«گزارش کار پروژه اول AI »

برنا فروهری 810101480

سوال1-نحوه مدل کردن مسئله شامل تعریف action ، state goal، state و initial ... را به طور دقیق توضیح دهید.

State:

در این سوال state ما یک وضعیت از سالن و وضعیت خاموش و روشن بودن لامپ هایش میباشد که هر state یک پازل nدرn شامل اعداد 0 و 1 به معنی خاموش و روشن بودن لامپ ها میباشد.

Initial state :

پازلی که در ابتدا به عنوان آرگومان به تابع داده شده و الگوریتم ها روی آن اجرا میشود.

Goal state:

پازلی که در اخر به عنوان خروجی میدهد و همه خانه های آن باید عدد 0 به معنای خاموش بودن تمام لامپ ها باشد.

Action:

با هر بار فشردن کلید های پازل مختصات خود کلید ،سمت راست،چپ،بالا،پایین از 0 به 1 یا برعکس تغییر حالت میدهد.

Transition model:

پازل بعد از هربار toggle کردن به یک استیت دیگر میرود که پس از تغییر وضعیت آن و لامپ های مجاورش ایجاد شده است.

Path cost:

در این سوال هر toggle کردن یک هزینه ثابت دارد که آن را 1 در نظر میگیریم و هزینه ی مسیر ما برابر تعداد toggleها یعنی تعداد تغییر وضعیت هایی که در کل برای رسیدن به goal state اعمال می کنیم.

سوال2-هر یک از الگوریتم های پیاده سازی شده را توضیح دهید و تفاوتها و مزیت های هر یک نسبت به دیگری را قید کرده و عنوان کنید که کدام الگوریتم ها جواب بهینه تولید میکنند.

در ابتدا الگوریتم BFS:

```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]:
    curstate = deepcopy(puzzle)
    if (curstate.is_solved()):
        return [], 1
    bfslist = [curstate]
    explored = set()
    availablelights = curstate.get_moves()
    while(len(bfslist) > 0):
        matrix = bfslist.pop(0)
        explored.add(matrix)
        for item in availablelights:
            new = deepcopy(matrix)
            new.toggle(item[0], item[1])
            if str(new.board) not in explored not in bfslist:
                new.lights.append(tuple((item[0], item[1])))
                bfslist.append(new)
                if (new.is_solved()):
                    return new.lights, len(explored)

    return [], 0
```

0.0s Python

در این الگوریتم ابتدا با داشتن یک لیستbfslist و یک مجموعهexplored به ترتیب برای ذخیره استیت های فرزند و استیت های دیده شده تعریف میکنیم .
همچنین لیستی از دوتایی ها برای تمامی حالت های موجود در ماتریس و امکان حرکت در ان را در availablelights نگهداری میکنیم.

تا زمانی که bfslist خالی نشده و یا به goalstate نرسیدیم کار زیر را تکرار میکنیم.

ابتدا از سر صف پاپ کرده و در explored قرار میدهیم. به ازای تمامی حالات ممکن درماتریس برای تمامی انها یکبار toggle کرده و لامپ toggle شده را در خود همان state ذخیره میکنیم که بدانیم از زدن چه لامپ هایی ساخته شدند.

حال اگر این state عضو اعضای تکراری در explored و bfslist نبود انگاه به صف اضافه میکنیم که بچه های ان مورد بررسی قرار بگیرند. پاسخ نهایی بهینه است.

در الگوریتم 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]:
    curstate = deepcopy(puzzle)
    if (curstate.is_solved()):
        return [], 1
    d = 0
    final_explored = []
    availablelights = curstate.get_moves()
    while(True):
        dfslist = [curstate]
        explored = set()
        while(len(dfslist) > 0):
            matrix = dfslist.pop()
            explored.add(matrix)
            if matrix.depth <= d:
                final_explored.append(matrix)
            for item in availablelights:
                new = deepcopy(matrix)
                new.toggle(item[0], item[1])
                new.depth += 1
                if str(new.board) not in explored not in dfslist:
                    if new.depth <= d+1:
                        new.lights.append(tuple((item[0], item[1])))
                        dfslist.append(new)
                        if (new.is_solved()):
                            return new.lights, len(final_explored)
                    else:
                        break
        d += 1
```

در این الگوریتم ابتدا باید یک d به عنوان محدودیت بررسی تا آن لایه را مشخص کنیم و یک final_explored برای مشخص شدن نود های دیده شده از اول اول تا زمان رسیدن به goalstate .

تا زمانی که dfslist خالی نشده و یا به goalstate نرسیدیم کار زیر را تکرار میکنیم.

ابتدا برای هر بار بررسی یک dfslist و مجموعه ی explored و تا زمانی که عمق آن نود از محدودیت عمق مجاز بزرگتر نباشد آن را به

final_explored اضافه کرده و بررسی را ادامه میدهیم و به ازای تمامی حالات ممکن درماتریس برای تمامی انها یکبار toggle کرده و لامپ toggle شده را در خود همان state ذخیره میکنیم که بدانیم از زدن چه لامپ هایی ساخته شدند و باید متغیر depth آن نود که در کلاس اضافه کردیم را برابر عمق نود پدر +1 قرار دهیم.

حال اگر این state عضو اعضای تکراری در explored و bfslist نبود انگاه به صف اضافه میکنیم که بچه های ان مورد بررسی قرار بگیرند.

در انتها پس از بررسی کامل تا لایه محدود شده d را یکی زیاد کرده و دوباره به فرم DFS تا عمق جدید را بررسی میکنیم. پاسخ نهایی بهینه است.

در الگوریتم *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]:
    curstate = deepcopy(puzzle)
    if (curstate.is_solved()):
        return [], 1
    state_priority = 0
    astarlist = []
    heapq.heappush(astarlist,(heuristic(curstate), state_priority, curstate))
    explored = set()
    availablelights = curstate.get_moves()
    while(True):
        fn,prior,matrix = heapq.heappop(astarlist)
        explored.add(matrix)
        if (matrix.is_solved()):
            return matrix.lights, len(explored)
        for item in availablelights:
            new = deepcopy(matrix)
            new.toggle(item[0], item[1])
            new.fn = heuristic(new) + len(new.lights) + 1
            if str(new.board) not in explored:
                new.lights.append(tuple((item[0], item[1])))
                state_priority += 1
                heapq.heappush(astarlist, (new.fn,state_priority, new))
```

✓ 0.0s Python

در این الگوریتم تقریبا مشابه BFS عمل میکنیم با این تفاوت که باید بجای پاپ کردن از اول صف عضوی را پاپ کرد که f(n) ان کمینه باشد. f(n) هم که جمع h(n) و g(n) میباشد. h(n) را برای هر state با توجه به آرگومان داده شده میتوان یافت و g(n) عر state نیز برابر تعداد کلید های زده شده برای رسیدن به آن state میباشد که برای آنکار len(new.lights) را حساب میکنیم.برای پیاده سازی لیستی که بدانیم کدام nodeها در مرز های ما هستند از heapاستفاده میکنیم که بر اساس f(n) هر نود که کمترین باشد را برگرداند و در

صورت یکسان بودن f(n) ها ان نود که زود تر createشده را در راس قرار میدهد. بقیه عملیات ها مانند BFS و تفاوتش در انتخاب کدام نود برای ادامه بررسی مسیر بود.پاسخ نهایی بهینه است.

در الگوریتم *weighted_A: این الگوریتم نسخهای از الگوریتم A* است که در آن از یک وزنw برای تسریع فرایند استفاده میکنیم .تفاوت این الگوریتم با A* در این است که تابع heuristic را در یک ضریب ثابت ضرب میکنیم. در این صورت، تعداد نود های کمتری ملاقات می شوند اما ممکن است optimality پاسخ نهایی از بین برود و لذا پاسخ نهایی در این الگوریتم لزوما optimal نخواهد بود.اما با این کار تفاوت بین دو مقدار مختلف در heuristic بیشتر شده و فرایند جستجو سریعتر خواهد شد. ما بقی الگوریتم دقیقا مثل الگوریتم قبل میباشد و این بار معیار برگداندن heap کمینه بودن این f(n) جدید میباشد.

تمامی الگوریتم ها کامل هستند وبه جواب نهایی می رسند. با توجه به ویژگی این الگوریتم ها و به طور کلی weighted A* سریع تر از A* سریع تر از IDS برابر با BFS می باشد و فضای مورد نیاز در IDS کمتر از BFS کمتر از weighted A*برابر با A* می باشد.

-3

h(n) اول را برابر تعداد 1 ها یعنی تعداد لامپ های روشن در هر state تعریف میکنیم که این تابع admissible نیست. با مثال این را نشان میدهیم فرض کنید که در یک ماتریس 3*3 لامپ وسط دو ردیف اول و اخر و کل لامپ عای ردیف وسط روشن است h(n)این state برار 5 است اما هزینه واقعی آن برای خاموش کردن این لامپ ها 1 است زیرا تنها کافی است لامپ وسط ردیف دوم را خاموش کنیم و چون h(n) از هزینه واقعی بیشتر است پس قطعا admissible نیست. همچنین consistent هم نیست زیرا مثلا در حالت اول که 5 لامپ روشن است با خاموش کردن لامپ وسط ردیف دوم به حالت دوم که تمامی لامپ ها خاموش است میرسیم که h(n) 0 است و با هزینه 1 به ان رسیدیم پس رابطه ی h(A) < h(B) + cost(A to B) برقرار نیست.

h(n) دوم را برابر تعداد 1 ها یعنی تعداد لامپ های روشن در هر state تقسیم بر 3 تعریف میکنیم که این تابع admissible نیست. با مثال این را نشان میدهیم فرض کنید که در یک ماتریس 3*3 لامپ وسط دو ردیف اول و اخر و کل لامپ عای ردیف وسط روشن است h(n)این state برار 1.66 = 5/3 است اما هزینه واقعی آن برای خاموش کردن این لامپ ها 1 است زیرا تنها کافی است لامپ وسط ردیف دوم را خاموش کنیم و چون h(n) از هزینه واقعی بیشتر است پس قطعا admissible نیست. همچنین consistent هم نیست زیرا مثلا در حالت اول که 5 لامپ روشن است با خاموش کردن لامپ وسط ردیف دوم به حالت دوم که تمامی لامپ ها خاموش است میرسیم که h(n) 0 است و با هزینه 1 به ان رسیدیم پس رابطه ی h(A) < cost(A to B) + h(B) برقرار نیست.

h(n) سوم را برابر تعداد 1 ها یعنی تعداد لامپ های روشن در هر state تقسیم بر 5 تعریف می کنیم. این تابع هم consistent و هم admissible است. چون حداکثر 9 لامپ می توانند 1 باشند و h(A) حد اکثر برابر 1.8 = 9/5 می شود و در این حالت h(B) هم حد اقل برابر 0.8 = 4/5 است که در این صورت هم رابطه ی بالا درست است و consistent است و در نتیجه admissible هم هست.

4- در اجرای الگوریتم A* با توابع h(n)های متفاوت و مشاهده نتایج و مسیری که برای toggle کردن لامپ ها داده به نکته اصلی استفاده از تابع consistent پی میبریم و میبینیم که در تمامی تست ها مسیری که تابع consistent به ما معرفی میکند بهینه و کمینه می باشد یعنی کمترین تعداد لامپ را برای رسیدن به goalstate به ما نشان میدهد در صورتی که اگر تابع h(n) که تابع اول ما میباشد consistent نباشد فقط میتوان اطمینان داشت که به جواب میرسیم ولی لزومی بر اینکه بهینه باشد ندارد.

دقت شود که تابع h(n) اول ما که consistent و admissible نیست اما تخمین بسیار مناسبی برای این سوال میباشد به گونه ای که در تست ها مسیر بهینه را به ما نشان میدهد.

تحلیل بالا یک بررسی کلی از توابعی که consistent ومیباشند و یا نمیباشند بود.

5- برای به دست آوردن میانگین زمانی هر اجرا دقت شود که به ازای مواردی که در جدول نهایی پاسخ ها time limit خوردیم همان بیشینه زمان مجاز برای بررسی آن الگوریتم را که در صورت سوال تعیین شده قرار میدهیم.

برای BFS -> 40.800

برای IDS -> 60.852

برای *A : برای اولی -> 20.376 برای دومی -> 40.228
برای سومی -> 42.074

برای weighted A* با a=2: برای اولی -> 20.091 برای دومی -> 1.657
برای سومی -> 20.302

برای weighted A* با a = 5: برای اولی -> 20.083 برای دومی -> 40.110
برای سومی -> 20.489