گزارش پروژه ی اول هوش مصنوعی

البرز محمودیان 810101514

فایل مربوط به کد پایتون در html: AI_CA1_F03_student_notebook.html

سوال 1)

state: استیت در واقع حالت کنونی برد ما است (یک پازل). که یک ارایه ی دو بعدی می باشد که هر خانه از ارایه دو بعدی 1 یا 0 است که 0 یعنی ان خانه خاموش و 1 یعنی که ان خانه روشن است.

Initial state: استیت اولیه استیتی است که ما از ان شروع می کنیم که توسط تابع create random board ساخته می شود

Actions: اکشن های ما در واقع toggle کردن لامپ ها است. که ما در هر استیت به اندازه ی خانه های ارایه دو بعدی می توانیم رو هر خانه این اکشن را انجام دهیم. و هر اکشن ما را از استیتی به استیت دیگر می برد. فقط toggle کردن هر خانه از ارایه دو بعدی نه تنها لامپ ان خانه را برعکس می کند بلکه لامپ خانه های همسایه اش را نیز برعکس می کند.

Path cost: هزینه اینکه از هر استیت به استیت دیگر برویم به اندازه ی کمترین تعداد toggle های مورد نیاز است که ما را از استیت اولیه به استیت ثانویه می رساند. مثلا اگر از یک استیت با toggle کردن یک خانه به استیت دیگری برسیم ، هزینه رسیدن از استیت قبلی به استیت فعلی 1 می باشد.

Goal state: استیت هدف استیتی است که در ان تمام خانه های ارایه ی دو بعدی 0 باشد. یعنی تمام لامپ ها خاموش باشد. که باید از استیت اولیه با انجام دادن اکشن های مختلف به استیت نهایی برسیم

سوال2)

```python
# TODO: Must return a list of tuples and the number of visited nodes
def bfs_solve(puzzle: LightsOutPuzzle) -> Tuple[List[Tuple[int, int]], int]:
    frontier = [puzzle]
    explored = []
    if (puzzle.is_solved()):
        return ([], 1)
    while(len(frontier)):
        chosen_puzzle = frontier.pop(0)
        parent_actions = chosen_puzzle.actions
        explored.append(chosen_puzzle)
        for (x,y) in chosen_puzzle.get_moves():
            child = deepcopy(chosen_puzzle)
            child.actions = parent_actions + [(x, y)]
            child.toggle(x, y)
            if(not_in_explore(child, explored) and not_in_frontier(child, frontier)):
                frontier.append(child)
                if (child.is_solved()):
                    return (child.actions, len(explored))
```

BFS : در پیپادی سازه الگوریتم BFS ابتدا یک لیست frontier و یک لیست explored درست می کنیم. که frontier ما یک FIFO queue می باشد.پس همیشه باید نودی که اول صف هست را برای بسط دادن انتخاب کنیم که اینکار را با pop(0) انجام میدهیم. حال نود انتخاب شده را وارد explored می کنیم و به ازای هر اکشنی که وجود دارد فرزندان نود انتخاب شده را به وجود می آوریم. اکشن هایی که وجود دارد به اینصورت است که ما هر المان از ماتریس n*n را می توانیم toggle کنیم . پس یعنی هر نود انتخاب شده می تواند n*n فرزند تولید کند که این فرزندان باید وارد لیست frontier شوند. فقط چون frontier ما یک FIFO queue است پس فرزندان را به انتهای لیست اضافه می کنیم (append()) . حال فقط باید چک کنیم که فرزندانی که می خواهیم به frontier اضافه کنیم قبلا بسط داده نشده باشند یعنی در explored نباشند که ما مسیر های تکراری را چک نکنیم. نکته ی دیگری که وجود دارد این است که الگوریتم BFS از انجایی که به صورت سطحی جلو می رود ، هنگامی که به یک نود می رسد که می خواهد ان را وارد frontier کند ، تضمین می شود که این مسیر کوتاه ترین مسیری بوده است که می توانستیم به این نود برسیم و قرار نیست مسیر دیگری وجود داشته باشد که با هزینه کمتری به این نود برسد. پس می توانیم هنگامی که می خواهیم یک نود را وارد frontier کنیم علاوه بر اینکه چک کنیم که در explored نباشد بلکه چک کنیم که در frontier هم نباشد . چرا که اگر نودی که می خواهیم به frontier اضافه کنیم قبلا در frontier وجود داشته باشد، هزینه قبلی قطعا کوچکتر مساوی هزینه فعلی است و نیازی نیست که نود را با یک هزینه بیشتر وارد frontier کنیم. در پی همین نکته که گفتیم ،هنگامی که می خواهیم یک نود را وارد frontier کنیم تضمین می شود که این کوتاه ترین مسیر رسیدن به این نود است، پس می توانیم test goal را هنگامی انجام دهیم که می خواهیم نود را وارد frontier کنیم . چرا که مطمعنیم که بعدا قرار نیست این نود با هزینه بهتری وارد frontier شود و این خاصیت الگوریتم BFS است. پس الگوریتم BFS، بهینه می باشد و همواره جواب optimal را بر می گرداند

: IDS

```python
# TODO: Must return a list of tuples and the number of visited nodes
def ids_solve(puzzle: LightsOutPuzzle) -> Tuple[List[Tuple[int, int]], int]:
    if (puzzle.is_solved()):
        return ([], 1)

    layer = 1
    while(True):
        (child_actions , len_explored) = dfs_with_depth_n(layer, puzzle)
        if(child_actions and len_explored):
            return (child_actions, len_explored)
        layer += 1
```

می دانیم که در این الگوریتم باید با لایه های مختلف DFS بزنیم تا بالاخره به هدف برسیم. پس یک تابع می زنیم که برای ما DFS را تا لایه ی مشخص n بزند.حال در تابع ids_solve ، تابعی که برای DFS زدیم را با لایه های مختلف صدا می زنیم. یعنی یک layer تعریف می کنیم که از 1 شروع می شود. یعنی بار اول تا لایه ی اول DFS می زنیم. سپس layer را یکی اضافه می کنیم و این دفعه تا لایه ی دوم DFS می زنیم و انقدر ادامه می دهیم تا بالاخره در لایه ی n ام به هدف برسیم.

```python
def dfs_with_depth_n(n, puzzle):
    frontier = [puzzle]
    explored = []
    layer = 0

    while(frontier):
        chosen_puzzle = frontier.pop()
        if(chosen_puzzle.layer < n):
            (parent_actions, parent_layer) = (chosen_puzzle.actions, chosen_puzzle.layer)
            explored.append(chosen_puzzle)
            for (x,y) in chosen_puzzle.get_moves():
                child = deepcopy(chosen_puzzle)
                child.actions = parent_actions + [(x, y)]
                child.layer = parent_layer + 1
                child.toggle(x, y)
                if(not_in_explore(child, explored) and not_in_frontier(child, frontier)):
                    if (child.is_solved()):
                        return (True, child.actions, len(explored))
                    frontier.append(child)
    return (False, 0, len(explored))
```

حال برای پیاده سازی الگوریتم DFS تا لایه ی مشخص n ام، همان مانند الگوریتم BFS عمل می کنیم با این تفاوت که این دفعه frontier ما یک LIFO queue یا همان استک است. یعنی ایندفعه به جای اینکه هر دفعه نود اول لیست را برای بسط دادن انتخاب کنیم، نود آخر لیست را برای بسط دادن انتخاب می کنیم. الگوریتم IDS مانند BFS نیز به گونه ای است که اگر یک نود بسط داده شود باید روی بچه های ان تست گل را انجام دهیم. چرا که هنگامی که یک نود وارد frontier می شود مطمعنیم که دیگر این نود قرار نیست با هزینه ای بهتر وارد frontier شود. پس در پی آن هنگامی که می خواهیم یک نود را وارد frontier کنیم چک می کنیم که قبلا در frontier نیز نباشد. پس الگوریتم IDS نیز بهینه است و همواره جواب اپتیمال را برمی گرداند.

همچنین برای اینکه DFS تا لایه ی n ام اجرا شود، داخل هر puzzle یک attribute به نام layer نگه می داریم که به ما می گوید هر puzzle در لایه ی چندم است. سپس هر نود را که بسط دادیم تا بچه هایش را وارد frontier کنیم layer بچه ها را یکی بیشتر از layer نود بسط داده شده قرار می دهیم .پس در تابع DFS چک می کنیم که هر بار که به یک puzzle رسیدیم که خواستیم ان را بسط دهیم، اگر در لایه ای بزرگتر یا مساوی لایه ی n ام باشد ، دیگر ادامه نمی دهیم و عنصر بعدی را برای بسط دادن انتخاب می کنیم .

: A*

```python
# TODO: Must return a list of tuples and the number of visited nodes
def astar_solve(puzzle: LightsOutPuzzle, heuristic: Callable[[LightsOutPuzzle], int]) -> Tuple[List[Tuple[int, int]], int]:
    state_count = 0
    priority_queue = [(heuristic(puzzle), state_count, 0, puzzle)]
    explored = []
    if (puzzle.is_solved()):
        return ([], 1)
    while(priority_queue):
        (_, _, chosen_puzzle_gn, chosen_puzzle) = heapq.heappop(priority_queue)
        parent_actions = chosen_puzzle.actions
        explored.append(chosen_puzzle)
        if (chosen_puzzle.is_solved()):
            return (chosen_puzzle.actions, len(explored))
        for (x,y) in chosen_puzzle.get_moves():
            child = deepcopy(chosen_puzzle)
            child.actions = parent_actions + [(x, y)]
            child.toggle(x, y)
            if(not_in_explore(child, explored)):
                state_count += 1
                child_gn = chosen_puzzle_gn + 1
                child_fn = heuristic(child) + child_gn
                heapq.heappush(priority_queue, (child_fn, state_count, child_gn, child))
    return ([], 0)
```

در این الگوریتم نیز همانند قبل عمل می کنیم البته با این فرق که frontier ما یک priority queue است که باید نودی را که کمترین f(n) را دارد انتخاب کنیم تا ان را بسط دهیم که ان را توسط heapq پیاده سازی می کنیم. حال پیاده سازی الگوریتم بستگی به تابع h() دارد. مثلا اگه تابع h()، consistence نباشد پس اولین مسیری که به یک نود پیدا می کنیم، لزوما کوتاه ترین مسیر به نود نیست و امکان دارد در اینده مسیر بهتری از مسیر فعلی پیدا کنیم. پس دیگر مثل دو الگوریتم قبلی نمی گوییم که اگر یک نودی از قبلا در frontier است دیگر ان را در frontier اضافه نمی کنیم. بلکه اگر به نودی رسیدیم که از قبل در frontier بوده است، دوباره می تواند با هزینه جدید وارد frontier شود.

البته می توانستیم کاری کنیم که مثلا هنگامی که به نودی رسیدیم که از قبل در frontier بوده است، باید چک کنیم که ایا مسیری که الان به دست اورده ایم کوتاه تر است یا مسیر قبلی.

1- اگر مسیر قبلی کوتاه تر بود که یعنی مسیر جدیدی که به دست اورده ایم بهتر نمی باشد پس لازم نیست frontier را دست بزنیم.

2- اگر مسیر فعلی کوتاه تر بود پس مسیر فعلی بهتر از مسیری است که قبلا در frontier بوده است پس باید مسیر قبلی در frontier را به مسیر فعلی آپدیت کنیم.

اما این کار هزینه زمانی زیادی را ایجاد می کرد چرا که باید به ازای اضافه شدن هر نود در فرانتیر چک می کرد که اگر قبلا، ان نود با هزینه بیشتری در فرانتیر بوده باشد، هزینه را به هزینه جدید اپدیت کند.

همچنین تست گل را نمی توانیم مانند دو الگوریتم قبلی هنگامی انجام دهیم که یک نود را می خواهیم به frontier اضافه کنیم. چرا که ممکن است هزینه ای که تا الان به دست اورده ایم بهینه نباشد و امکان داشته باشد که این نود در اینده با هزینه بهتری وارد frontier شود. پس تست گل را باید هنگامی انجام دهیم که می خواهیم یک نود را بسط دهیم و ان را وارد لیست explored کنیم. چرا که دیگر مطمعنیم قرار نیست هزینه بهتری به دست اید.
همچنین این الگوریتم زمانی بهینه خواهد بود که تابع h() ما admissible باشد. در این صورت مسیر به دست امده optimal خواهد بود.

سوال3)

Heuristic اول:

```python
def heuristic1(puzzle: LightsOutPuzzle):
    ones = []
    distance = 0
    for i in range(puzzle.size):
        for j in range(puzzle.size):
            if(puzzle.board[i][j] == 1):
                ones.append((i, j))
    for i in range(len(ones)):
        for j in range(len(ones)):
            distance += (abs(ones[i][0] - ones[j][0]) + abs(ones[i][1] - ones[j][1]))
    return distance / 2
```

این تابع در واقع کاری که میکند این است که فاصله ی هر یک در پازل را نبست به بقیه یک ها در پازل در می اورد و تمام این فاصله ها را به ازای تمام یک ها با هم جمع می کند. و در نهایت این فاصله را تقسیم بر دو می کند چرا که ما در واقع فاصله بین هر دو یک را دو بار حساب کرده ایم. این heuristic ، admissible نمی باشد چرا که مقدار تخمین زده شده می تواند از هزینه مورد نیاز واقعی برای رسیدن به هدف بیشتر باشد.

Heuristic دوم:

```python
def heuristic2(puzzle: LightsOutPuzzle):

    return sum(sum(row) for row in puzzle.board)

heuristics = [heuristic1 , heuristic2]
```

✓ 0.0s

این تابع در واقع تعداد یک های هر puzzle را بر می گرداند .

این تعریف admissible نیست چرا که مقدار این تابع بعضی وقت ها بیشتر از مقدار واقعی هزینه ای است که باید بپردازیم تا به گل برسیم.

مثلا در حالت روبه رو h() برابر 5 است در حالی که ما با پرداخت یک هزینه و تنها toggle کردن لامپ وسط می توانیم به استیت گل برسیم:

[0 1 0]
[1 1 1]
[0 1 0]

پس این heuristic نه admissible است و نه consistent و همیشه راه حل optimal را به ما نمی دهد.

Heuristic سوم:

```
def heuristic3(puzzle: LightsOutPuzzle):

    return sum(sum(row) for row in puzzle.board) / 5
```

برای اینکه بتوانیم یک consistent heuristic بسازیم کافی است که خروجی heuristic2 را تقسیم بر 5 کنیم. چرا که برای خاموش کردن هر پنج چراغ ما حداقل به یک toggle کردن نیاز داریم و از انجایی که ما تعداد یک ها را تقسیم بر پنج می کنیم پس هیچگاه مقدار تخمین زده شده بیشتر از حداقل مقدار toggle مورد نیاز نمی باشد. پس heuristic ما admissible خواهد بود و همواره جواب بهینه را برخواهد گرداند.

Heuristic چهارم:

```
def heuristic4(puzzle: LightsOutPuzzle):
    count = 0
    m = 0
    for i in range(puzzle.size):
        for j in range(m, puzzle.size, 2):
            if(puzzle.board[i][j]):
                count += 1
        if m == 0:
            m = 1
        else:
            m = 0
    return count
```

این تابع تعداد یک های هر پازل را به صورت یک در میان می شمارد. مثلا برای یک پازل 5 * 5 یک هایی که در جایگاه های مشخص شده هستند می شمارد :

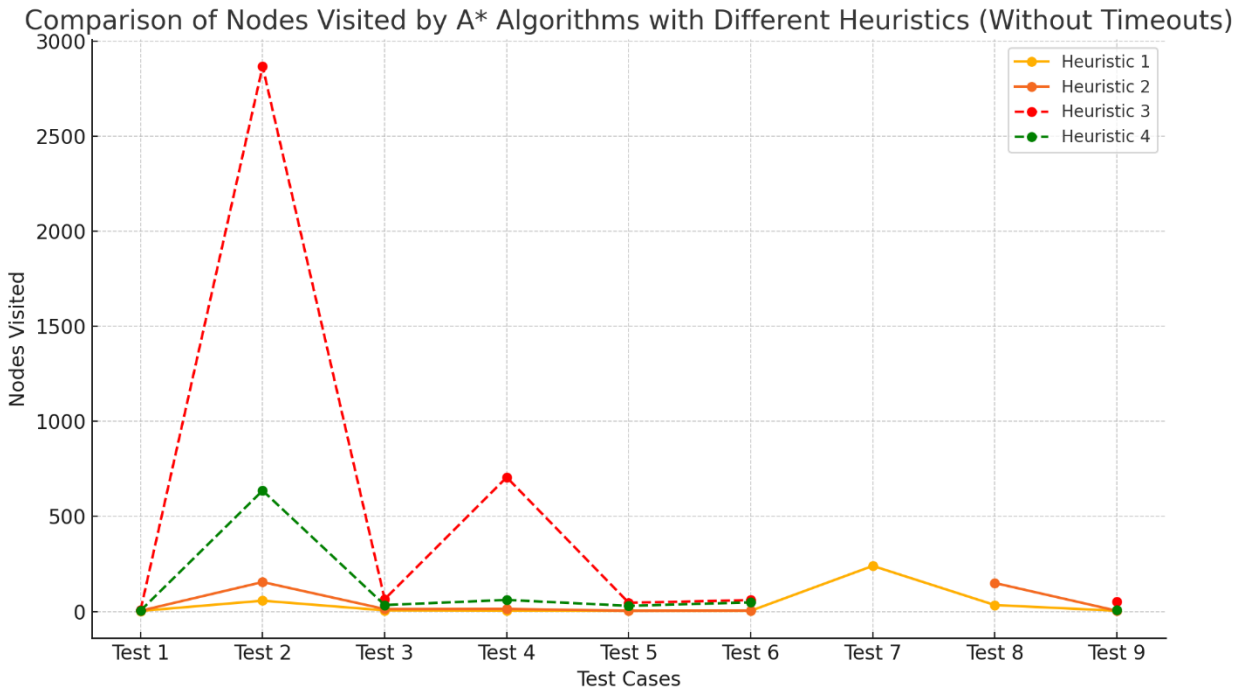
[1 1 0 1 0]
[0 1 0 0 1]
[0 0 1 1 1]
[1 0 1 0 0]
[1 1 0 0 1]

که مثلا در این حالت تابع عدد 6 را برمیگرداند.
این heuristic نیز admissible نمی باشد چرا که مثلا در حالت زیر تابع به ما عدد 3 را بر می گرداند در حالی که می توان با یک بار انجان عملل toggle به گل رسید. یعنی مقدار تخمین زده شده بیشتر از مقدار واقعی است. پس این heuristic نه admissible است و نه consistent.

[1 1 1]
[0 1 0]
[0 0 0]

سوال 4)

حال می توانیم به صورت کلی تعداد نود های ویزیت شده را در 9 تا تست کیس به ازای هر چهار تا heuristic در نمودار زیر ببینیم:

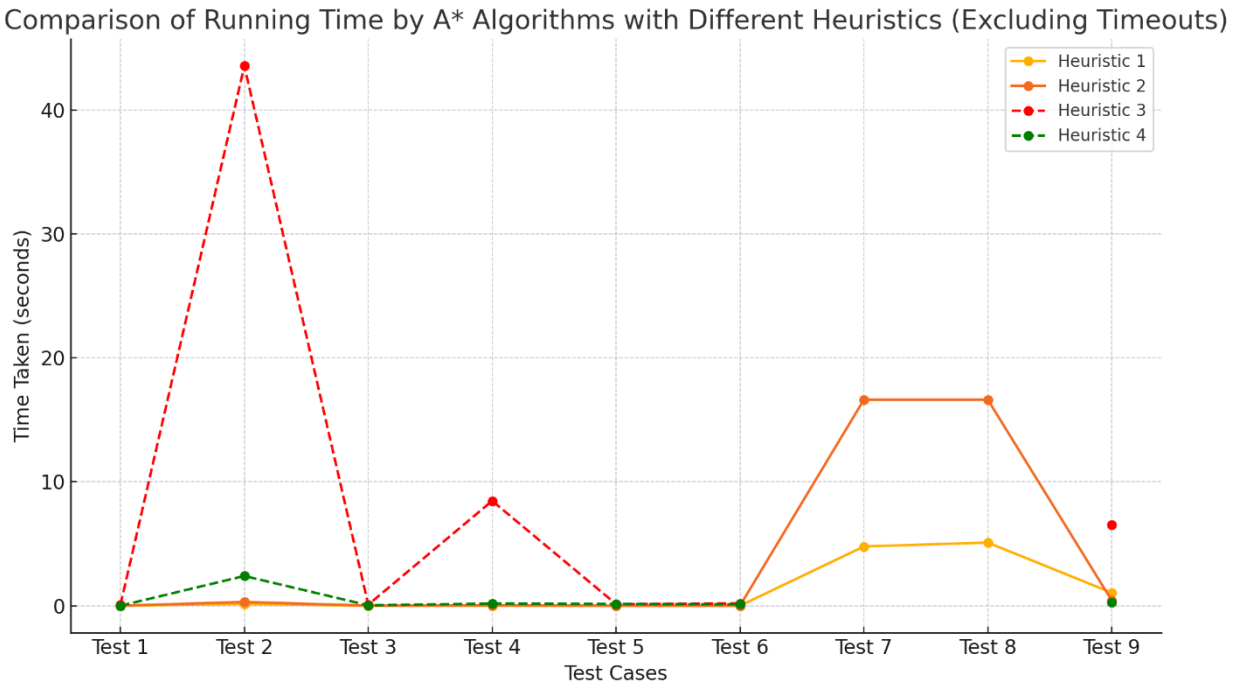


- *A Heuristic1 Average Nodes**: 39.78
- *A Heuristic2 Average Nodes**: 78.00
- *A Heuristic3 Average Nodes**: 527.89
- *A Heuristic4 Average Nodes**: 124.67

در این تست کیس ها تمامی heuristic ها جواب optimal را به ما برگردانده اند. البته اینکه می گوییم heuristic های 1و2و4 admissible نیستند دلیل بر این نیست که هیچگاه جواب optimal را نمی دهند.

البته این چهار heuristic در تعداد نود های visit شده متفاوت هستند که می توان دید heuristic اول به طور میانگین تعداد نود های کمتری از بقیه visit کرده است و هچنین heuristic سوم با اینکه admissible است اما با visit کردن میانگین تعداد نود های بیشتری نسبت به جواب می رسد.

همچنین در نمودار زیر می توانیم مدت زمان هر تست کیس را برای هر heuristic ببینیم:



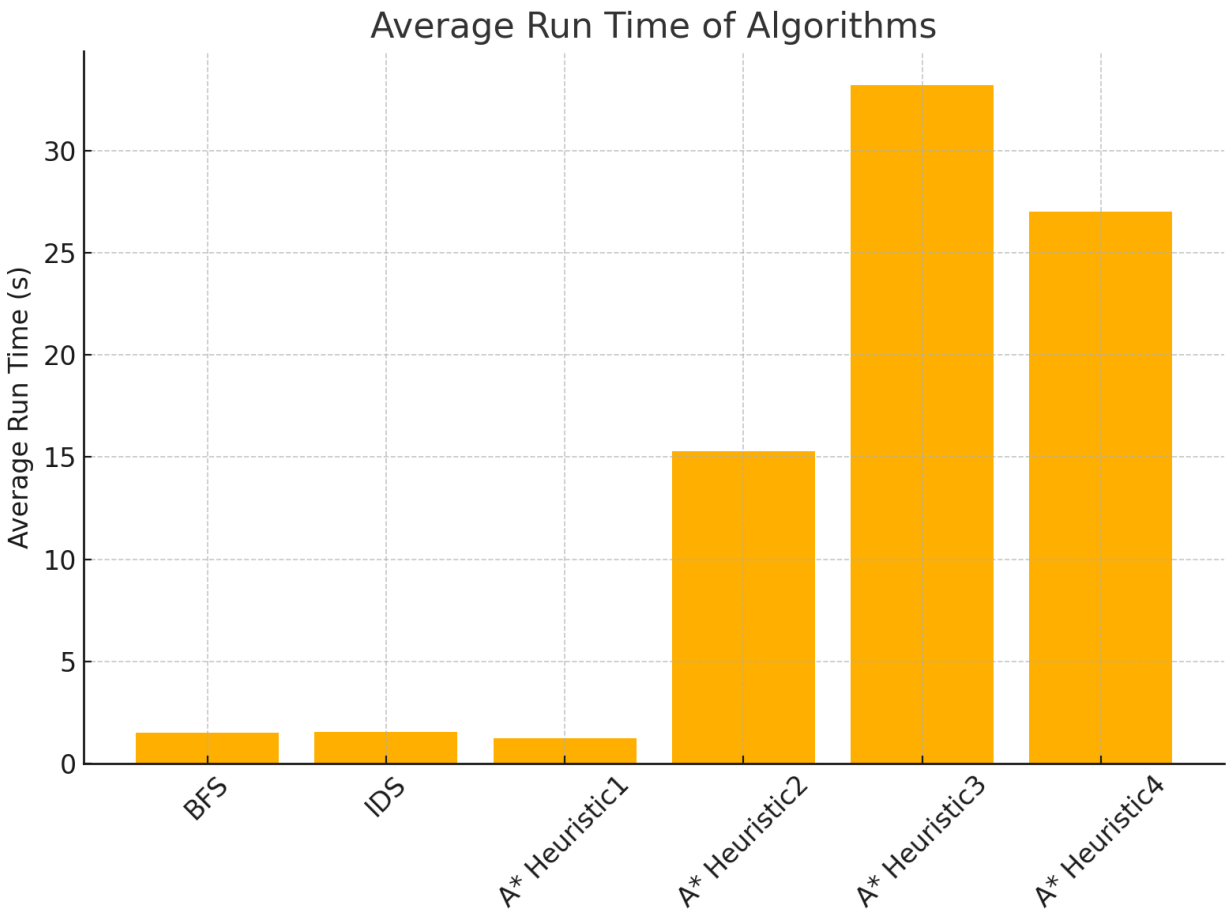
- *A Heuristic1 Average Time**: 1.23 seconds
- *A Heuristic2 Average Time**: 15.27 seconds
- *A Heuristic3 Average Time**: 33.22 seconds
- *A Heuristic4 Average Time**: 27 seconds

می توان دید که طبق انتظار heuristic سوم از انجایی که نسبت به بقیه نود های بیشتری را ویزیت می کرد پس میانگین مدت زمان بیشتری نیز طول کشیده است تا به هدف برسد. همچنین heuristic اول با میانگین مدت زمان کمتری نسبت به بقیه به هدف رسیده است.

پس به طور کلی heuristic اول هم از نظر تعداد نود های ویزیت شده و هم از نظر زمانی از دیگر heuristic ها بهتر و heuristic سوم با اینکه اپتیمال بود ولی از بقیه بدتر بود. اما دلیل ان چیست؟ دلیل ان به خاطر این است که گرچه heuristic سوم optimal است و مقادیر تخمین زده شده در ان کمتر از مقدار واقعی است ، اما مقدار ان انقدر از واقیت دور است که لزوما در زمان خوبی به جواب بهینه نمی رسد. اما مثلا heuristic اول با اینکه اپتیمال نبود اما در بعضی مواقع مقادیر تخمین زده شده به واقعیت نزدیک بود و باعث می شد که در زمان بهتر و با دیدن نود های کمتری نسبت به بقیه به هدف برسد. در واقع تنها خوبی heuristic سوم نسبت به بقیه این بود که حداقل مطمعنیم در این heuristic ما بالاخره با جواب بهینه خواهیم رسید اما در زمان زیاد. ولی در سه heuristic دیگر که admissible نبودند صرفا به صورت اتفاقی در این تست کیس ها به جواب بهینه رسیدند و زمان بهتری نسبت به heuristic سوم ثبت کردند

سوال5)

در نمودار زیر می توانیم میانگین مدت زمان هر الگوریتم را ببینیم: (لازم به ذکر است برای پازل های 5*5 برای الگوریتم های BFS و IDS زمان 0 در نظر گرفته شده چرا که خواسته ی سوال نبودند)

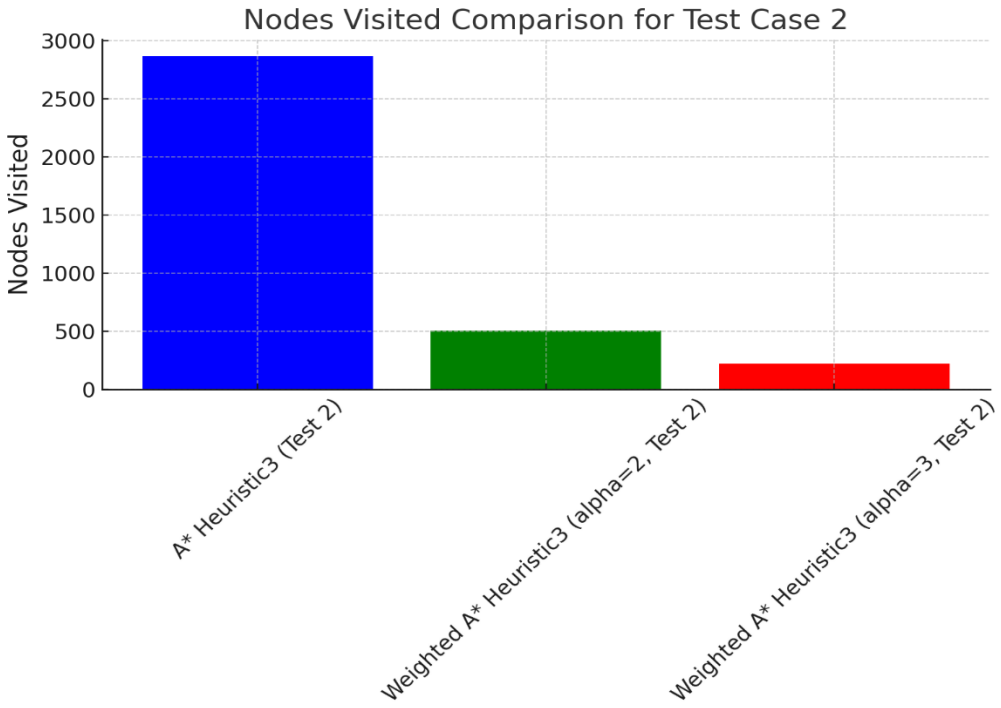


که به ترتیب الگوریتم های A* heuristic 1 و BFS و IDS میانگین کمترین زمان ها را داشته اند.

الگوریتم weighted A*: در این الگوریتم تنها باید یک وزن انتخاب کنیم که این وزن ها در مقدار تخمین زده شده ی هر تابع heuristic ضرب می شود. خوبی این الگوریتم این است که باعث می شود اگر مثلا heuristic ما admissible باشد ولی مثلا از مقدار واقعی خود خیلی دورتر و کمتر باشد باعث شود که به مقدار واقعی اش بعد از این که در یک عددی ضرب می شود نزدیک تر شود. فقط باید حواسمان باشد که این ضرب شدن باعث نشود که مقدار تخمین زده شده از مقدار واقعی فراتر رود.

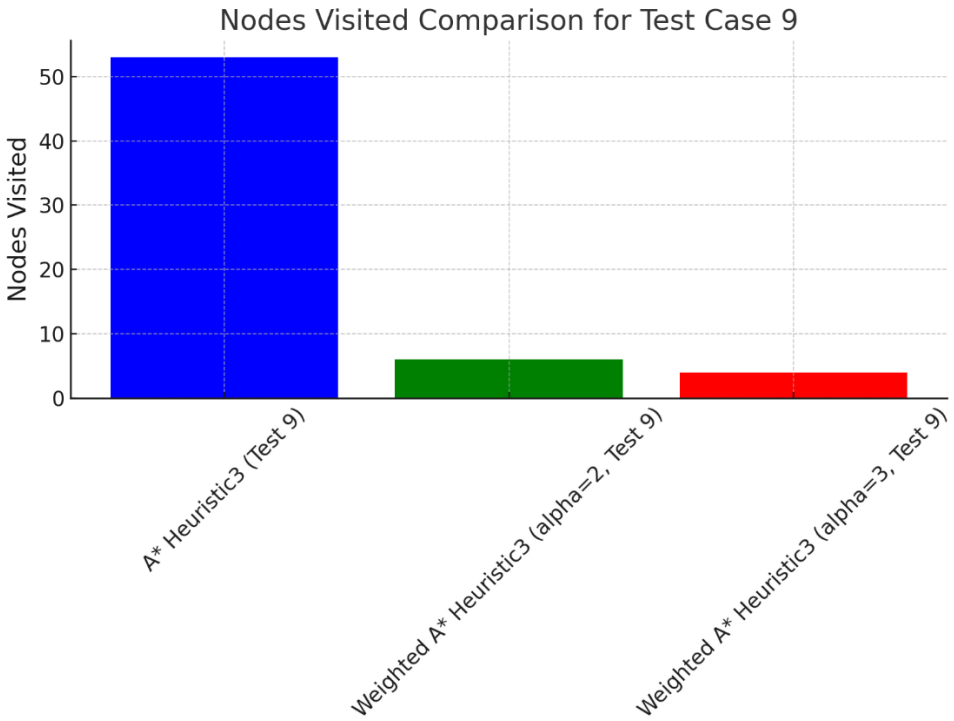
مثلا در مثال های قبل دیدیم که دلیل بد بودن heuristic سوم این بود که با وجود admissible بودن اما از مقدار واقعی بسیار کوچکتر بود. حال ما از weighted A* استفاده می کنیم و مقدار تخمین زده شده را یکبار در a = 2 و بار دیگر در a = 3 ضرب می کنیم. انتظار داریم که مقدار تخمین زده شده به مقدار واقعی نزدیکتر شود و الگوریتم با visit کردن نود های کمتر و در زمان کمتری به هدف برسد.

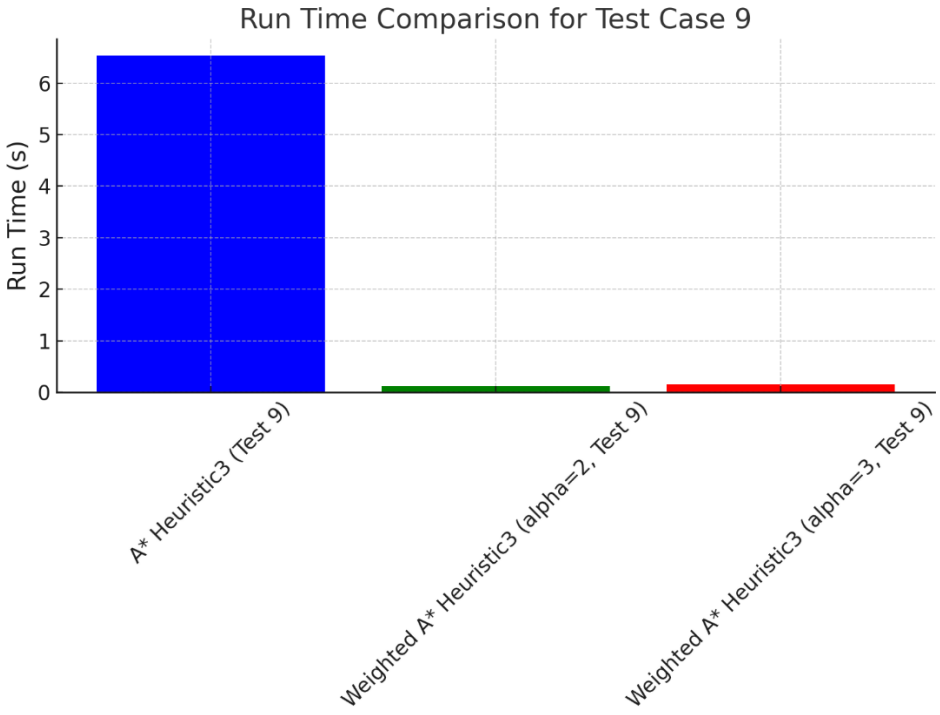
مثلا در تست کیس دوم این قابلیت بسیار مشهود است که A* در حالت عادی حدود 2800 نود ویزیت می کند در حالی که همین الگوریتم با a = 2 تنها با ویزیت کردن 500 نود به استیت هدف می رسد و همچنین با a = 3 این مقدار به زیر ۵۰۰ می رسد.



همچنین از نظر مدت زمانی نیز این تفاوت قابل شهود است که با زیاد شدن a مقادیر تخمین زده شده به واقعیت نزدیک تر شده و ما قدم های اینده را در مسیر بهتری بر میداریم و سریع تر به استیت هدف خود می رسیم.

همچنین می توانیم همین خصوصیت را در تست کیس نهم نیز ببینیم:





در اخر نیز تمام اطلاعات مربوط به تست کیس ها در فایل table.txt موجود است که می توانید انها را ببینید.