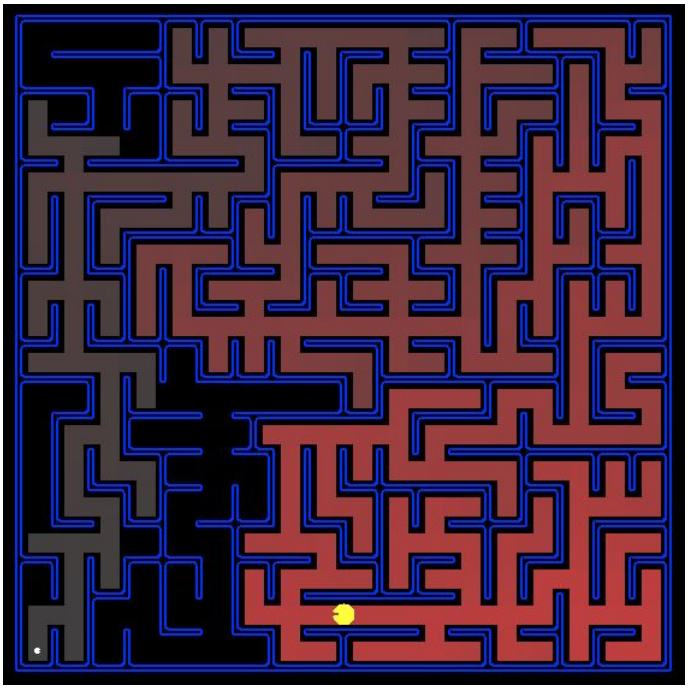

Figura 2: Heurística: Distância de Manhattan

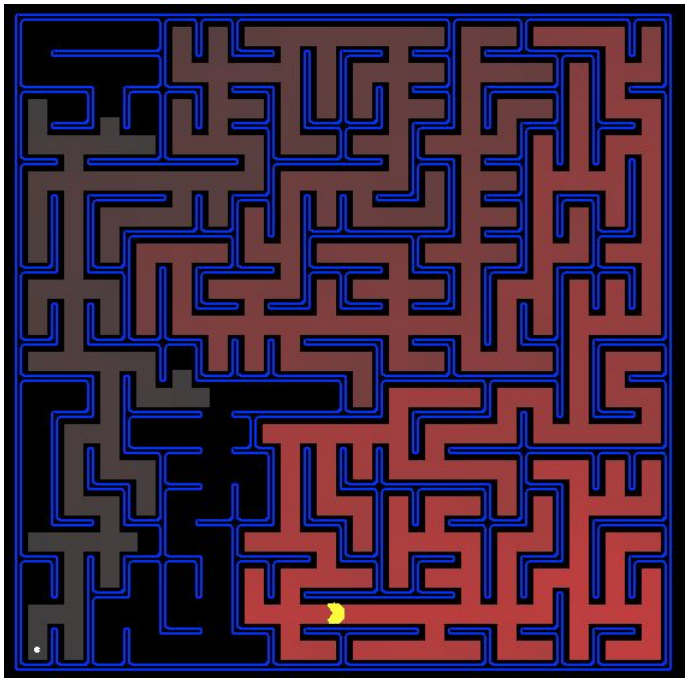

Figura 3: Heurística:Distância Euclidiana

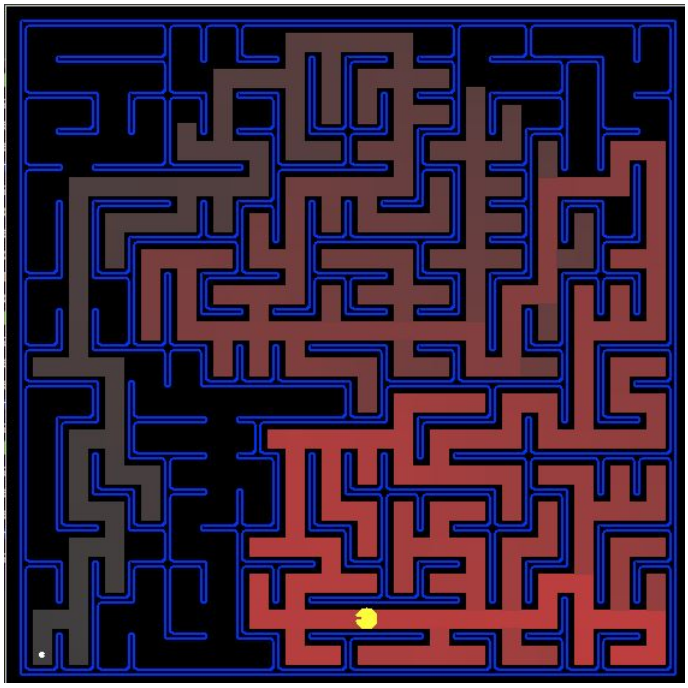

Figura 4: Heurística: Quadrado da Distância Euclidiana

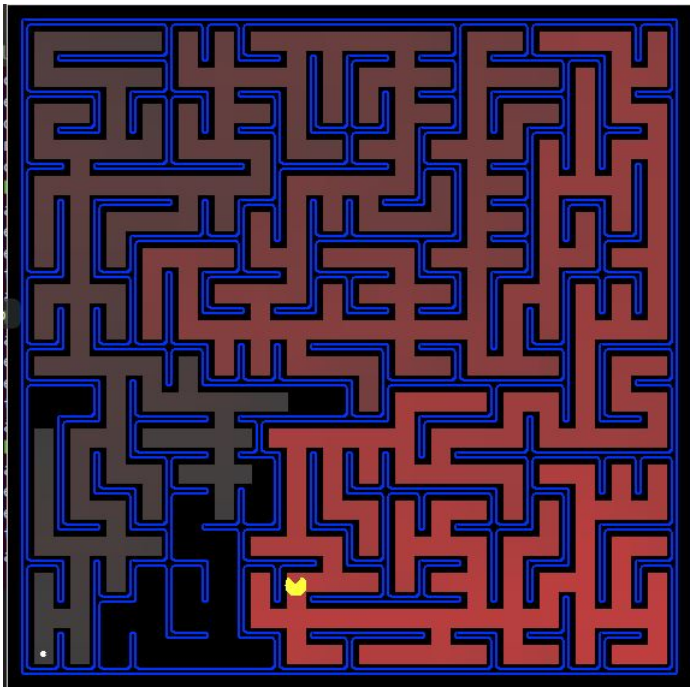

Figura 5: Heurística: Distâncias em relação ao eixo X

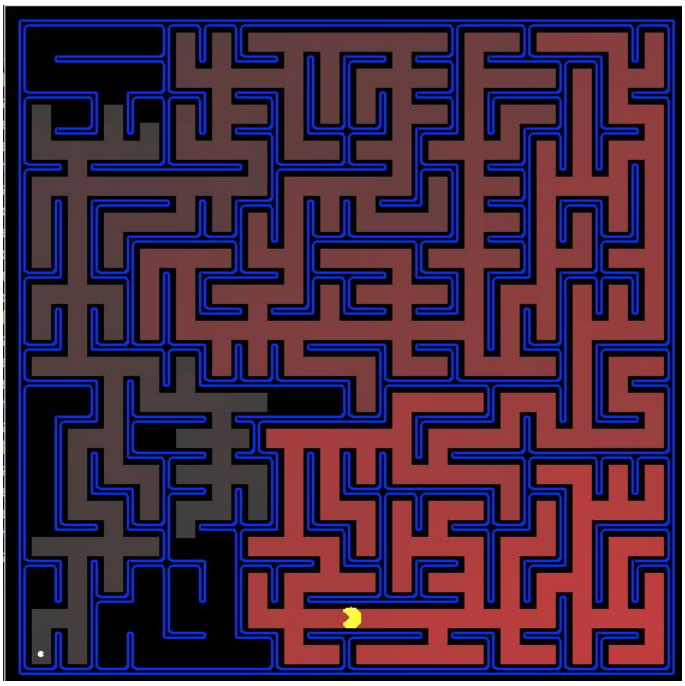

Figura 6: Heurística: Distâncias em relação ao eixo Y

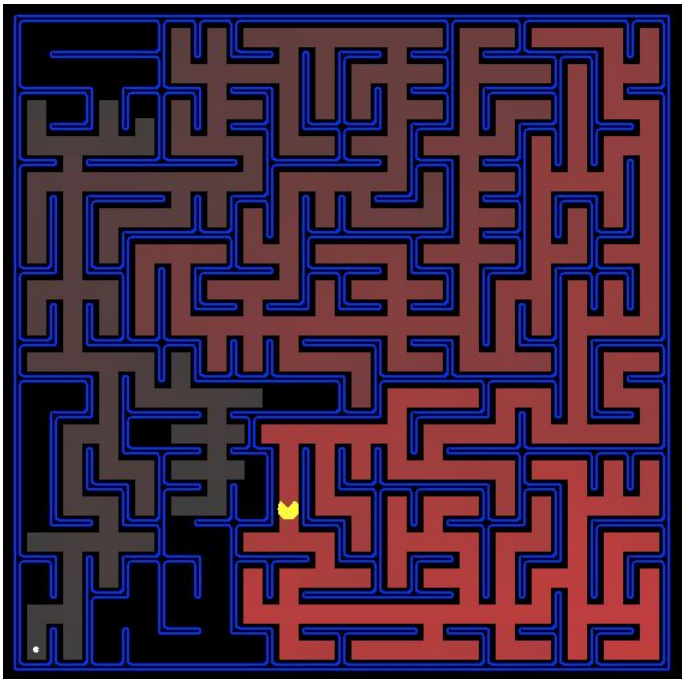

Figura 7: Heurística: Máximo entre as distâncias em relação aos eixos X e Y

menor em relação às demais heurísticas (culminando portanto em um menor b*, mais próximo de 1, visto que possui menor N.