گزارش پروژه اول هوش مصنوعی الهه حبیب زاده شجاعی 9931014 استاد جوانمردی

0)سوال:

مسائل جست و جو در واقع کلیت هدف ما را مشخص میکنند. مثلا رسیدن به x و y خروج از نقشه ی پک من یا خوردن همه ی غذا ها از جمله مسائل جست و جویی هستند که در پک من مطرح میشوند.

در درس دیدیم مسائل جست و جو یا search problems شامل موارد زیر هستند:

فضای حالت state space

تابع پسین با اقدامات و هزینه ها successor functions

حالت شروع و آزمون هدف

کلاس SearchProblems کلاسی است که برای یک مسئله جست و جو استفاده میشود و شامل توابعیست که حالت شروع، حالت هدف، تابع پسین و هزینه را برمیگردانند. با توجه به آبسترکت بودن این کلاس، این توابع مستقیما در آن تعریف نشده اند ، بلکه در کلاس هایی که از SearchProblems ارثبری میکنند این توابع را override میکنیم و بر اساس مسئله جست و جویمان آن ها را تعریف میکنیم.

getStartState : تابعیست که حالت شروع را برمیگرداند. مثلا در مساله رسیدن به نقطه خروجی ایکس و ایگرگ شروع میشود حالت شروع.

getSuccessors : successor function تابعیست که حالت بعدی را برمیگرداند. تابع getSuccessors که در کلاسSearchProblems قرار دارد، لیستی از ساکسسور ها (حالت های بعدی ممکن) به همراه هزینه لازم و اقدام لازم برای رسیدن به آنها را برمیگرداند.

getCostOfActions : لیستی از action ها را به عنوان ورودی میگیرد و در نهایت هزینه انجام کل این لیست را برمیگرداند.

isGoalState : برای اینکه مساله تمام شود باید به هدف نهایی رسیده باشیم، بنابراین بعد از رفتن به هر حالت کنترل میکنیم ایا به حالت هدفمان رسیده ایم یا نه و اینکار توسط این تابع انجام میشود.

کلاس های game.py

Agent : index برای شمردن حرکاتیست که طی شده . در متد getAction حرکت بعدی را با استفاده از state فعلی برمیگرداند(در کلاس هایی که از Agent ارثبری میکنند این متد overrite میشود). registerInitialState نیز در صورت نیاز نوشته میشود و در آن قبل از شروع حرکت عامل، مسیر حرکت و ... حاصل شده و ذخیره میشوند.

Directions : 4 جهت جغرافیایی و stop را دارد. سه دیکشنری دارد که با آنها به آسانی میتوان جهت عامل پس از چرخش به راست ، چرخش به چپ یا حرکت به جهت مخالف را به دست آورد. (به وضوح اگر با چرخش به راست از x به y تغییر کنیم با چرخش به چپ از y به x برمیگردیم پس برای دیکشنری چرخش به چپ فقط کافیست دیکشنری چرخش به راست را برعکس کنیم.)

Configuration: فیلد های مختصات و جهت حرکت یک عامل را دارد و متد هایی دارد که آنهارا برگرداند. متدی برای بررسی صحیح بودن x و y نیز دارد. متد های hash و str نیز override شده اند. generateSuccessor بردار حرکت دریافت میکند. جهت بردار را مشخص کرده و بعد کانفیگوریشن جدید را با حرکت در جهت و به اندازه بردار برمیگرداند.

AgentState : فیلد های state یک agent را در خود نگه میدارد که شامل کانفیگوریشن ان عامل، سرعت حرکتش و ترسیده بودنش، امتیاز کسب شده و ... میشود.توابعی نیز برای برگرداندن جهت فعلی عامل و موقعیت فعلیش دارد.

Grid : این کلاس یک آرایه 2 بعدی است که خانه های آن مطابق خانه های صفحه بازیست. از این کلاس میتوان برای نگهداری دیوار ها یا غذا ها یا موقعیت روح ها و ... استفاده کرد. خانه ها از جنس بولین هستند. متد هایی برای بازگردانی به شکل لیست، (تغییر محتوای کپی، اینستنس اصلی را نیز عوض میکند)deep copy و shallow copy و تبدیل ایندکس یک خانه به موقعیتش و ... را نیز دارد.

پیاده سازی:

Priority queue درواقع شامل یک min heap است. میدانیم بعد از update یک node در هیپ، نیاز به هیپیفای کردن داریم تا ویژگی های هیپ برقرار بمانند. طبق توضیحات اگر node ای که میخواستیم update کنیم در لیستمان نبود، آن را در heap اضافه میکنیم (متد از پیش نوشته شده ی heappush). اگر node در لیست بود اما اولویت آن بیشتر از مقدار جدیدش بود، مقدار جدید را به اولیت آن میدهیم. سپس لیست را heapify میکنیم. اگر اولویت کمتر یا مساوی بود نیاز نیست کاری کنیم.

```
def update(self, item, priority):
    # If item already in priority queue with higher priority, update its priority and rebuild the heap.
    # If item already in priority queue with equal or lower priority, do nothing.
    # If item not in priority queue, do the same thing as self.push.
    heap_list = list(map(list, self.heap))
    not_in_heap = True
    for k in heap_list:
        if k[2] == item:
            not_in_heap = False
            if k[0] > priority:
                k[0] = priority
                heap_list = list(map(tuple, heap_list))
                heapq.heapify(heap_list)
                self.heap = heap_list
            break
    if not_in_heap:
        heapq.heappush(self.heap, (priority, self.count, item))
```

1)

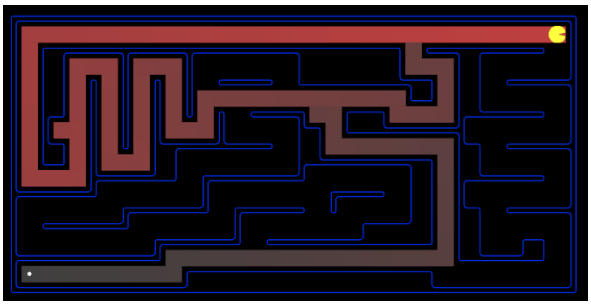
پیاده سازی:

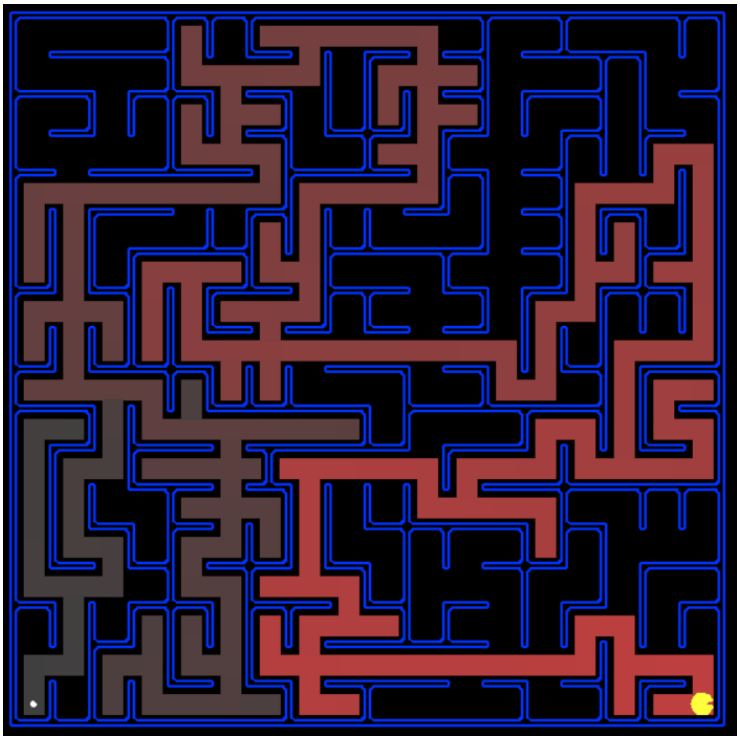
جست و جوی اول عمق همانطور که از اسمش مشخص است، یک گره را آنقدر expand میکند که به عمیق ترین قسمت آن برسد (درواقع عمیق ترین گره را expand میکند). در صورتی که به هدف نرسیده باشد مسیر را به عقب برمیگردد و عمیق ترین گره بعدی را ادامه میدهد. با توجه به توضیح داده شده، برای پیاده سازی آن از ساختمان داده ی پشته استفاده میکنیم. نود ها از استک pop میشوند و بعد expand میشوند. با توجه به خواص استک، آخرین item وارد شده در آن، اولین item ایست که ازش خارج میشود پس آخرین node پوش شده در استک که به تبع عمیق ترین node است، اولین نودی است که برای expand شدن خارج میشود.

Visited شامل استیت هایی است که قبلا از آنها عبور شده و اکسپلور شده اند.

Fringe همان استکیست که گره ها را برایexpand شدن در آن نگهداری میکنیم. استیت فعلی همان node ایست که از fringe پاپ میشود. تابع getSuccessors را فراخوانی میکنیم تا حالات ممکن بعدی را برگرداند. از بین حالات برگردانده شده آنهایی را در fringe پوش میکنیم که explore نشده باشند. این زمان است که درواقع node ها را تولید کرده ایم و چک کردن هدف در زمان تولید انجام میشود پس چک میکنیم اگر هدف بودند flag را falseمیکنیم تا جست و جو تمام شود. ضمنا برای هر استیتی که در fringe پوش میشود مسیر رسیدن به آن را نیز پوش میکنیم تا در نهایت وقتی به استیت هدف رسیدیم بتوانیم مسیر رسیدن به آن را نیز داشته باشیم.

تعداد گره های گسترش یافته شده	هزینه کل	maze
390	210	Big
146	130	Medium
15	10	tiny



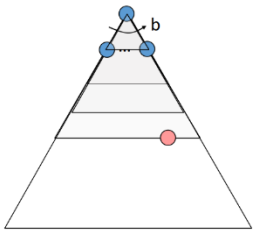


همانطور که در تصاویر مشخص است، در روش جست و جوی dfs استیت آنقدر گسترش میابید تا به بن بست برسد و بعد برمیگردد عقب تا مسیر جدیدی را گسترش دهد و رنگ های مشاهده شده به همین دلیل است و هرچه قرمز ترند یعنی زودتر گسترش یافته اند.

سوال:

الگوریتم IDS

Iterative Deepening Search الگوریتمی است که درواقع از bfs نیز بهره میگیرد. به این شکل که تا مثلا عمق ده نود به صورت dfs پیش میرویم ، بعد به جای اینکه دوباره عمیق ترین نود را گسترش دهیم، برمیگردیم عقب و به dfs در نود های تا عمق 10 ادامه میدهیم. نکته قابل توجه اینست نود های بالایی چندین بار سرچ میشوند.



مشابه چیزی که در شکل است، به طور سطح سطح عمق جست و جوی dfs را زیادتر میکنیم.

شبه کد کلی این الگوریتم به شکل زیر است

شبه کد

```
IDS(Graph , Limit){
        flag = false
        While flag is false do {
                flag , goal = Limited_dfs(graph , limit)
                limit = limit + limit
        }
        Print(goal)
}
limited_dfs(graph , limit){
        stack
        stack.push(graph.root)
        while stack is not empty do{
                s = stack.pop()
                if s is not visited and s.level is lower or equal to limit{
                        get children of s
                        for c in s.children{
                                if c is goal{
                                        return (true, c)
        }
                                push c in stack
                        }
                }
        }
        return (false,none)
}
```

نکته:

برای تبدیل الگوریتم dfs به ids اگر در هر نود بدانیم در چه عمقی نسبت به root هستیم کار سختی نداریم. در dfs ای که ما برای مساله جست و جو پیاده سازی کردیم، هر نود مسیری که توسط آن از نقطه شروع به آن رسیده بودیم را نیز در خود داشت. پس فقط کافیست جز چک کردن visited نبودن فرزند، چک کنیم طول مسیر رسیدن به آن فرزند از limit کمتر باشد (طول لیستی از حرکات است) و بعد آن را در fringe پوش کنیم. ضمنا در صورتی که دیدم فرینج خالی شده یعنی همه گره های با عمق کمتر از limit اکسپلور شدند و هدف نبودند پس به لیمیت اضافه میکنیم و ids را از اول اجرا میکنیم.

در بخش هشت ids پیاده سازی شده

2)

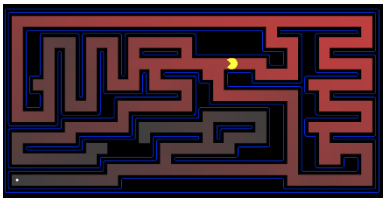
پیاده سازی:

جست و جوی اول سطح را میتوانیم مانند جست و جوی اول عمق پیاده کنیم و صرفا تغییراتی در اولویت بندی fringe بدهیم. در bfs اولویت pop شدن گره های fringe به گونه ایست که گره ای که اول وارد شده اول هم باید خارج شود (fifo) (اما در dfs به شکل استک بود). اعمال این تغییرات جزئی، کد ما را به bfs تبدیل میکند. الگوریتم bfs درواقع سطح به سطح گره ها را میگردد. این الگوریتم بهینه است و پاسخ با کمترین هزینه را برمیگرداند.

Medium maze

Cost = 68

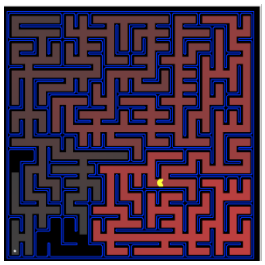
Expanded = 269



big maze

cost = 210

expanded=620



همانطور که از مقایسه با قسمت قبل معلوم است ، bfs نسبت به dfs گره های بیشتری گسترش میدهد چون سطح به سطح همه ی گره ها را بررسی میکند اما از طرفی چون بهینه است همیشه path با کمترین cost را پیدا میکند.

برای سنجش صحت الگوریتمbfs توانایی آن را در حل کردن مساله 8 پازل میسنجیم.

مساله 8 پازل به این شکل است که صفحه ای 3 در 3 داریم که در آن کاشی هایی با قابلیت بالا و پایین و چپ و راست رفتن هست و از 1 تا 8 شماره گذاری شده اند.

حال باید به ازای حالتی که آنها قرار دارند، لیستی از حرکات یافت شود که توسط آنها کاشی ها به ترتیب قرار بگیرند.

استیت در این مساله لیستی از ترتیب قرار گیری کاشی هاست مثلا استیت شروع در عکس روبه رو[1,0,5,3,2,4,6,7,8] است (جای خالی 0 قرار میگیرد).

توسط الگوریتم bfs کم هزینه ترین روشی پیدا میشود که هر ترتیبی به [0,1,2,3,4,5,6,7,8] برسد.

```
A random puzzle:
-------------
|  1  |     |  5  |
-------------
|  3  |  2  |  4  |
-------------
|  6  |  7  |  8  |
-------------
BFS found a path of 5 moves: ['down', 'right', 'up', 'left', 'left']
After 1 move: down
-------------
|  1  |  2  |  5  |
-------------
|  3  |     |  4  |
-------------
|  6  |  7  |  8  |
-------------
Press return for the next state...
After 2 moves: right
-------------
|  1  |  2  |  5  |
-------------
|  3  |  4  |     |
-------------
|  6  |  7  |  8  |
-------------
Press return for the next state...
After 3 moves: up
-------------
|  1  |  2  |     |
-------------
|  3  |  4  |  5  |
-------------
|  6  |  7  |  8  |
-------------
After 4 moves: left
-------------
|  1  |     |  2  |
-------------
|  3  |  4  |  5  |
-------------
|  6  |  7  |  8  |
-------------
Press return for the next state...
After 5 moves: left
-------------
|     |  1  |  2  |
-------------
|  3  |  4  |  5  |
-------------
|  6  |  7  |  8  |
-------------
```

سوال:

BBFS مخفف Bi-directional BFS است. در این الگوریتم دو bfs اجرا میشود، یکبار از استیت اولیه برای یافتن استیت هدف و یکبار هم از استیت هدف برای یافتن استیت اولیه. در حین اینکه این دو bfs اجرا میشوند، اگر جایی یک گره همسان را explore کنند الگوریتم پایان میابد و مسیر یافت میشود. bbfs بهینه است چون دو bfs ای که اجرا میشوند کوتاه ترین مسیر هارا تا نقطه مشترک به دست میاورند و جمع کوتاه ترین مسیر ها تا یک نقطه مشترک همان کوتاه ترین مسیر بین هدف و استیت اولیه است.

شبه کد زیر برای درک بهتر الگوریتم نوشته شده است:

در این شبه کد q1 و q2 به عنوان فرینج های دو bfs در نظر گرفته شده. تا زمانی که فرینج ها خالی نشده باشند هر بار یکی از فرینج q1 پاپ میکنیم. چک میکنیم استیت هدف نباشد یا در فرینج 2 نباشد چون اگر در فرینج دو باشد یعنی به گره مشترک دو bfs رسیده ایم و وقت اینست که مسیر را برگردانیم. برگرداندن مسیر به این شکل است که مسیر در bfs از استیت شروع به نقطه اشتراک را با معکوس مسیر از هدف به نقطه اشتراک جمع کرده و برمیگردانیم.

اگر نه هدف بود و نه اشتراک با فرینج 2 ، صرفا ساکسسور ها را اضافه میکنیم.

به طور مشابه همین کار ها را برای bfs دوم نیز انجام میدهیم.

```
def BBFS(problem):
    q1 is Queue
    q2 is Queue
    explored1 is set_of_explored_nodes
    explored2 is set_of_explored_nodes
    q1.push((problem.getStartState(), []))
    q2.push((problem.goal, []))
    while not (q1.isEmpty() or q2.isEmpty()):
        if q1 is not empty:
            curr_state = q1.pop()
            curr_actions = curr_state.actions
            if curr_state not in explored1:
                explored1.add(curr_state)
                if curr_stat is GoalState or curr_state is in q2:
                        return curr_actions + reverse(q2_actions_upto_intersected_node)
                add_successors_of_curr_state()
        if q2 is not empty:
            curr_state = q2.pop()
            curr_actions = curr_state.actions
            if curr_state not in explored2:
                explored2.add(curr_state)
                if curr_stat is startState or curr_state is in q1:
                    return  q1_actions_upto_intersected_node + reverse(curr_actions)
                add_successors_of_curr_state()
```

چند هدف بودن:

اگر چندین هدف وجود داشته باشند و ما مساله مان این باشد که مسیری به نزدیکترین هدف را پیدا کنیم میتوانیم از روش زیر استفاده کنیم. یک گره ی فرضی به گراف جست و جو اضافه میکنیم. همه ی اهداف ، یالی مستقیم به این گره دارند(هزینه ها یکسان). حال BBFS را بین این استیت جدید و استیت اولیه اجرا میکنیم. مسیری حاصل میشود که نقطه شروع را به هدف فرضی وصل کند. تمام یالهای وارد شده بر هدف فرضی از اهداف واقعی بودند پس قطعا در مسیری که بدست آمده از یکی از این اهداف گذر کرده ایم. در bbfs گفتیم کوتاه ترین مسیر بین هدف و شروع به دست می اید پس مسیر حاصل شده کمترین هزینه را دارد و با توجه به اینکه ابتدا مشخص کردیم همه ی یال های وارد شده بر استیت فرضی هم هزینه اند یعنی هدف با کمترین هزینه در مسیر قرار دارد.

3) پیاده سازی:

UCS : Uniform Cost Search

در این روش از جست و جو گره هایی که کمترین هزینه را دارند گسترش میدهیم (به این علت که با کمترین هزینه به هدف برسیم) . به همین دلیل نیاز داریم در هر مرحله، گره با کمترین هزینه در fringe را پیدا کنیم و آنرا گسترش دهیم. این الگوریتم هم complete است و هم optimal یعنی هم حتما به هدف میرسد و هم روش بهینه ی رسیدن به هدف را بدست میاورد.

در بخش صفر PriorityQueue را دیدیم و بررسی کردیم که این ساختمان داده ، گره با کمترین اولویت را در o(1) بر میگرداند . در واقع درون این ساختمان داده از یک min heap استفاده شده . حال ما fringe را از نوع همین PriorityQueue قرار میدهیم و اولیت گره ها را هزینه آن قرار میدهیم در نتیجه گره با کمترین اولویت همان گره با کمترین هزینه است و میتوانیم در زمان o(1) به آن دسترسی داشته باشیم. (دقت شود منظور از هزینه ، مجموع هزینه ی حرکت هاییست که تا رسیدن به آن گره انجام شده)

در پیاده سازی این جست و جو ، در هر دور از fringe گره با کمترین اولویت(هزینه) را pop میکنیم. سپس اگر گره قبلا ویزیت نشده بود آنرا گسترش میدهیم. ابتدا کنترل میکنیم استیت هدف نباشد(اگر بود کار تمام است و کافیست مسیر رسیدن به آن گره را برگردانیم). سپس ساکسور های آن گره را بدست میاوریم. ساکسور ها را به شکل (location,path,cost) در فرینج اضافه میکنیم. Path همان مسیر تا استیت فعلی است که با حرکتی که برای رسیدن به آن ساکسور لازم است جمع شده. Cost شامل هزینه لازم برای رسیدن به ساکسور و هزینه ی رسیدن به استیت فعلی است.

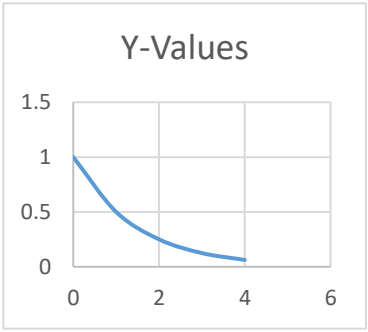
در نهایت وقتی به استیتی رسیدیم که هدف بود از حلقه بیرون میایم و مسیر را که یک string گرفته بودیم به لیست تبدیل کرده و برمیگردانیم.

گفته شده agent هایی داریم که تلاش میکنند بیشتر در شرق maze یا غرب آن باشند. این agent ها تابع هزینه های بخصوصی دارند.

StayEastSearchAgent :

این agent متمایل به گزینش state هایی باید باشد که شرق ترند. تابع هزینه برای آن x ^ ½ در نظر گرفته شده.

تابع هزینه آن رشد نمایی دارد، اما با توجه به اینکه هر دور در ½ ضرب میشود مقدارش کم و کمتر میشود.

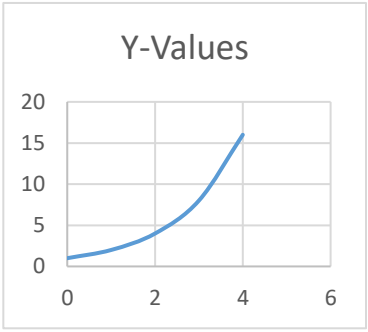


توضیح:

سمت شرق بودن در maze باعث میشود x مقدار بیشتری باشد(0 مختصات در غرب است) پس ما نیاز داریم گره هایی با x بیشتر ، هزینه کمتر داشته باشند تا در الگوریتم ucs زودتر explore شوند. همانطور که در نمودار این تابع مشخص است، x های با مقدار بیشتر ، هزینه ی کمتری دارند. در نتیجه در هر استیتی باشیم، آمدن به غرب هزینه بیشتری از بالا یا پایین یا راست رفتن دارد .

StayWestSearchAgent :

این agent متمایل به گزینش state هایی باید باشد که غرب ترند. تابع هزینه برای آن 2^x در نظر گرفته شده. تابع هزینه آن رشد نمایی دارد



توضیح:

سمت غرب بودن در maze باعث میشود x مقدار کمتری باشد(0 مختصات در غرب است) پس ما نیاز داریم گره هایی با x کمتر ، هزینه کمتر داشته باشند تا در الگوریتم ucs زودتر explore شوند. همانطور که در نمودار این تابع مشخص است، x های با مقدار کمتر ، هزینه ی کمتری دارند. در نتیجه در هر استیتی باشیم، آمدن به غرب هزینه کمتری از بالا یا پایین یا راست رفتن دارد .

سوال:

اگر هزینه همه حرکات را 1 (یا یک عدد ثابت مثبت) بگیریم میتوان گفت همان BFS است. در BFS ابتدا گره های با کمترین عمق را گسترش میدهیم و بعد به عمق بعدی میرویم. اگر تابع هزینه عددی ثابت باشد مطمین هستیم در هر گسترش ، گره های بعدی که دقیقا یک مرحله عمیقترند، هزینه یکسانی خواهند داشت و در نتیجه هم عمق و هم اولویت خواهند بود. با توجه به اینکه الگوریتمی که پیاده کردیم fringe از نوع PriorityQueue بود، هر بار گره های با کمترین عمق (عمق در اینجا با هزینه یکی است چون به ازای هر مرحله عمیق شدن 1 واحد به هزینه اضافه میشود) را برمیگرداند که مطابق با BFS است. برای تبدیل ucs به BFS کافیست در مرحله ای که هزینه ساکسسور ها را محاسبه میکنیم، هزینه رسیدن به استیت فعلی را فقط با 1 (یا عددی ثابت مثبت) جمع کنیم.

به طور کلی نمیتوان صرفا با تغییر تابع هزینه به dfs رسید اما میتوان با تغییرات زیر در ucs به dfs رسید.

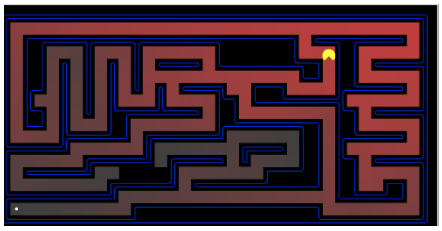
در dfs ما نیاز داریم در هر مرحله ، گره با عمیق ترین موقعیت را گسترش دهیم. در روش اول میتوان هزینه استیت اولیه را بیشترین مقدار ممکن(اگر فرض کنیم تمام گره ها بتوانند طی شوند میشود تعداد خانه های maze) قرار دهیم و در هر دور حین افزودن ساکسسور ها، هزینه استیت جدیدی که به fringe اضافه میکنیم را از کم کردن 1 از استیت فعلی به دست اوریم در نتیجه گره های عمیق تر هزینه کمتری خواهند داشت پس در صف اولویت زودتر برمیگردند.

حالت دوم اینست که هزینه ها را باز هم عددی ثابت بگیریم و fringe را یک max heap در نظر بگیریم در نتیجه هر دور گره با بیشترین هزینه در o(1) برگردانده میشود. هر مرحله هزینه ای که برای رسیدن به ساکسسور ها در نظر میگیریم را از جمع هزینه استیت فعلی با یک عدد ثابت مثبت به دست میاوریم در نتیجه عمیق ترین گره بیشترین هزینه را خواهد داشت و در pop کردن از fringeبرگردانده میشود.

medium maze

total cost = 68

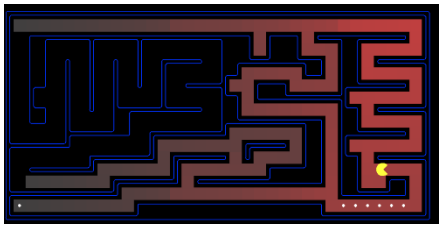
nodes explored = 269



medium dotted maze (stay east agent)

total cost = 1

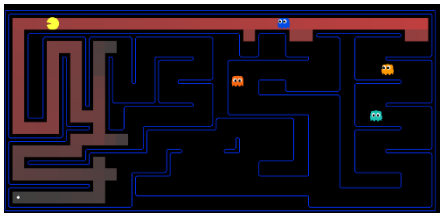
nodes explored = 186



medium scary maze (stay west agent)

total cost = 68719479864

nodes explored = 108



– دقت شود با توجه به توضیحات راجه به تابع هزینه در عامل های غرب یا شرق رونده ، هزینه در تابع غرب رونده در اوردر 2^x است و بسیار زیاد است و هزینه در تابع شرق رونده از اوردر ½^x است و بسیار کم است (انقدر کم بوده که جمع تقریبی آن 1 شده)

4) پیاده سازی:

در جست و جوی ucs ما صرفا با نگاه به هزینه های گذشته اولویت بندی میکنیم. بهبود یافته ی ucs روش a استار است . در a استار ، علاوه بر نگاه به هزینه های گذشته، از تابعی تحت عنوان هیوریستیک استفاده میشود تا توسط آن میزان نزدیکی به هدف(آینده نگری) نیز سنجیده شود. جمع هزینه و هیوریستیک ، اولویتی است که طبق آن در الگوریتم a استار از فرینج گره ها را explore میکنیم.

هیوریستیک منهتن به این شکل به دست می اید. اگر استیت فعلی و استیت هدف را دو راس رو به روی مستطیلی در نظر بگیریم که اضلاعش موازی محور ایکس و ایگرگ اند، جمع طول و عرض این مستطیل همان فاصله منهتنی است. میتوانیم به آسانی هیوریستیک فاصله منهتن را از جمع ایکس و ایگرگ استیت منهای x و y در استیت هدف به دست آوریم.

هیوریستیک اقلیدسی در واقع همان فاصله اقلیدسی دو نقطه یا کوتاه ترین فاصله بین دو نقطه است. طبق توضیح منهتن ، هیوریستیک اقلیدسی همان قطر مستطیل ذکر شده است. برای به دست آوردن طول این قطر از رابطه فیثاغورث برای طول و عرض مستطیل استفاده میشود. در مسائل ما goal state (x',y') است پس هیوریستیک اقلیدسی در استیت (x,y) را میتوانیم به آسانی با رابطه $((x-x')^2 + (y-y')^2)^{1/2}$ به دست آوریم.

پیاده سازی الگوریتم a استار مانند ucs است صرفا با این تفاوت که هنگامی که state های جدید را میخواهیم به فرینج پوش کنیم، اولویت (تابع f) را جمع هیوریستیک آن استیت با هزینه ی رسیدن به آن استیت قرار میدهیم. $f(s) = h(s) + c(s)$

سوال:

همه ی الگوریتم های BFS,DFS,UCS,A* را روی OpenMaze اجرا کردیم و تفاوت ها به شرح زیر است.

UCS => COST = 54 , EXPANDED = 682 , score = 456

BFS => COST=54 , EXPNDED = 682 , score = 456

DFS => COST=298, EXPANDED = 586 , score = 212

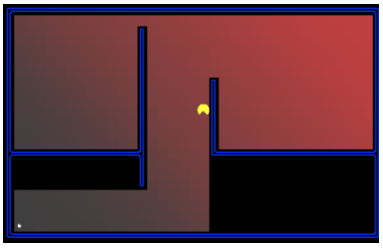
A* => COST=54, EXPANDED = 535 , score = 456

Agent هر چهار الگوریتم حالت عادی دارد. Cost در اینجا با طول مسیر برابر است چون هزینه هر حرکت را 1 در نظر گرفته ایم.

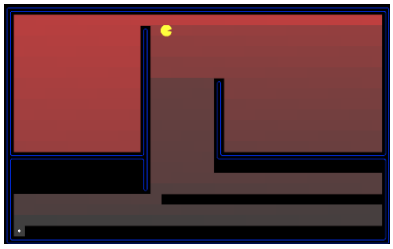
با توجه به اینکه گفتیم ucs با تابع هزینه ی ثابت و مثبت، همانBFS است انتظار میرفت نتایج آنها مانند هم باشد. هر دو راه حل بهینه را پیدا کرده اند و تعداد یکسانی گره را گسترش داده اند.

الگوریتم DFS صرفا عمیق ترین گره اش را گسترش داده، به طور تصادفی شکل maze باعث شده گره های کمتری را نیاز باشد گسترش دهد. هزینه (طول مسیر) بسیار بیشتر از حالت بهینه است چون گفتیم dfs بهینگی را تضمین نمیکند. همچنین مسیری که dfs طی میکند بسیار پیچ در پیچ و غیر بهینه است بنابراین امتیاز منفی بیشتری دارد و در کل امتیاز کمتری نصیبمان میشود.

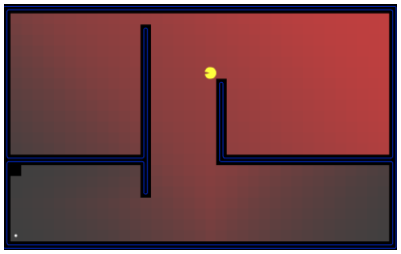
در این الگوریتم A* از هیورستیک فاصله منهتن استفاده شده، همان طور که مشاهده میشود استفاده از این هیوریستیک باعث شده برخی گره ها گسترش نیابند(دور از هدف) بنابراین تعداد گره های explored شده کمتر از بقیه است. هیوریستیک این الگوریتم admissible است چون فاصله منهتن دو خانه در maze قطعا کمتر یا مساوی طول مسیر واقعی بین آنهاست (هزینه در طی کردن هر خانه 1 است و صرفا به 4 جهات جغرافیایی میتوانیم برویم). با داشتن هیوریستیک admissible میتوانیم به بهینگی نتیجه الگوریتم A* اطمینان داشته باشیم بنابراین طول مسیرش 54 و برابر با bfsو ucs است. (البته اگر از هیوریستیک فاصله اقلیدسی استفاده میکردیم تعداد گره های explore شده 665 میشد پس تاثیر هیوریستیک های متفاوت قطعا یکسان نیست و هیوریستیکی که به واقعیت نزدیکتر باشد بهتر است.)



A*



dfs



ucs,bfs

5) پیاده سازی:

هدف ما در این مساله پیمایش هر چهار گوشه است بنابراین در هر استیت نیاز داریم بدانیم چه گوشه هایی تا الان پیمایش شده اند تا استراتژی خودمان در ادامه مسیر را بتوانیم مشخص کنیم. به همین منظور در هر استیت علاوه بر مختصات ، لیست مختصات گوشه های پیمایش شده را نیز نگه میداریم.

استیت اولیه توسط تابع getStartState برگردانده میشود. شامل startPosition یا مختصات اولیه و مجموعه ای تهی است (در صورتی که نقطه شروع روی یک گوشه نباشد در غیر اینصورت مختصات خود نقطه شروع (که یک گوشه است) به مجموعه اضافه میشود)

سنجش اینکه به استیت هدف رسیده ایم یا نه توسط تابعisGoalState به دست می آید. ورودی تابع state است. هدف پیمایش 4 گوشه بود پس کافیست مجموعه ی گوشه های پیمایش شده (که در state قرار دارد) تعداد اعضایش 4 باشد (maze ها مستطیلی اند پس 4 گوشه دارند). اگر تعداد اعضای مجموعه گوشه های پیمایش شده 4 بود یعنی ما هدفمان را به دست آورده ایم.

ساکسسور های یک state توسط تابع getSuccessors برگردانده میشوند. در این تابع حرکت به 4 جهت جغرافیایی چک میشود. اگر مجاز بودند (دیوار نبود) چک میشود که آیا گوشه اند یا نه. اگر گوشه بودند به مجموعه ی گوشه های پیمایش شده شان اضافه میشوند.(هزینه را به طور پیشفرض 1 قرار داده ایم.)

tinyCorners/dfs => cost = 48 , eplored = 72

mediumCorners/dfs => cost =141 , explored=411

استفاده از dfs بهینگی را تضمین نمیکرد. حال اگر از الگوریتم bfs استفاده کنیم به اعداد بهینه خواهیم رسید اما تعداد گره های explore شده زیاد خواهد بود.

tinyCorners/bfs => cost = 28 , eplored = 249

mediumCorners/bfs => cost =106 , explored=1937

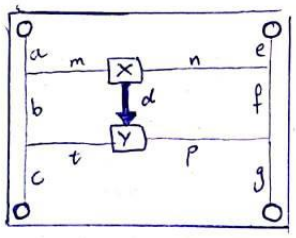
هدف اینست در قسمتهای بعد هیوریستیکی برای a* نوشته شود تا مانند bfs طول مسیر بهینه را بدهد و همچنین گره های کمتری را گسترش بدهد تا سریعتر به جواب برسیم.

6)

دو هیوریستیک برای مساله پیمایش هر چهار گوشه انتخاب شدند که به شرح زیرند:

هیوریستیک اول: 2 / (جمع فاصله تا گوشه های باقی مانده)

جمع فاصله ی منهتن استیت فعلی از گوشه های باقی مانده = f(state)



فرض کنید هر ۴ گوشه برای ویزیت شدن باقی مانده اند و از X به Y رفته‌ایم .
طی این حرکت d خانه طی شده و چون هر جابجایی هزینه ۱ دارد ، پس هزینه واقعی رفتن از X به Y ، d است .

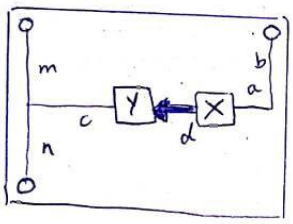
$f(x) = (m+a) + (n+e) + (f+g+n) + (m+b+c)$

$f(y) = (t+c) + (t+b+a) + (p+g) + (f+e+p)$

با توجه به سوازی بودن خطوط، $p=n$ و $m=t$ است.

$f(x) = 2m+2n+a+b+c+e+f+g$
$f(y) = 2p+2t+a+b+c+e+f+g$
$\Rightarrow f(x) = f(y) \longrightarrow f(x) - f(y) = 0$

فرض کنید ۳ گوشه باقی مانده و از X به Y رفته‌ایم .

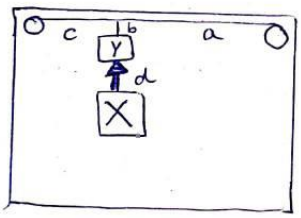


$f(x) = (d+c+n) + (d+c+m) + (a+b) = 2d+2c+m+n+a+b$

$f(y) = (c+n) + (c+m) + (d+a+b) = d+2c+m+n+a+b$

$f(x) - f(y) = d$

فرض کنید ۲ گوشه باقی مانده و از X به Y رفته‌ایم



« حالت روبه رو بیشترین تغییر f را می دهد ، حرکت افقی تغییر f را نمی کند ، اگر ۲ گوشه روی قطر مربع هم باشند تغییر f بازه می شود »

$f(x) = d+b+a + d+b+c$
$f(y) = b+a + b+c$
$\Rightarrow f(x) - f(y) = 2d$

هزینه واقعی در حرکت با d بود. با تابع f ، تفاضل f در رفتن از X به Y بین ۰ ، d و ۲d بود. حال اگر این f را تقسیم بر ۲ کنیم ، تفاضل ها به ۰ ، $\frac{d}{2}$ و d تقلیل می‌یابند . f/2 می‌تواند هیوریستیکی سازگار باشد چون $\frac{f(x)}{2} - \frac{f(y)}{2}$ از هزینه رفتن از X به Y که d است کمتر یا مساوی است.

هنگامیکه یک گوشه باقی مانده باشد فاصله مان دقیقا d واحد تغییر میکند.

پس هیوریستیک را اینگونه به دست میاوریم که از هزینه واقعی کمتر باشد. h(x) = f(x)/2

ثابت میشود هر هیوریستیک سازگار، قابل قبول هم هست. ثابت کردیم هیوریستیک بالا سازگار است پس حتما قابل قبول نیز هست.

با این هیوریستیک 1256 گره explore میشوند.

هیوریستیک دوم:

فاصله منهتن تا دورترین گوشه.

این هیوریستیک در استیت های متوالی مقدارش بین –1 تا 1 واحد میتواند تغییر کند(به دورترین گوشه نزدیک شویم یک واحد، یا دور شویم یک واحد یا با حرکتمان دورترین گوشه عوض شود و فاصله مان با گوشه جدید با فاصله با گوشه قبلی تغییر نکند) هزینه واقعی جا به جایی نیز 1 است. پس تفاضل h های استیت های متوالی کمتر یا مساوی هزینه واقعی خواهد بود پس این هیوریستیک سازگار است.

ثابت میشود هر هیوریستیک سازگار، قابل قبول هم هست. ثابت کردیم هیوریستیک بالا سازگار است پس حتما قابل قبول نیز هست.

با این هیوریستیک 1136 گره explore میشوند. (با توجه به نحوه امتیاز دهی، هیوریستیک دوم را در کد قرار داده ام که از 1200 کمتر باشد.)

7)

دو هیوریستیک که برای این مساله انتخاب شد که به شرح زیرند :

هیوریستیک اول:

h(state) = (طول مسیر در maze از state فعلی تا دورترین غذا + تعداد غذا های باقی مانده) /2

اگر در استیت هدف باشیم تعداد غذا های باقی مانده صفر است و ضمنا فاصله ای بین ما و دورترین غذا وجود ندارد پس آنهم صفر است در نتیجه h(goal) = 0

حال به استدلال اینکه چرا این هیوریستیک سازگار است میپردازیم.

فرض کنید یک خانه حرکت کنیم. فاصله ما تا دورترین غذا یا تغییر نمیکند، یا فقط یک واحد تغییر میکند. تعداد غذا هایی که در صفحه باقی مانده اند یا تغییر نمیکند یا فقط یک واحد تغییر میکند. بازهی مجموع تغییرات تعداد غذا و فاصله تا دورتیرن غذا در دو استیت متوالی بین 1- تا 2 است. حال طبق هیوریستیک انتخاب شده این عدد را بر 2 تقسیم میکنیم پس بازه تفاضل هیوریستیک تو استیت متوالی بین 1/2- و 1 است. هزینه واقعی در رفتن از یک خانه به خانه کناری 1 است. تفاضل هیوریتسیک دو استیت ، کمتر یا مساوی هزینه واقعی در رفتن از استیتی به استیت دیگر به دست آمد پس هیوریستیک سازگار است.

(جای تقسیم بر 2 میتوانستیم تقسیم بر هر عددی بزرگتر از 2 نیز قرار دهیم چون باز هم تفاضل هیوریستیک ها را از 1 کمتر میکرد اما نکته اینجاست اعداد بزرگتر از 2 باعث میشوند گره های بیشتری نیز explore شوند و این به صرفه نیست.)

ثابت میشود هر هیوریستیک سازگار، قابل قبول هم هست. ثابت کردیم هیوریستیک بالا سازگار است پس حتما قابل قبول نیز هست.

با این هیوریستیک 7895 گره explore میشوند.

هیوریستیک دوم:

صرفا طول مسیر درون maze تا دورترین غذا را به دست میاوریم.

اگر در استیت هدف باشیم قطعا طول این مسیر صفر است چون مسیری وجود ندارد.

در هر حرکت از استیتی به استیت متوالی، میتوانیم صرفا با هزینه 1 به خانه کناری برویم. یک خانه جا به جا شدن در maze یا ما را از دورترین نقطه 1 واحد دور میکند، یا فاصله مان از دورترین نقطه تغییر نمیکند و یا یک واحد به دور ترین نقطه نزدیکتر میشویم. در اینصورت تفاضل هیوریستیک دو استیت متوالی بین منفی 1 تا 1 است. این بازه قطعا کوچکتر یا برابر با 1 است پس از هزینه واقعی جابهجایی کمتر است پس هیوریستیک ما سازگار است.

ثابت میشود هر هیوریستیک سازگار، قابل قبول هم هست. ثابت کردیم هیوریستیک بالا سازگار است پس حتما قابل قبول نیز هست.

با این هیوریستیک 4137 گره explore میشوند.

با توجه به جدول امتیاز ها از هیوریستیک دوم برای حل سوال استفاده کرده ام.

هر دو هیوریستیک بالا ، با استفاده از تابع از پیش نوشته شده ی mazeDistance کوتاه ترین مسیر تا دورترین غذا به دست میاید. در این تابع از BFS برای یافتن این کوتاه ترین مسیر استفاده شده و یکی از دلایل طولانی بودن مدت محاسبات ، همین بارها جست و جوی BFS ایست که انجام میشود.

از تفاوت های هیوریستیک در این بخش و بخش قبل میتوان به استفاده از تابع mazedistance در یافتن دورترین گوشه اشاره کرد که تخمینی واقع بینانه نسبت به فاصله منهتن است که صرفا مجموع فاصله عمودی و افقی تا هدف را بدست میاورد. در کل میتوانستیم در سوال 6 به علت تعداد کم هدف ها هیوریستیکی تعریف کنیم که همه هدف ها

را در نظر بگیرد و خیلی نتایج بدی نداشته باشد اما در صورتی که در سوال 7 هم به دنبال هیوریستیکی میبودیم که موقعیت همه ی غذا ها را در خود داشته باشد ، تعداد گره های exploreشده به شدت زیاد میشد که برای ما بهینه نبود.

8)آزمون هدف در کلاس AnyFoodSearchProblem کافیست صرفا بگوید استیتی که میخواهیم هدف بودنش را بسنجیم غذا دارد یا نه و برای اینکار، در لیست غذا های maze میبینیم آیا state مورد نظر هست یا نه.

میخواهیم با استفاده از الگوریتم IDS تابع findPathToClosestDot را پیاده سازی کنیم.

Limit متغییریست که در آن عمق dfs در هر مرحله را محدود میکنیم.

از 1 شروع میشود و تا حداکثر 100 میرود در صورتی که limit از 100 بگذرد و به هدف نرسیم صرفا لیست خالی را برمیگردانیم.

در حلقه اصلی ابتدا چک میکنیم فرینج خالی نباشد. اگر فرینج خالی باشد یعنی همه ی گره های تا عمق لیمیت چک شده و هدف پیدا نشده. پس در صورت خالی بودن فرینج لیمیت را یک واحد زیاد میکنیم و اگر از 100 بیشتر نشد دوباره از استیت اولیه dfs را انجام میدهیم به همین علت fringe و visited را به حالت اول برمیگردانیم.

اگر فرینج خالی نبود استیت را پاپ میکنیم و چک میکنیم ایا هدف است (در آن غذا قرار دارد؟) اگر هدف بود حال کافیست کوتاه ترین مسیر از استیت اولیه تا استیت حال را با bfs به دست بیاوریم و برگردانیم.

اگر هدف نبود ساکسسور ها را اضافه میکنیم و بعد به ادامه حلقه میپردازیم.

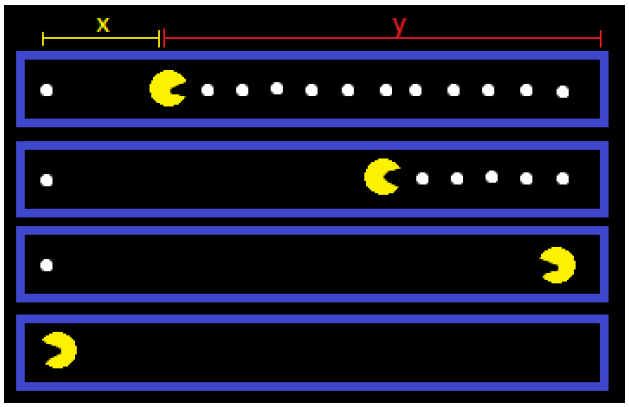
با این روش 342 گرهexpand میشوند و همه ی غذا ها خورده میشوند.

اگر در اخر با bfs کوتاه ترین مسیر را برنگردانیم 378 گره expand میشوند.

*توضیح
در کل میتوانستیم به جای الگوریتم ids از bfs استفاده کنیم. با توجه به اینکه در bfs سطح به سطح شعاع گشتن بزرگتر میشد و ما مطمین بودیم همیشه گره های کم سطح تر زودتر explore میشوند قطعا اولین جایی که با bfs به غذا میرسیم نیز نزدیکترین غذا به ما میبود.
با الگوریتم bfs تعداد گره های expand شده 350 میشود.

```
def findPathToClosestDot(self, gameState):
    """
    Returns a path (a list of actions) to the closest dot, starting from
    gameState.
    """
    # Here are some useful elements of the startState
    limit = 1
    startPosition = gameState.getPacmanPosition()
    problem = AnyFoodSearchProblem(gameState)
    fringe = util.Stack()
    curr_state = (startPosition, [])
    fringe.push((startPosition, []))
    visited = []


    while True:
        if fringe.isEmpty():
            limit += 1
            if limit > 100:
                break
            fringe = util.Stack()
            fringe.push((startPosition, []))
            visited = []

        curr_state = fringe.pop()
        curr_path = curr_state[1]
        curr_state = curr_state[0]
        if problem.isGoalState(curr_state):
            break


        if curr_state not in visited and len(curr_path) < limit:
            visited.append(curr_state)
            successors = problem.getSuccessors(curr_state)
            for s in successors:
                if s[0] not in visited:
                    new_path = list(curr_path)
                    new_path.append(s[1])
                    fringe.push((s[0], new_path))


    if limit <= 100:
        new_prob = PositionSearchProblem(gameState, start=startPosition,
                                         goal=curr_state, warn=False, visualize=False)
        return search.bfs(new_prob)
    else:
        return []
```

سوال) جست و جوی حریصانه برای نزدیکترین هدف آینده را پیش بینی نمیکند و به همین دلیل ممکن است کوتاه ترین مسیر را برنگرداند. به مثال زیر توجه کنید.



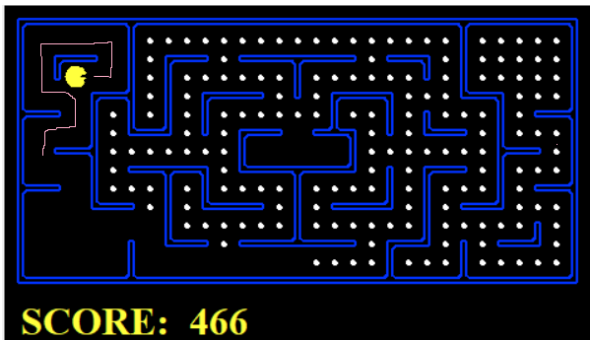
در این مثال میدانیم x کمتر از y است. حال عامل ما اگر در هر لحظه بخواهد نزدیکترین غذا را بخورد مسیر را ابتدا تا ته به راست میرود و سپس برای تنهای غذایی که در غرب است به چپ برمیگردد. هزینه کل میشود 2y + x چون مسیر yرا دوبار طی میکند(رفت و برگشت)

میدانیم روش بهینه این بود که ابتدا به چپ میامد و تک غذا را میخورد و بعد تا انتها به راست میرفت و در اینصورت هزینه میشد 2x + y که کمتر از 2y+xاست.

x < y => x + y < y + y => x + y + x < y + y + x => 2x+y < 2y+x

توضیح بیشتر در مورد bfs اخر:

در همین bigSearch وقتی به این حالت رسیدیم توقع داریم agent سپس به پایین امده و به نزدیکترین غذا برود اما میبینیم در عوض مسیر صورتی را طی میکند



علت اینست ترتیب ساکسسور ها به گونه ایست که برای اعداد بزرگ لیمیت، مسیر صورتی در الگوریتم dfs زودتر از مسیر صاف و رو به پایین پیمایش میشود به همین علت گره های درون این مسیر صورتی در visited قرار میگیرند و چون شرط پیمایش یک گره visited نبودن آنست دیگر مسیر صاف و رو به پایین اصلا پیمایش نمیشود که ببیند به هدف میرسد یا نه و فرینج خالی شده و لیمیت بیشتر و بیشتر میشود.

بنابراین میتوانیم در انتها با کمک گیری از الگوریتمی برای یافتن فاصله بهینه (مثل bfs) هزینه را کاهش دهیم.