

به نام خدا

ایمان علیپور

۹۸۱۰۲۰۲۴

تمرین عملی ۱

هوش مصنوعی

مفروضات:

در این بخش به مفروضاتی که کردم می‌پردازم:

1. ربات طبق تعریف تابع هیوریستیک، نیاز دارد ایده‌ای از فاصله خودش تا باتری داشته باشد، من فرض کردم به هر خانه جدید که می‌رویم، در آن خانه اطلاعاتی در مورد وضعیت خانه‌های کناری تا مقصد وجود دارد. (مثلا فاصله منتهی یا اقلیدسی هر همسایه تا مقصد نوشته شده است و به ربات داده می‌شود!)
2. ربات زمانی که نیاز دارد از مکانی، جست‌وجو را قطع کند و به مکان دیگری برود که ارتباط مستقیم ندارند، میزان backtrack را حساب نمی‌کند (در واقع این یک مسئله جداگانه است، با داشتن اطلاعات محیطی دو نقطه که قبلا دیده‌ایم این مسیر بصورت یکتا (یا چندتا) و بهینه در زمان $O(1)$ قابل حل شدن است ولی من ایده‌ای نداشتم چگونه آن را حل کنم، مگر اینکه از ایده‌های dynamic programming استفاده کنیم و مسیر بهینه را در هر مقطع ننگه داریم و آپدیت کنیم، که بخاطر کمبود وقت از انجام آن صرف نظر کردم.
3. ممکن است هدف غیرقابل دسترس باشد.
4. برای سادگی طراحی برای ربات یک فضای اولیه تعریف کردم که نیاز به دانستن تعداد ردیف و ستون‌ها داشت، چرا که اگر هربار می‌خواستم یک ردیف و ستون به دید ربات اضافه کنم کمی پیچیدگی بوجود می‌آمد، اما این داده خاص زیادی به ربات نمی‌دهد و کلیت مسئله را تغییر نمی‌دهد، چراکه ربات خارج از محدوده اصلا نمیتواند قدم بگذارد.

منطق پیاده سازی:

من الگوریتم A^* را به همان نحوی که آموزش داده شد پیاده سازی کردم، ابتدا در frontier در یک تاپل، همسایه‌های جدید و فاصله تخمینی آنها تا هدف را نگه‌می‌دارم و کمترین آنها را بسط می‌دهم، اگر frontier خالی شود و هدف پیدا نشود هم خطا نشان می‌دهم. همچنین بعد از هر پیمایش چیزی که ربات دیده است را آپدیت می‌کنم.

پیاده سازی:

ابتدا تابعی نوشتم که ورودی را بخواند (فایل xml) و داده‌های آن را استخراج کند: کد آن به شکل روبرو است:

```
def create_atmosphere():
    #####
    ##### In this function, we create the atmosphere
    ##### This function take no input
    #####
    tree = ET.parse('SampleRoom.xml') # This line reads the xml file and parses it, now we have the root
    root = tree.getroot()

    number_of_rows = len(root) # Assuming we have an n*m grid, I find n and m from the root
    number_of_columns = len(root[0])
    atmosphere = [[' ' for i in range(number_of_columns)] for j in range(number_of_rows)]
    #graphical_atmosphere = [[' ' for i in range(number_of_columns)] for j in range(number_of_rows)]
    robot_x_initial = 0
    robot_y_initial = 0

    for i in range(number_of_rows):
        for j in range(number_of_columns):
            atmosphere[i][j] = ' ' + root[i][j].text
            if(atmosphere[i][j] == 'robot'):
                robot_x_initial = i
                robot_y_initial = j
            if(atmosphere[i][j] == 'Battery'):
                battery_x = i
                battery_y = j
    return atmosphere, robot_x_initial, robot_y_initial, battery_x, battery_y

atmosphere, robot_x_initial, robot_y_initial, battery_x, battery_y = create_atmosphere()
```

خروجی این کد آرایه ای به شکل زیر است که تمام فضای حالت را درون خود دارد:

```
['empty', 'empty', 'empty', 'robot', 'empty', 'obstacle']
['obstacle', 'empty', 'obstacle', 'empty', 'empty', 'empty']
['empty', 'empty', 'empty', 'obstacle', 'empty', 'empty']
['empty', 'obstacle', 'obstacle', 'Battery', 'obstacle', 'empty']
['empty', 'obstacle', 'empty', 'empty', 'obstacle', 'empty']
['empty', 'empty', 'empty', 'empty', 'empty', 'empty']
```

حال تابع `g_n` و توابع هیوریستیک مختلف را معرفی می‌کنم:

```
def g_n(i, j):
    return 1

def bfs_h(i, j):
    return 0

def manhatan_dist_h(i, j):
    return abs(battery_x - i + 1) + abs(battery_y - j + 1)

def euclidean_dist_h(i, j):
    return math.floor(math.sqrt(abs(battery_x - i + 1)**2 + abs(battery_y - j + 1)**2))
```

اولین تابع، تابع `g_n` است که بخاطر ماهیت مسئله مقدار آن همیشه ۱ است، اما با این طراحی به سادگی میتوان حالت های دیگر را نیز پوشش داد. (توجه کنید که از کد جدا شده و چیزی هارد کد نشده است.)

تابع دوم هیوریستیکی است که اگر آنرا به A^* بدهیم جست‌وجوی BFS را انجام می‌دهد، انی تابع برای هر همسایه مقدار ۰ را برمی‌گرداند.

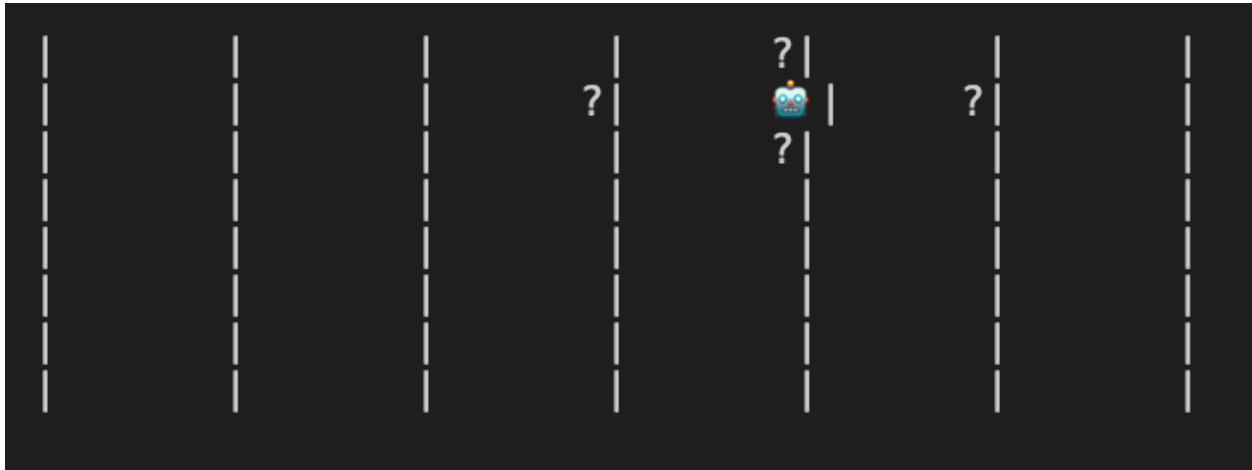
سومین تابع هیوریستیکی است که برای این مسئله بسیار خوب می باشد چراکه فقط حرکت به چپ و راست داریم و س فاصله منتهی هیوریستیک بسیار خوبی است.

چهارمین تابع، فاصله اقلیدسی دو نقطه (همسایه و باتری) را حساب می‌کند.

حال تابعی را معرفی میکنم که چیزهایی که ربات در طی جست‌وجو می داند را در ابتدا تولید میکند، این تابع دید ربات از مسئله را در ابتدا تولید میکند.

```
def create_robot_initial_view():
    # Returns robots initial view
    number_of_rows = len(atmosphere)
    number_of_coloumns = len(atmosphere[0])
    view = [[' ' for i in range(number_of_coloumns+2)] for j in range(number_of_rows+2)]
    view[robot_x_initial+1][robot_y_initial+1] = u"\U0001F916"
    view[robot_x_initial][robot_y_initial+1] = '?'
    view[robot_x_initial+1][robot_y_initial] = '?'
    view[robot_x_initial+2][robot_y_initial+1] = '?'
    view[robot_x_initial+1][robot_y_initial+2] = '?'
    return view
```

خروجی آن شکل زیر خواهد بود:



در بالای ربات دیوار قرار دارد، ربات هیچوقت خانه‌های خارج این جدول را مفروض خود نمی‌داند، بنابراین با داشتن این جدول داده اضافی به ربات ندادم، فقط کمی طراحی را ساده‌تر کردم، اگر اینگونه طراحی نمی‌کردم هر بار باید یک ردیف یا ستون به جدول اضافه می‌کردم که پیچیده‌تر بود و امکان خطا در آن زیاد بود.

حال تابعی را معرفی میکنم که همسایه‌ها هر نقطه را به ربات می‌دهد:

```
def get_neighbors(robot_x, robot_y, view):
    neighbors = []
    if view[robot_x-1][robot_y] == '?':
        neighbors.append((manhatan_dist_h(robot_x-1, robot_y)+g_n(robot_x-1, robot_y), robot_x-1, robot_y))
    if view[robot_x][robot_y-1] == '?':
        neighbors.append((manhatan_dist_h(robot_x, robot_y-1)+g_n(robot_x, robot_y-1), robot_x, robot_y-1))
    if view[robot_x+1][robot_y] == '?':
        neighbors.append((manhatan_dist_h(robot_x+1, robot_y)+g_n(robot_x+1, robot_y), robot_x+1, robot_y))
    if view[robot_x][robot_y+1] == '?':
        neighbors.append((manhatan_dist_h(robot_x, robot_y+1)+g_n(robot_x, robot_y+1), robot_x, robot_y+1))
    return neighbors
```

این تابع ۴ حالت ممکن را نگاه میکند و اگر مفروض ربات باشند آنها را به لیست frontier اضافه میکند، این اضافه شدن در این تابع رخ نمیدهد.

تابع پیمایش:

```
def explore(node, view, x, y):
    cost, i, j = node
    if i == 0 or j == 0 or i == len(atmosphere)+1 or j == len(atmosphere[0])+1:
        view[i][j] = '#'
        i = x
        j = y
    elif atmosphere[i-1][j-1] == 'obstacle':
        view[i][j] = '#'
        i = x
        j = y
    elif atmosphere[i-1][j-1] == 'empty':
        view[i][j] = '.' + u"\U0001F916"
        view[x][y] = 'X'
        if view[i+1][j] != 'X' and view[i+1][j] != '#':
            view[i+1][j] = '?'
        if view[i][j+1] != 'X' and view[i][j+1] != '#':
            view[i][j+1] = '?'
        if view[i][j-1] != 'X' and view[i][j-1] != '#':
            view[i][j-1] = '?'
        if view[i-1][j] != 'X' and view[i-1][j] != '#':
            view[i-1][j] = '?'
    else:
        view[i][j] = 'Y'
        view[x][y] = 'X'
    return i, j
```

این تابع اگر نود جدید مانع یا نباشد، ربات را به خانه جدید می‌برد، همچنین در خانه‌ای که ربات در آن بوده و خارج شده X قرار میدهد یعنی جست‌وجو شده و هدف در آن نبوده، همچنین اینکه همسایه‌ها جدید را با علامت ؟ نشان‌دار میکند.

خروجی تصادفی این تابع:

			? #	# X X #	? X X # #	? X X ? 	? # ?
--	--	--	--------	------------------	-----------------------	----------------------	-------------

تابع نمایش دهنده دید ربات:

```
def print_view(view):  
    s = [[str(e) for e in row] for row in view]  
    lens = [max(map(len, col)) for col in zip(*s)]  
    fmt = '|\\t'.join('{{:{}'.format(x) for x in lens})  
    table = [fmt.format(*row) for row in s]  
    print ('\\n'.join(table))  
    print()  
    print()
```

این تابع با فرمتی که در مثال های قبلی آمده دید ربات را چاپ میکند.

تابع جست و جوی A^* :

```
def a_star():
    # Returns a path from start to end
    frontier = []
    reached_goal = 0
    robot_x = robot_x_initial+1
    robot_y = robot_y_initial+1
    robot_view = create_robot_initial_view()
    moves = 0

    while reached_goal == 0:
        new_neighbors = get_neighbors(robot_x, robot_y, robot_view)
        for i in new_neighbors:
            if i not in frontier:
                frontier.append(i)
        print('Current frontier(f_n, x, y): ' + str(frontier))
        node_to_be_explored = frontier.pop(frontier.index(min(frontier, key = lambda t: t[0])))
        moves += 1
        print()
        print('Node, chosen to be extended(f_n, x, y): ' + str(node_to_be_explored))
        print()
        print_view(robot_view)
        robot_x, robot_y = explore(node_to_be_explored, robot_view, robot_x, robot_y)
        if robot_view[robot_x][robot_y] == 'Y':
            print('End of algorithm.')
            print()
            reached_goal = 1
            print_view(robot_view)
            print()
            print('I found the battery with ' + str(moves) + ' number of moves! :)')
            print()
            break
        elif len(frontier) == 0:
            print('End of algorithm.')
            print()
            print_view(robot_view)
            print()
            print('Goal is unreachable! :(')
            print()
            reached_goal == -1
            break
    return moves
```

طولانی بودن این تابع بیشتر بخاطر پیام های چاپ کردنی است که برای فهم پیاده سازی قرار داده شده اند، این در ابتدا مکان اولیه ربات را میگیرد (برای پرسش از atmosphere، این اطلاعات به ربات داده نمیشود) و همسایه های جدید را به frontier اضافه میکند و کمترین فاصله تخمینی را بسط میدهد و اگر به هدف برسد کار را تمام میکند.

یک نمونه خروجی اتمام الگوریتم:

			?	#	?		
		?	X	X	X	?	
			#	X	X	X	?
				#	X	X	#
		?	?	?	#	X	?
		?	X	X	X	X	?
			?	?	?	?	

End of algorithm.

			?	#	?		
		?	X	X	X	?	
			#	X	X	X	?
				#	X	X	#
		?	Y	#	#	X	?
		?	X	X	X	X	?
			X	X	X	X	?
			?	?	?	?	

تابع BFS:

```
def bfs():
    # Returns a path from start to end
    frontier = []
    reached_goal = 0
    robot_x = robot_x_initial+1
    robot_y = robot_y_initial+1
    robot_view = create_robot_initial_view()
    moves = 0

    while reached_goal == 0:
        new_neighbors = get_neighbors2(robot_x, robot_y, robot_view)
        for i in new_neighbors:
            if i not in frontier:
                frontier.append(i)
        print('Current frontier(f_n, x, y): ' + str(frontier))
        node_to_be_explored = frontier.pop(frontier.index(min(frontier, key = lambda t: t[0])))
        moves += 1
        print()
        print('Node, chosen to be extended(f_n, x, y): ' + str(node_to_be_explored))
        print()
        print_view(robot_view)
        robot_x, robot_y = explore(node_to_be_explored, robot_view, robot_x, robot_y)
        if robot_view[robot_x][robot_y] == 'Y':
            reached_goal == 1
            print()
            print('End of algorithm.')
            print_view(robot_view)
            print()
            print('I found the battery with ' + str(moves) + ' number of moves! :)')
            print()
            break
    return moves
```

بسیار شبیه A^* است، فقط هیوریستیک آن همیشه ۰ است.

خواسته ها:

1. در این سوال از ۳ هیوریستیک ثابت صفر، فاصله منتهی و فاصله اقلیدسی استفاده کردم که هر سه آنها به وضوح تقریبی خوش بینانه دارند، برای تابع ثابت صفر که بدیهی است، همواره فاصله عددی بیشتر مساوی صفر است پس قابل قبول است، اگر مانعی وجود نداشته باشد، بهترین مسیر همان مسیری است که فاصله منتهی نشان میدهد، با برخورد به مانع فاصله ما همواره بیشتر میشود، پس این هیوریستیک نیز مقادیر را کمتر یا مساوی مقدار واقعی نشان میدهد، هیوریستیک فاصله اقلیدسی هم چون اگر یال قطری داشتیم مسیر بهینه را در صورت عدم وجود مانع نشان میداد، همواره مقدار را کمتر یا مساوی مقدار واقعی نشان میدهد پس قابل قبول است، هیوریستیک های دیگر هم قابل استفاده بود، مثلا هیوریستیکی که در چندین حرکت ابتدایی مقادیری بین ۰ تا n به صورت تصادفی به همسایه ها اساین کند و با پیشرفت جست و جو این بازه به ۰ همگرا شود و احتمال ۰ بودن نیز بیشتر شود(البته به شرطی که مقدار اسایت شده از فاصله منتهی کمتر باشند) قابل قبول خواهد بود.
2. مراحل به شکل زیر است:


```
Current frontier(f_n, x, y): [(5, 0, 4), (5, 1, 3), (3, 2, 4),
```

```
Node, chosen to be extended(f_n, x, y): (3, 2, 4)
```

Handwriting practice lines with dashed midlines and solid top/bottom lines. The first line contains a question mark. The second line contains a question mark, a small blue robot icon, and a question mark. The third line contains a question mark.

```
Current frontier(f_n, x, y): [(5, 0, 4), (5, 1, 3), (5, 1, 5),
```

```
Node, chosen to be extended(f_n, x, y): (2, 3, 4)
```

[illegible]

```
Current frontier(f_n, x, y): [(5, 0, 4), (5, 1, 3), (5, 1, 5),
```

```
Node, chosen to be extended(f_n, x, y): (4, 2, 3)
```

[illegible]

```
Current frontier(f_n, x, y): [(5, 0, 4), (5, 1, 3), (5, 1, 5),
```

```
Node, chosen to be extended(f_n, x, y): (4, 2, 5)
```

			? #	? X #	? ?
--	--	--	--------	-------------	--------

```
Current frontier(f_n, x, y): [(5, 0, 4), (5, 1, 3), (5, 1, 5),
```

```
Node, chosen to be extended(f_n, x, y): (3, 3, 5)
```

			#	X X #	? ? ?	
--	--	--	---	-------------	-------------	--

```
Current frontier(f_n, x, y): [(5, 0, 4), (5, 1, 3), (5, 1, 5)
```

```
Node, chosen to be extended(f_n, x, y): (2, 4, 5)
```

			? #		? X X #		? X ? 		? ?

```
Current frontier(f_n, x, y): [(5, 0, 4), (5, 1, 3), (5, 1, 5)
```

```
Node, chosen to be extended(f_n, x, y): (4, 3, 6)
```

			? X #	? X X X #	? X #	? ?
--	--	--	-------------	-----------------------	-------------	--------

```
Current frontier(f_n, x, y): [(5, 0, 4), (5, 1, 3), (5, 1, 5)
```

```
Node, chosen to be extended(f_n, x, y): (3, 4, 6)
```

			? #	? X X #	? X X #	? ? ?
--	--	--	--------	------------------	------------------	-------------

```
Current frontier(f_n, x, y): [(5, 0, 4), (5, 1, 3), (5, 1, 5
```

```
Node, chosen to be extended(f_n, x, y): (4, 5, 6)
```

			? X #	? X X #	? X X #	? ? X
--	--	--	-------------	------------------	------------------	-----------------


```
Current frontier(f_n, x, y): [(5, 0, 4), (5, 1, 3), (5, 1, 5)
```


```
Node, chosen to be extended(f_n, x, y): (3, 5, 5)
```

			#	? X X #		? X X # ?		? X X ? ?
--	--	--	---	------------------	--	-----------------------	--	-----------------------

		?	#	?	?	?
		X	X	X	X	?
		#	X	X	X	👤
			#	X	X	
				#	X	
					?	

[illegible][illegible]

		?	#	?		
	?	X	X	X	?	
		#	X	X	X	?
			#	X	X	#
				#	X	#
			?		X	?
				?	X	?
					?	

		?	#	?		
	?	X	X	X	?	
		#	X	X	X	?
			#	X	X	#
			?	#	X	?
		?		X	X	?
			?	?	?	

			? #	? X X X #	? X X X #	? X X ? 🧠 ?	? ? ?
--	--	--	--------	-----------------------	---------------------------	-------------------------	-------------

			? #	? X X X #	? X X X #	? X X X ?	? # ?
--	--	--	--------	-----------------------	-----------------------	-----------------------	-------------

[illegible][illegible][illegible]

			?	#	?		
		?	X	X	X	?	
			#	X	X	X	?
				#	X	X	#
			?	?	#	X	#
			?	🧊	#	X	?
			?	X	X	X	?
				?	?	?	

			?	#	?		
		?	X	X	X	?	
			#	X	X	X	?
				#	X	X	#
				Y	#	X	#
			?	X	#	X	?
			?	X	X	X	?
				?	?	?	

3. طبعاً با این تغییرات، مسئله مسئله جدیدی خواهد شد و تعداد گامها ممکن است کمتر یا بیشتر شوند.

?	?	?	?	?	?	?	?	?
X	X	X	X	X	X	X	X	?
#	X	#	X	X	X	X	X	?
?	X	X	X	#	#	?	?	?
	#	#					🗨️	?
							?	

?	?	?	?	?	?		
	X	X	X	X	X	?	
	#	X	#	X	X		?
	?	X	X	#	X	X	
		#	#		#	X	?
					?	X	?
						Y	?

```
[ 'robot', 'empty', 'empty', 'empty', 'empty', 'obstacle' ]
[ 'obstacle', 'empty', 'obstacle', 'empty', 'empty', 'empty' ]
[ 'empty', 'empty', 'empty', 'obstacle', 'empty', 'empty' ]
[ 'empty', 'obstacle', 'obstacle', 'empty', 'obstacle', 'empty' ]
[ 'empty', 'obstacle', 'empty', 'empty', 'obstacle', 'empty' ]
[ 'empty', 'empty', 'empty', 'empty', 'empty', 'Battery' ]
```

در اینجا ربات از خانه بالا چپ شروع کرده و باتری در خانه پایین راست را پیدا میکند. تعداد گامها کمتر شده و دلیل آن تعداد کمتر نیاز به تغییر در مکان ادامه frontier بوده است.

	?		?		#		?	
?	X		X		X		X	
	#		X		#		X	
?	X		X		#		X	
?	X		#		?		X	
	?				🔋		X	
			?		X		X	
					?		?	

End of algorithm.

	?		?		#		?	
?	X		X		X		X	
	#		X		#		X	
?	X		X		#		X	
?	X		#		#		X	
	?				Y		#	
					X		#	
			?		X		X	
					?		?	

I found the battery with 31 number of moves! :)

```
['robot', 'empty', 'empty', 'empty', 'empty', 'obstacle']
['obstacle', 'empty', 'obstacle', 'empty', 'empty', 'empty']
['empty', 'empty', 'empty', 'obstacle', 'empty', 'empty']
['empty', 'obstacle', 'obstacle', 'Battery', 'obstacle', 'empty']
['empty', 'obstacle', 'empty', 'empty', 'obstacle', 'empty']
['empty', 'empty', 'empty', 'empty', 'empty', 'empty']
```

مثالی دیگر، با تغییر مکان اولیه
تعداد گامها بیشتر شده که
دلیل آن نیاز بیشتر بیشتر به
تغییر نودی که بسط داده میشود
بوده است(بخش زیادی از
گراف بسط داده شده است)

با تغییر هیوریستیک نودی که برای بسط دادن انتخاب می شود ممکن است متفاوت بشود و حتی مسیر
انتهایی نیز همینطور که با فرض قابل قبول و سازگار بودن هیوریستیک مسیر بهینه خواهد بود اما این
مسیر بهینه ممکن است یکتا نباشد. برای هیوریستیک همواره ۰ در مثال بالا داریم:

Node, chosen to be extended(f_n, x, y): (1, 4, 4)

	#		#		#		#	
#	X		X		X		X	
	#		X		#		X	
#	X		X		#		X	
#	X		#		?		#	
#	X		#		X		#	
#	X		X		X		🔋	
	#		#		#		?	

End of algorithm.

	#		#		#		#	
#	X		X		X		X	
	#		X		#		X	
#	X		X		#		X	
#	X		#		#		X	
#	X		#		X		#	
#	X		X		X		X	
	#		#		#		?	

I found the battery with 55 number of moves! :)

```
['robot', 'empty', 'empty', 'empty', 'empty', 'obstacle']
['obstacle', 'empty', 'obstacle', 'empty', 'empty', 'empty']
['empty', 'empty', 'empty', 'obstacle', 'empty', 'empty']
['empty', 'obstacle', 'obstacle', 'Battery', 'obstacle', 'empty']
['empty', 'obstacle', 'empty', 'empty', 'obstacle', 'empty']
['empty', 'empty', 'empty', 'empty', 'empty', 'empty']
```

که واضحا تعداد گامها بیشتر شده چون
هیوریستیک منتهی هیوریستیک بهتری برای
این سوال است(خوش بینانه تقریب میزند
اما تقریب آن منطقی تر است)
و در این حالت تقریبا کل فضای حالت بررسی
شده است.

برای هیوریستیک اقلیدسی داریم:

```
Node, chosen to be extended(f_n, x, y): (1, 4, 4)

#|   #|   #|   #|   #|   #|   #|
#|   X|   X|   X|   X|   X|   #|
#|   #|   X|   #|   X|   X|   X|   #
#|   X|   X|   X|   #|   X|   X|   #
#|   X|   #|   X|   X|   #|   X|   #
#|   X|   X|   X|   X|   X|   X|   #
#|   #|   #|   #|   #|   ?|   #|

End of algorithm.

#|   #|   #|   #|   #|   #|   #|
#|   X|   X|   X|   X|   X|   X|   #
#|   #|   X|   #|   #|   X|   X|   X|   #
#|   X|   X|   #|   #|   Y|   #|   X|   #
#|   X|   #|   X|   X|   X|   #|   X|   #
#|   X|   X|   X|   X|   X|   X|   X|   #
#|   #|   #|   #|   #|   ?|   #|

I found the battery with 55 number of moves! :)

['robot', 'empty', 'empty', 'empty', 'empty', 'obstacle']
['obstacle', 'empty', 'obstacle', 'empty', 'empty', 'empty']
['empty', 'empty', 'empty', 'obstacle', 'empty', 'empty']
['empty', 'obstacle', 'obstacle', 'Battery', 'obstacle', 'empty']
['empty', 'obstacle', 'empty', 'empty', 'obstacle', 'empty']
['empty', 'empty', 'empty', 'empty', 'empty', 'empty']
```

همانطور که مشاهده می شود و انتظار می رفت به خوبی هیوریستیک منتهی نیست و تعداد گامهای آن بسیار بیشتر است.

5. در مثال قبلی جست و جوی BFS را نیز نشان دادم که همان A^* با هیوریستیک همواره برابر صفر است.

```
Node, chosen to be extended(f_n, x, y): (1, 4, 4)

#|   #|   #|   #|   #|   #|   #|
#|   X|   X|   X|   X|   X|   X|   #
#|   #|   X|   #|   X|   X|   X|   #
#|   X|   X|   X|   #|   X|   X|   #
#|   X|   #|   #|   ?|   #|   X|   #
#|   X|   #|   X|   X|   #|   X|   #
#|   X|   X|   X|   X|   X|   X|   #
#|   #|   #|   #|   #|   ?|   #|

End of algorithm.

#|   #|   #|   #|   #|   #|   #|
#|   X|   X|   X|   X|   X|   X|   #
#|   #|   X|   #|   X|   X|   X|   X|   #
#|   X|   X|   X|   #|   X|   X|   X|   #
#|   X|   #|   X|   X|   #|   X|   X|   #
#|   X|   X|   X|   X|   X|   X|   X|   #
#|   #|   #|   #|   #|   ?|   #|

I found the battery with 55 number of moves! :)

['robot', 'empty', 'empty', 'empty', 'empty', 'obstacle']
['obstacle', 'empty', 'obstacle', 'empty', 'empty', 'empty']
['empty', 'empty', 'empty', 'obstacle', 'empty', 'empty']
['empty', 'obstacle', 'obstacle', 'Battery', 'obstacle', 'empty']
['empty', 'obstacle', 'empty', 'empty', 'obstacle', 'empty']
['empty', 'empty', 'empty', 'empty', 'empty', 'empty']
```

همانطور که مشاهده می شود و انتظار می رفت با هیوریستیک منتهی نتایج بهتراند و با گامهای کمتری به جواب میرسیم.