سنا ساری نوایی

شماره دانشجویی : 810199435

**هوش مصنوعی**

**پروژه یک**

---

**توضیحات :** در این پروژه با استفاده از الگوریتم های BFS ، IDS و A* مسئله را حل می‌کنیم و در انتها آن ها را از نظر زمان اجرا با یکدیگر مقایسه کرده و نتیجه گیری می‌کنیم.

مسئله در یک گراف با n راس و m یال انجام می‌شود. یک نقطه شروع داریم که مکان ابتدایی سید است و رئوس دیگر میتوانند شامل مریدان، دستور پخت های سری و رئوس صعب العبور باشد. راس های یک گراف خالی نیز میتوانند باشند. هدف این است که حداقل زمان مورد نیاز برای رساندن دیزی به همه مریدان را با استفاده از الگوریتم های مختلف سرچ به دست آوریم.

## نحوه مدل کردن مسئله

- **Initial state:** برای شروع هر سرچ، ابتدا تابع initialize() فراخوانی می‌شود. در این تابع وضعیت اولیه استیت قرار داده میشود و یک کلاس برای استیت تعریف میگردد که در آن متغیر هایی مثل مسیر طی شده، وضعیت فعلی استیت، تعداد دستورالعمل های بدست آورده شده و ... در آن قرار میگیرد.
- **Actions :** حرکت هایی که میتوانیم انجام دهیم، این است که در هر خانه که هستیم، به خانه های مجاور میتوانیم حرکت کنیم و اگر در راس صعب العبور قرار داریم، با توجه به اینکه چند بار در این خانه قرار گرفته ایم، باید مدتی را در این خانه بمانیم و سپس حرکت کنیم.
- **Goal state :** یعنی استیتی که وقتی به آن برسیم، کار ما پایان می‌یابد. در اینجا goal state را وقتی در نظر گرفتم که تعداد مریدانی که دستور عمل های مورد نظر به دست آنان رسیده، با تعداد کل مریدان مسئله برابر باشد.
- **Transition model :** در هر مرحله که هستیم، باید مشخص شود به کدام خانه ها میتوانیم برویم، به عبارت دیگر باید استیت های همسایه را بررسی کنیم و بهترین راه را برای حرکت بعدی انتخاب کنیم. این کار در تابع next_states() انجام میگردد.
- **Path cost :** به دلیل اینکه یال های گراف وزنی ندارند، همه آن هارا یک در نظر می‌گیریم.

---

# BFS

در الگوریتم bfs جست و جو را از start_state شروع می‌کنیم و در صورتی به عمق بعدی میرویم که تمامی خانه در عمق فعلی را دیده باشیم. از مزیت الگوریتم bfs میتوان به این اشاره کرد که کامل میباشد یعنی اگر جوابی موجود باشد، حتما آن را بدست می‌آورد. مزیت دیگر آن این است که جوابی که میدهد همواره optimal است زیرا تمامی خانه ها را از سطح کم عمق به سطح پر عمق بررسی میکند.

در این قسمت ابتدا ورودی ها را از فایل میخوانیم. در edges_list جفت رئوسی که بین آنها یال قرار دارد، ذخیره میشود. در hard_to_pass شماره رئوس صعب العبور داده میشود. سپس دیکشنری به نام recipes درست کردیم که key در آن، شماره راسی است که مرید در آن است و value آن شماره رئوسی است که مرید مشخص کرده است و دستور پخت های سری در آن نوشته شده است. در دو لیست جداگانه نیز تمام مریدان و دستور پخت های سری را قرار دادیم تا دسترسی برایمان راحتتر شود.

```
f = open(FILE_NAME)

nodes , edges = f.readline().split()
nodes = int(nodes)
edges = int(edges)
type = []

#read edges
edges_list = {}
for i in range(edges):
    u,v = f.readline().split()
    u = int(u)
    v = int(v)
    if u not in edges_list:
        new_list = []
        new_list.append(v)
        edges_list[u] = new_list
    else:
        new_list = []
        new_list.append(v)
        edges_list[u].extend(new_list)
    if v not in edges_list:
        new_list = []
        new_list.append(u)
        edges_list[v] = new_list
    else:
        new_list = []
        new_list.append(u)
        edges_list[v].extend(new_list)
```

```
#hard to pass
hard_to_pass = []
hard_to_pass_number = int(f.readline())
hard_to_pass = [int(x) for x in f.readline().split()]

#recipes
follower_number = int(f.readline())
all_recipe = set()
morid = []
recipes = {}
for i in range(follower_number):
    lists = []
    p,q, *arr_follower = [int(x) for x in f.readline().split()]
    morid.append(p)
    all_recipe = all_recipe.union(arr_follower)
    recipes[p] = arr_follower

#start state
start_state = int(f.readline())

all_recipes = list(all_recipe)
f.close()
```

در قسمت بعدی یک کلاس به اسم State تشکیل می‌دهیم. در این کلاس وضعیت یک استیت نگهداری می‌شود به عنوان مثال نود فعلی، مسیر طی شده، دستور عمل هایی که تا الان بدست آورده ایم یا مریدانی که از آنان عبور کرده ایم در آن قرار دارند.

```
class State:
    def __init__(self,path,position,pass_recipes,pass_morid,hard_to_pass,hard_time):
        self.path = path
        self.position = position
        self.pass_recipes = pass_recipes
        self.pass_morid = pass_morid
        self.hard_to_pass = hard_to_pass
        self.hard_time = hard_time

    def __lt__(self,other):
        return self.path < other.path

    def __eq__(self, state2):
        if self.position == state2.position and self.hard_time == state2.hard_time and self.pass_recipes == state2.pass_recipes and self.pass_morid == state2
            return True
        else:
            return False
```

در تابع initialize() کلاس تعریف شده در قسمت قبل، مقدار دهی اولیه می‌شود و قبل از شروع به انجام هرگونه جست و جو، نیاز به این تابع داریم.

```
    def initialize():
        hard_dict = {}
        pass_recipes = set()
        pass_morid = []
        path = [start_state]
        for hard in hard_to_pass:
            hard_dict[hard] = 0
        start = State(path,start_state,pass_recipes,pass_morid,hard_dict,0)
        return start
```
✓ 0.4s

تابع next_states برای تعیین استیت بعدی براساس استیت فعلی مان می‌باشد یعنی در هر استیتی که الان قرار داریم، تمام استیت های همسایه آن را چک میکنیم و حالات مختلف را در نظر می‌گیریم. اگر استیتی که الان در آن قرار داریم، صعب العبور است، باید به اندازه تعداد باری که از آن عبور کرده ایم، در آن استیت صبر کنیم پس ممکن است استیت مان عوض نشود برای همین این حالت را جدا در نظر گرفته ایم. در بقیه حالات، نود ها را با توجه به ویژگی شان بررسی میکنیم و در نهایت تمامی استیت های ایجاد شده را ریترن می‌کنیم.

```
    def next_states(state):
        next_state = []
        if_in_hard = check_hard_state(state)
        if if_in_hard:
            #current state is hard
            neww = copy.deepcopy(state)
            neww.hard_time =neww.hard_time + 1
            next_state.append(neww)
            return next_state
        else:
            for neighbour in edges_list[state.position]:
                #create a new state for each neighbour
                neww = copy.deepcopy(state)
                neww.path.append(neighbour)
                neww.position = neighbour
                neww.pass_recipes = new_recipes(neww, neighbour)
                neww.pass_morid = new_morid(neww, neighbour)
                neww.hard_time = 0
                next_state.append(neww)
            return next_state
```
✓ 0.4s

در کد بالا تعدادی تابع استفاده شده است که آنهارا بررسی می‌کنیم. در تابع check_hard_state ابتدا بررسی میکنیم آیا این استیت صعب العبور هست یا خیر و سپس بررسی میکنیم اگر زمانی که در این استیت هستیم، کمتر از زمانیست که باید در این استیت باشیم، نتیجه میگیریم در استیت بعدی همینجا قرار داریم. در تابع new_recipes بررسی میکنیم اگر استیت همسایه دستور عمل سری داشت، آن را اضافه میکنیم در غیر اینصورت کاری نمی‌کنیم. در تابع new_morid نیز بررسیی می‌کنیم اگر استیت همسایه مرید باشد و تمامی دستور عمل سری آن را داشته باشیم، آن را به state.pass_morid اضافه می‌کنیم و در غیر اینصورت کاری نمی‌کنیم.

```
def check_hard_state(state):
    if state.position in hard_to_pass:
        if state.hard_to_pass[state.position] > state.hard_time:
            return True
        else:
            state.hard_to_pass[state.position] += 1
    return False
```
✓ 0.6s

```
def new_morid(state, neighbour):
    if neighbour in morid:
        if set(recipes[neighbour]).issubset(state.pass_recipes):
            if neighbour not in state.pass_morid:
                state.pass_morid.append(neighbour)
    return state.pass_morid
```
✓ 0.4s

```
def new_recipes(state, neighbour):
    if neighbour in all_recipes:
        state.pass_recipes.add(neighbour)
    return state.pass_recipes
```
✓ 0.5s

تابع ()bfs به صورت زیر می‌باشد. در ابتدا استیت شروع را به frontier اضافه می‌کنیم. سپس در هر مرحله یک استیت را از frontier را برمیداریم و از آن pop میکنیم و به visited اضافه می‌نماییم. سپس همه استیت های همسایه آن را بررسی می‌کنیم. پس از آن چک میکنیم که آیا در goal state قرار داریم یا خیر و این مراحل تا رسیدن به نتیجه همچنان تکرار میشود.

```
def BFS():
    frontier = []
    visited = []
    state = initialize()

    #check if we are in the goal state
    if len(morid) == len(state.pass_morid):
        return state.path
    #add the start state to the frontier
    frontier.append(state)

    while frontier:
        #get the first state in the frontier
        state = frontier.pop(0)
        visited.append(state)

        #get all the next states
        next_state = next_states(state)
        for s in next_state:
            checker = False
            if s not in visited:
                for i in range(len(frontier)):
                    if state_equal(s,frontier[i]):
                        checker = True
                        break
                if checker == False:
                    #check if we are in the goal state
                    if len(morid) == len(state.pass_morid):
                        return state.path

                    #add the state to the frontier
                    frontier.append(s)
    return state.path
```
✓ 0.5s

نتایج برای سه ورودی داده شده به صورت زیر است.

```
file name:  input.txt
visited states:  40
BFS :  1 3 4 5 7 10 11 9 8
BFS cost:  8
BFS average time:  0.0030331611633300078
```

```
file name:  input2.txt
visited states:  3897
BFS :  28 19 13 3 11 24 9 23 28 23 5 7 29
BFS cost:  12
BFS average time:  8.328672568003336
```

```
file name:  input3.txt
visited states:  5150
BFS :  40 42 38 24 31 45 30 48 41 18 1 19 43 49 47 49 9 34 25 50 12 16
BFS cost:  21
BFS average time:  6.02601679166158
```

---

## IDS

Iterative deeping search یا همان IDS یک الگوریتم سرچ ناآگاهانه میباشد که از DFS بهره می‌برد ولی با این تفاوت که عمق آن در هر مرحله یکی اضافه می‌شود. به عنوان مثال ابتدا با عمق 1، DFS می‌زنیم. اگر به جواب نرسیدیم، با عمق 2، DFS می‌زنیم و این مراحل تکرار می‌شود. از مزیت های این روش می‌توان به این اشاره کرد که IDS کامل میباشد یعنی اگر جوابی وجود داشته باشد، حتما آن را می‌یابد. مزیت دیگر آن این است که جواب بدست آمده حتما optimal است. IDS از الگوریتم BFS کندتر می‌باشد ولی برتری آن نسبت به BFS این است که از حافظه کمتری اشغال میکند.

برای این قسمت از سایت زیر استفاده شده است.

https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs

در تابع ()IDS یک حلقه داریم که در هر مرحله DFS را انجام میدهد و اگر به جواب نرسید، عمق آن را یک واحد افزایش می‌دهیم. در تابع ()DFS نیز به ازای هر عمق مشخص شده، ابتدا تمام همسایه های استیت کنونی را بررسی میکند و عمل DFS را به صورت بازگشتی انجام میدهد و اگر عمق کمتر از 1 شد، ریترن میکند.

```python
def DFS(level, state):
    #check if we are in the goal state
    if len(morid) == len(state.pass_morid):
        return True,state.path

    #stop if we reach the max level
    if level < 1:
        return False,None

    #get all the next states
    next_state = next_states(state)
    for s in next_state:
        situation,path = DFS(level - 1, s)
        if situation:
            return True,path
    return False,None
```

✓ 0.6s

```python
def IDS(level):
    max_level = level
    while max_level >= 0:
        state = initialize()
        situation,IDS_path = DFS(level,state)
        if situation:
            return IDS_path
        level = level + 1
        max_level = max_level + 1
```

✓ 0.3s

به علت طولانی بودن زمان پاسخگویی این الگوریتم برای ورودی هایی که در ابتدا داده شده است، از سری جدید ورودی که در سایت قرار دارد، استفاده می‌کنیم تا در زمان کمتر نتیجه را مشاهده کنیم.

```
file name:  testcases/input.txt
IDS :  1 3 4 5 7 10 11 9 8
IDS cost:  8
IDS visited:  2519
IDS average time:  0.10573164621988933
```

```
file name:  testcases/input2.txt
IDS :  9 10 9 4 12 3 7 5 8
IDS cost:  8
IDS visited:  6012
IDS average time:  0.2679746945699056
```

```
file name:  testcases/input3.txt
IDS :  13 11 10 3 2 6 12 5 9 4 1 13 11 10
IDS cost:  13
IDS visited:  128773
IDS average time:  7.084542751312256
```

## A*

حال به پیاده سازی الگوریتم A* می‌پردازیم. این یک الگوریتم آگاهانه است و نسبت به دو الگوریتم پیشین برتری دارد. برای استفاده از این الگوریتم ابتدا یک تابع heuristic تعریف میکنیم که consistent باشد. تابع heuristic مقدار هزینه تخمینی تا رسیدن به goal state را مشخص میکند و ما اینطور در نظر میگیریم که مقدار جمع دستور عمل های باقی مانده و مریدان باقی مانده را محاسبه میکنیم و آن را برمیگردانیم. وقتی به goal state برسیم، جمع آنها صفر میشود و در ابتدا نیز جمع آنها برابر جمع کل مریدان و دستورعمل های سری میباشد.

تابع heuristic در شکل زیر آمده است.

```
def heuristic(state):
    #calculate the heuristic value
    h = []
    for m in morid:
        if m not in state.pass_morid:
            if state.position not in h:
                h.append(state.position)
    for r in all_recipes:
        if r not in state.pass_recipes:
            if state.position not in h:
                h.append(state.position)
    return len(h)
```

به بررسی consistent بودن آن می‌پردازیم. چون در اینجا یال های گراف بدون وزن هستند، آن ها را یک در نظر میگیریم. فرض میکنیم میخواهیم از خانه A به خانه B برویم.تفاوت تعداد مجموع دستور عمل های باقی مانده و مریدان x است. با توجه به اینکه تفاوت هر نود با نود همسایه اش یک است، پس یعنی از خانه A تا خانه B حداقل x خانه فاصله است(میتواند بیشتر باشد زیرا ممکن است در هر خانه، دستور عمل یا مریدی نباشد در نتیجه خواسته ما ارضا نمیشود) پس هزینه واقعی از خانه A به B رفتن، همواره کمتر مساوی تفاوت تابع هیوریستیک میباشد.

تابع A_star نیز در شکل زیر آمده است. الگوریتم آن تقریبا شبیه به BFS است ولی به دلیل اینکه آگاهانه عمل میکنیم، در مدت زمان کمتری نسبت به BFS نتیجه میگیریم. برای اضافه کردن به frontier، از heapq استفاده میکنیم به اینصورت که کمترین h + g را در نظر میگیرد و انتخاب میکند که h همان خروجی تابع heuristic و g مسیری است که تا الان طی کرده ایم تا به هدفمان برسیم. بقیه مراحل شبیه BFS اجرا میشود.

برای قسمت weighted a* نیز یک متغیر alpha تعریف میکنیم و دو عدد مختلف برایش در نظر میگیریم و سه ورودی را برای هر دو حالت امتحان میکنیم. این الگوریتم نسبت به a* در زمان کمتری به جواب میرسد اما دقت آن کمتر است. در واقع به جای تولید جواب بهینه، سریعترین جواب را به ما میدهد.

```python
def A_star(alpha):
    frontier = []
    visited = []
    state = initialize()
    visited_state = 1

    #check if we are in the goal state
    if len(morid) == len(state.pass_morid):
        return len(state.path)-1,visited_state,state.path

    #add the start state to the frontier
    heapq.heappush(frontier,(alpha*heuristic(state)+len(state.path),state))

    while frontier:
        #get the first state in the frontier
        state = heapq.heappop(frontier)[1]

        #get all the next states
        next_state = next_states(state)
        for s in next_state:
            checker = False
            if s not in visited:
                for i in range(len(frontier)):
                    if s == frontier[i][1]:
                        checker = True
                        break
                if checker == False:
                    visited_state += 1

                    #check if we are in the goal state
                    if len(morid) == len(state.pass_morid):
                        return len(state.path)-1,visited_state,state.path

                    #add the state to the frontier
                    heapq.heappush(frontier,(alpha*heuristic(s)+len(s.path),s))
        visited.append(state)
    return len(state.path)-1,visited_state,state.path
```

✓ 0.3s

نتایج برای سه ورودی داده شده به صورت زیر میباشد:

```
file name:  input.txt
A* :  1 3 4 5 7 10 11 9 8
A* cost:  8
A* visited:  35
A* average time:  0.0036718050638834634
```

```
file name:  input2.txt
A* :  28 19 13 3 11 24 9 2 5 7 29 22 28
A* cost:  12
A* visited:  3626
A* average time:  5.870636701583862
```

```
file name:  input3.txt
A* :  40 42 38 24 31 45 30 48 41 18 1 19 43 49 47 49 9 34 25 50 12 16
A* cost:  21
A* visited:  4593
A* average time:  4.021674156188965
```

نتایج به ازای alpha = 1.6 به صورت زیر میباشد:

```
file name:  input.txt
A* :  1 3 4 5 7 10 11 9 8
A* cost:  8
A* visited:  35
A* average time:  0.0026704470316569O1
```

```
file name:  input2.txt
A* :  28 19 13 3 11 24 9 2 5 7 29 22 28
A* cost:  12
A* visited:  3626
A* average time:  5.356201092402141
```

```
file name:  input3.txt
A* :  40 42 38 24 31 45 30 48 41 18 1 19 43 49 47 49 9 34 25 50 12 16
A* cost:  21
A* visited:  4593
A* average time:  4.301829655965169
```

نتایج به ازای alpha = 4 به صورت زیر میباشد:

```
file name:  input.txt
A* :  1 3 4 5 7 10 11 9 8
A* cost:  8
A* visited:  35
A* average time:  0.0036718050638834634
```

```
file name:  input2.txt
A* :  28 19 13 3 11 24 9 2 5 7 29 22 28
A* cost:  12
A* visited:  3626
A* average time:  5.673522075017293
```

```
file name:  input3.txt
A* :  40 42 38 24 31 45 30 48 41 18 1 19 43 49 47 49 9 34 25 50 12 16
A* cost:  21
A* visited:  4593
A* average time:  6.01321546236674
```

تست اول:

| | پاسخ مسئله( حداقل زمان لازم برای رساندن دیزی ها) | تعداد استیت های دیده شده | میانگین زمان اجرا |
|---|---|---|---|
| BFS | 8 | 40 | 0.003033 |
| IDS | 8 | 2519 | 0.105731 |
| A* | 8 | 35 | 0.00367 |
| Weighted A* 1.6 | 8 | 35 | 0.00267 |
| Weighted A* 4 | 8 | 35 | 0.00367 |

تست دوم:( برای تست کیس دوم، IDS را در نظر نگرفتم و از تست کیس جدید که در سایت قرار داده شد، استفاده کردم)

| | پاسخ مسئله( حداقل زمان لازم برای رساندن دیزی ها) | تعداد استیت های دیده شده | میانگین زمان اجرا |
|---|---|---|---|
| BFS | 12 | 3897 | 8.32867 |
| IDS | - | - | - |
| A* | 12 | 3626 | 6.43810 |
| Weighted A* 1.6 | 12 | 3626 | 5.35620 |
| Weighted A* 4 | 12 | 3626 | 5.67352 |

تست سوم:( برای تست کیس سوم، IDS را در نظر نگرفتم و از تست کیس جدید که در سایت قرار داده شد، استفاده کردم)

| | پاسخ مسئله( حداقل زمان لازم برای رساندن دیزی ها) | تعداد استیت های دیده شده | میانگین زمان اجرا |
|---|---|---|---|
| BFS | 21 | 5150 | 6.02601 |
| IDS | - | - | - |
| A* | 21 | 4593 | 4.02167 |
| Weighted A* 1.6 | 21 | 4593 | 4.30182 |
| Weighted A* 4 | 21 | 4593 | 6.01321 |

**نتیجه گیری :** در حالت کلی، الگوریتمی که انتخاب میکنیم، بستگی به عوامل متفاوتی دارد. به عنوان مثال در Weighted A* نسبت به A* ، مزیت این است که الگوریتم سریعتر اجرا میشود ولی نقص آن این است که ممکن است جواب بهینه را به ما ندهد. یا الگوریتم BFS نسبت به الگوریتم IDS ، بسیار سریعتر است ولی فضای بیشتری نسبت به آن نیاز دارد پس یعنی با توجه به نیازمان باید انتخاب کنیم. اگر زمان برای ما مسئله مهمی بود و امکان این وجود داشت که از بهینگی صرف نظر کنیم، الگوریتم weighted a* برای ما مناسب است. اگر حافظه کمی داشته باشیم و بهینگی نیز مهم است، الگوریتم IDS برای ما مناسب است و اگر زمان و بهینگی برای ما مهم بود، از BFS میتوان استفاده کرد. الگوریتم A* نیز نسبت به BFS برتری دارد زیرا یک الگوریتم سرچ آگاهانه میباشد.

پس در نهایت ما باید با توجه به شرایط تصمیم بگیریم و بهترین الگوریتم را انتخاب کنیم.