مقدمه

هدف این آزمایش انجام الگوریتم های سرچ روی یک نقشه است که در این پروژه از BFS, IDS, Astar برای حل یک سؤال که در آن یک ماشین آمبولانس باید یک سری مریض را به بیمارستان ها ببرد استفاده شده است

الگوریتم ها و روش به دست آمدن آن‌ها به ترتیب توضیح داده می شوند:

BFS:

```
def bfs(self, state_properties, states_array):
```

تابع bfs که در کلاس الگوریتم وجود دارد برای سرچ bfs استفاده می‌شود
در این Method ما ۲ ورودی به آن داده‌ایم که State_properties یک دیکشنری متشکل از state و آیا مریض برداشته است یا خیر و آیا به بیمارستان رسیده است یا خیر قرار دارد که در واقع اطلاعات هر state است و state_array یک لیست از تمامی استیت های گذرانده شده تا به الان است یعنی مسیری که آمبولانس تا الان رفته است

در این الگوریتم ابتدا state های ممکن از state کی در حال حاضر در آن هستیم را به دست می‌آوریم

```
states_generated = self.problem.allowed_actions(state_properties)
```

برای پیدا کردن state ها از کلاس problem استفاده شده است که در ادامه به نحوه پیدا کردن آن می‌پردازیم این استیت ها در واقع child های state حاضر هستند

سپس به ازای هر child ابتدا چک می‌شود که آیا آن‌ها جزو هدف‌های ما بودند یا خیر و اگر بودند همان استیت expand می شود. هدف‌های ما برداشتن مریض رساندن بیمار و تمام شدن بیماران است

```
for section_state in states_generated:
    self.maxNodeExplored += 1
    have_final_state = self.problem.is_final_state(section_state)
    if have_final_state == "Solve":
        new_copy = copy.deepcopy(states_array)
        new_copy.append(section_state)
        self.solved = True
        self.answerDepth = len(states_array)
        # print("Final for BFS")
        # print(np.array(section_state['state']).reshape(
        #     [len(section_state['state']), len(section_state['state'][-1])]))
        return section_state, states_array
    if have_final_state == "Patient":
        self.problem.have_patient = True
        preferred_state = section_state
        break
    if have_final_state == "Hospital":
        self.problem.have_patient = False
        preferred_state = section_state
        break
```

که با تمام شدن بیماران کار agent تمام می‌شود و در دو حالت دیگر می‌فهمیم که آیا آمبولانس بیمار دارد یا خیر
و این نود ها جزو explore ست ما قرار می‌گیرند

سپس می‌بینیم اگر یکی از استیت ها هدف بود آن استیت و در غیر این صورت استیت ها را به ترتیب ادامه داده و دوباره BFS را برای آن‌ها صدا می‌کنیم و آن‌ها را جزو Explanded ست برده و به ارایه ی state_array که مسیر رسیدن تا به همان استیت را نگه می‌دارد اضافه می‌کنیم

```
if len(preferred_state) != 0 and preferred_state['state'] not in states_array:
    # print(preferred_state)
    self.maxNodeExpanded += 1
    new_copy = copy.deepcopy(states_array)
    new_copy.append(preferred_state['state'])
    self.bfs(preferred_state, new_copy)
else:
    for section_state in states_generated:
        if section_state["state"] not in states_array:
            self.maxNodeExpanded += 1
            new_copy = copy.deepcopy(states_array)
            new_copy.append(section_state['state'])
            self.bfs(section_state, new_copy)
```

و این‌قدر همین کار را انجام می‌دهیم تا به استیت نهایی برسیم و کار تمام شود.

IDS

این الگوریتم همانند DFS است با این تفاوت که عمق سرچ را هم به آن می‌دهیم و agent فقط تا عمق داده شده می‌تواند سرچ را ادامه دهد که در این صورت ممکن است به استیت نهایی نرسد زیرا در آن عمق به جواب نرسیده است

```
def ids(self, state_properties, states_array, max_depth):
```

که همانند قبل استیت حال حاضر لیست ی که استیت های تا به الان یا همان مسیر را در خود دارد و عمقی که agent می‌تواند در آن سرچ کند می‌گیرد

در این الگوریتم ابتدا به ازای هر استیت که می‌بیند عمق مسیر یکی زیاد می‌شود و سپس چک می‌شود که آیا به یکی از استیت های هدف رسیده‌ایم یا خیر و چک می‌شود که آیا به عمق ای که می‌توانست برسد رسیده یا خیر که در این صورت باید از الگوریتم خارج شود

```
if have_final_state == "Solve":
    # print("Final for DFS_Growing_Depth")
    # print(np.array(state_properties['state']).reshape(
    #     [len(state_properties['state']), len(state_properties['state'][-1])]))
    self.solved = True
    self.answerDepth = len(states_array)
    return state_properties, states_array
if have_final_state == "Patient":
    self.problem.have_patient = True
if have_final_state == "Hospital":
    self.problem.have_patient = False
if len(states_array) >= max_depth:
    return -1
states_generated = self.problem.allowed_actions(state_properties)
self.maxNodeExpanded += 1
self.maxNodeExplored += 1
```

که در آن هر سری هر مانند dfs هر نود expand و سپس explore می‌شود

سپس مانند الگوریتم قبل به ازای هر استیت تولید شده اگر تکراری نبود آن را به ارایه مسیر همان state_array اضافه کرده و آن را بسط می‌دهیم

```
for section_state in states_generated:
    if section_state['state'] not in states_array:
        new_copy = copy.deepcopy(states_array)
        new_copy.append(section_state['state'])
        self.ids(section_state, new_copy, max_depth)
```

و همین کار را ادامه می‌دهیم تا به استیت نهایی برسیم

A*

تفاوت این الگوریتم با دیگر الگوریتم ها این است که یک تابع هزینه نیز دارد
این تابع هزینه هر استیت را تا رسیدن به subgoal یا هدف نهایی مشخص می‌کند پس این چنین agent می‌تواند تصمیم بگیرد که کدام مسیر هزینه کمتری دارد و از بین مسیر ها همان مسیر را انتخاب کند

```
def a_star(self, state_properties, states_array):
```

در این الگوریتم همانند قبلی‌ها استیت ای که در آن هستیم و ارایه ای که مسیر agent را مشخص می‌کند به آن داده می‌شود

سپس ابتدا تمام استیت هایی که از استیت حاضر می‌توان به آن رفت را پیدا میکنیم:

```
states_generated = self.problem.allowed_actions(state_properties)
```

سپس برای استیت های بدست آمده برای هر یک هزینه آن را با توجه به heuristic مربوط به الگوریتم پیدا می‌کنیم
برای این پروژه از ۲ heuristic استفاده کرده‌ایم پس یعنی فرق الگوریتم ها فقط در تابع هزینه و پیدا کردن مقدار آن است که در ادامه آن‌ها را توضیح می‌دهم
سپس برای هر استیت بدست آمده تابع هزینه را صدا کرده و هزینه همه ی استیت ها را بدست می‌آوریم

```
states_cost = []
for section_state in states_generated:
    section_state.update({"cost": self.state_cost(section_state)})
    states_cost.append(section_state)
    # print(np.array(section_state['state']).reshape(
    #     [len(section_state['state']), len(section_state['state'][-1])]))
sorted_states = sorted(states_cost, key=lambda kv: kv['cost'])
```

و سپس با sort کردن آن استیت ها را به ترتیب هزینه مرتب می‌کنیم پس استیت با هزینه کمتر در اول لیست قرار می‌گیرد

سپس مانند دیگر الگوریتم ها میبینیم که آیا هیچ یک از این استیت ها هدف‌های ما هستند یا خیر:

```
for section_rating in sorted_states:
    self.maxNodeExplored += 1
    have_final_state = self.problem.is_final_state(section_rating)
    if have_final_state == "Solve":
        self.solved = True
        new_copy = copy.deepcopy(states_array)
        new_copy.append(section_rating['state'])
        # print("Final for AStar")
        # print(np.array(section_rating['state']).reshape(
        #     [len(section_rating['state']), len(section_rating['state'][-1])]))
        self.answerDepth = len(states_array) + 1
        return section_rating, states_array
    if have_final_state == "Patient":
        self.problem.have_patient = True
        preferred_state = section_rating
        break
    if have_final_state == "Hospital":
        self.problem.have_patient = False
        preferred_state = section_rating
        break
```

در صورتی که یکی از آن‌ها هدف ما نیز بود همان استیت را بسط می‌دهیم و در غیر این صورت استیت با کمترین هزینه که درواقع عضو اول لیستمان است را بسط می‌دهیم

```
if len(preferred_state) != 0 and preferred_state['state'] not in states_array:
    new_copy = copy.deepcopy(states_array)
    new_copy.append(preferred_state['state'])
    self.maxNodeExpanded += 1
    self.a_star(preferred_state, new_copy)
else:
    for section_rating in sorted_states:
        if section_rating['state'] not in states_array:
            new_copy = copy.deepcopy(states_array)
            new_copy.append(section_rating['state'])
            self.maxNodeExpanded += 1
            self.a_star(section_rating, new_copy)
```

و این کار را ادامه می‌دهیم تا به هدف آخر برسیم و کار تمام شود

همان‌طور که گفته شد الگوریتم Astar را با استفاده از ۲ heuristic بدست می‌آوریم که یعنی ۲ تابع هزینه خواهیم داشت:

Heuristic 1

اولین الگوریتم را این‌طور میگیریم که هزینه ما درواقع مجموع تعداد خانه های افقی و عمودی فاصله ی آمبولانس تا هدف‌ها است. یعنی وقتی بیمار ندارد هزینه تا بیمار و وقتی دارد هزینه تا نزدیک‌ترین بیمارستان است

این الگوریتم admissable نیز هست زیرا این مقدار بدون در نظر گرفتن دیوار ها است پس همیشه از مقدار واقعی کمتر یا مساوی است

این تابع را به شکل زیر می نویسیم:

که در آن cost همان هزینه مربوط به هر استیت است

```
def state_cost(self, state_properties, first_heuristic):
    cost = 0
    distances = []
    ambulance_y, ambulance_x = self.problem.find_ambulance(state_properties['state'], True)
    if first_heuristic is True:
        if state_properties['have_patient'] is False and self.problem.have_patient is False:
            patient_list = self.problem.find_patients(state_properties['state'])
            for i in range(len(patient_list)):
                distances.append(
                    abs(patient_list[i][0] - ambulance_y) + abs(
                        patient_list[i][1] - ambulance_x))
            cost = min(distances)
        if state_properties['have_patient'] is True or self.problem.have_patient is True:
            hospital_list = self.problem.find_hospital(state_properties['state'], True)
            for i in range(len(hospital_list)):
                distances.append(
                    abs(hospital_list[i][0] - ambulance_y) + abs(
                        hospital_list[i][1] - ambulance_x))
            cost = min(distances)
```

که درصورتی که ورودی True باشد از این الگوریتم استفاده می‌شود

Heuristic 2

برای این الگوریتم می‌آیم فاصله هر بیمار با بیمارستان را محاسبه می‌کنیم و از بین آن‌ها نزدیک‌ترین بیمارستان به بیمار را پیدا میکنیم سپس برای تمام بیماران نیز این کار را می‌کنیم
در آخر نزدیک‌ترین فاصله (یعنی بیماری که بیمارستان نزدیک‌تر است) را انتخاب می‌کنیم و فاصله آمبولانس تا آن همان هزینه ما می‌شد مانند زیر:

```
else:
    if state_properties['have_patient'] is False and self.problem.have_patient is False:
        patient_list = self.problem.find_patients(state_properties['state'])
        hospital_list = self.problem.find_hospital(state_properties['state'], True)
        for i in range(len(patient_list)):
            distance = []
            for j in range(len(hospital_list)):
                distance.append(abs(patient_list[i][0] - hospital_list[j][0]) + abs(
                    patient_list[i][1] - hospital_list[j][1]))
            distances.append(min(distance))
        cost = min(distances)

return cost
```

که یک الگوریتم ضعیف است زیرا در بیشتر موارد تمام هزینه‌ها برابر می‌شوند و نمی‌تواند درست تصمیم بگیرد به همین دلیل بیشتر طول می‌کشد تا به نتیجه برسد و به استیت نهایی برسیم

این الگوریتم admissible هست ولی consistent نیست به همین دلیل جواب بهینه نمی‌دهد

در مقایسه با الگوریتم قبلی که consistent هم بود (قبلی جواب بهینه میدهد) بسیار ضعیف‌تر است و زمان بیشتری طول می کشد.

Problem:

برای پیدا کردن استیت های موجود از یک استیت دیگر و استفاده آن در الگوریتم های بالا یک کلاس به اسم Problem درست کرده‌ایم

در این کلاس استیت حاضر را میدهیم و آن استیت هایی که می‌توان به آن رفت را به ما بر می گرداند

```
states_generated = self.problem.allowed_actions(state_properties)
```

در هر الگوریتم صدا می‌شود و داریم:

```
#  To see if agent is busy or ready for a new action
def allowed_actions(self, state_properties):
    if self.actions is not None:
        return self.actions[state_properties]
    else:
        return self.state_generator(state_properties)
```

که در آن می‌بینیم آیا Agent در حال انجام کاری است یا خیر

سپس تابع بعدی صدا می‌شود و داریم:

```python
# To Generate the states that agent can move in that direction
def state_generator(self, state_properties):
    # To find where is the ambulance!
    self.find_ambulance(state_properties['state'], False)
    allowed_moves = ['up', 'down', 'right', 'left']
    # To remove the direction that there is an obstacle in the way
    if self.check_obstacle(state_properties, self.ambulance_x, self.ambulance_y - 1):
        allowed_moves.remove('up')
    if self.check_obstacle(state_properties, self.ambulance_x, self.ambulance_y + 1):
        allowed_moves.remove('down')
    if self.check_obstacle(state_properties, self.ambulance_x - 1, self.ambulance_y):
        allowed_moves.remove('left')
    if self.check_obstacle(state_properties, self.ambulance_x + 1, self.ambulance_y):
        allowed_moves.remove('right')
    new_map_properties = []  # New possible moves to iterate on
    for move in allowed_moves:
        new_map_properties.append(self.assign_movement(state_properties, move))
    return new_map_properties
```

در این تابع بعد از پیدا کردن مکان آمبولانس چون می‌دانیم فقط ۴ اکشن می‌تواند انجام دهد چک می‌کنیم که آیا مانعی سر راه دارد یا خیر و اگر بود آن اکشن را نمی‌توانیم انجام دهیم

```python
# To Check what is our next move obstacle and generate the allowed_movement
def check_obstacle(self, state_properties, x_coordinate, y_coordinate):
    # To see if there is a WALL or Patient or Hospital
    if (state_properties['state'][y_coordinate][x_coordinate] == '#') or (
            state_properties['state'][y_coordinate][x_coordinate] == '0') or (
            self.have_patient is True and state_properties['state'][y_coordinate][
            x_coordinate] == 'P') or (
            self.have_patient is False and state_properties['state'][y_coordinate][
            x_coordinate].isdigit()):
        return True
    else:
        return False
```

برای پیدا کردن مانع از تابع زیر استفاده می‌شودکه مختصات نقطه مقصد اکشن را می‌گیرد و می‌فهمد که آیا این حرکت ممکن است یا نه

سپس بعد از پیدا کردن اکشن ها برای هر کدام استیت را درست می‌کنیم که همان تابع assign_movement است و داریم:

```
def assign_movement(self, state_properties, movement):
    state_update_properties = copy.deepcopy(state_properties)  # to update for a new state

    # Check how to move in correct direction
    if movement == 'up':
        state_update_properties = self.create_movement(state_properties, self.ambulance_x, self.
                                                       movement)
    if movement == 'down':
        state_update_properties = self.create_movement(state_properties, self.ambulance_x, self.
                                                       movement)
    if movement == 'right':
        state_update_properties = self.create_movement(state_properties, self.ambulance_x + 1, s
                                                       movement)
    if movement == 'left':
        state_update_properties = self.create_movement(state_properties, self.ambulance_x - 1, s
                                                       movement)
    return state_update_properties

# To Generate the states that agent can move in that direction
```

این تابع نوع حرکت و استیت حاضر را می‌گیرد و با توجه به حرکت آنکه همان اکشن است استیت جدید را می‌سازد

تابع create_movement برای ساخت استیت جدید با توجه به نقطه مقصد است که داریم:

```
# Check if our new action needs changes in our agent and update for ne State
def create_movement(self, state_properties, x_coordinate, y_coordinate, movement):
    state_update = copy.deepcopy(state_properties['state'])
    state_properties = {'state': state_update, 'have_patient': False, 'is_hospital': False,
                        'object_position': movement, 'ambulance_x': self.ambulance_x,
                        'ambulance_y': self.ambulance_y
                        }
    # To see if we dont have patient and our next move is patient so we should pick it up!
    if self.have_patient is False and state_properties['state'][y_coordinate][x_coordinate] == 'P':
        state_update[y_coordinate][x_coordinate] = 'A'
        state_update[self.ambulance_y][self.ambulance_x] = ' '
        state_properties.update({'state': state_update, 'have_patient': True, 'is_hospital': False,
                                 'object_position': movement, 'ambulance_x': x_coordinate,
                                 'ambulance_y': y_coordinate})

    # To see if we already have a patient and out next move is a not empty hospital so we take off
    elif self.have_patient is True and state_properties['state'][y_coordinate][
        x_coordinate].isdigit() and (
            int(state_properties['state'][y_coordinate][x_coordinate])) != "0":
        state_update[y_coordinate][x_coordinate] = \
            str(int(state_properties['state'][y_coordinate][x_coordinate]) - 1)
        state_update[self.ambulance_y][self.ambulance_x] = 'A'
        state_properties.update({'state': state_update, 'have_patient': False, 'is_hospital': True,
                                 'object_position': movement, 'ambulance_x': self.ambulance_x,
                                 'ambulance_y': self.ambulance_y})
```

```
# To see if it is just a normal move
else:
    state_update[y_coordinate][x_coordinate] = 'A'
    state_update[self.ambulance_y][self.ambulance_x] = ' '
    state_properties.update({'state': state_update, 'ambulance_x': x_coordinate,
                             'ambulance_y': y_coordinate})
return state_properties
```

این تابع با توجه به اکشن و حالتی که استیت به آن می‌رود آن را می‌سازد و برای هر حرکت آن را بر می گرداند

سپس ما به ازای هر حرکت ممکن یک استیت ساخته‌ایم که آن‌ها را در یک لیست نوشته و همه را به الگوریتم اصلی می‌دهیم که از بین آن‌ها یکی را انتخاب کند و الگوریتم را ادامه دهد.

برای چک کردن اینکه آیا به هدف رسیده‌ایم نیز یک تابع داریم که استیت را گرفته و شرایط را چک می‌کند

```
# To check if we got the final state and our job is done!
def is_final_state(self, state_properties):
    empty_hospitals = 0
    self.find_hospital(state_properties['state'], False)
    for i in range(len(self.hospital_list)):
        if state_properties['state'][self.hospital_list[i][0]][self.hospital_list[i][1]] == '0':
            empty_hospitals = empty_hospitals + 1
            if empty_hospitals == len(self.hospital_list):
                return "Solve"
    if state_properties['have_patient'] is True:
        return "Patient"
    if state_properties['is_hospital'] is True:
        return "Hospital"
    return "Default"
```

که از آن در الگوریتم اصلی استفاده می شود.

مقایسه

می‌دانیم که الگوریتم ids از بقیه بسیار کند تر است و حتی ممکن است اگر عمق را کم بدهیم ممکن است نتواند جواب را در آن عمق پیدا کند
الگوریتم bfs می‌تواند سریع‌تر از ids به جواب برسد ولی اگر یک heuristic که هم admissible و هم consistent باشد برای astar درست کنیم astar سریع‌تر به جواب می رد و جواب آن بهینه است ولی اگر این heuristic یه consistent نباشد bfs ممکن است زود تر به نتیجه برسد ولی باز هم تعداد نود دیده شده و در نتیجه حافظه برای هر دو حالت astar از bfs می‌تواند کمتر باشد این در حالی است که زمان برای ids از همه بیشتر است ولی حافظه آن از BFS بیشتر و از astar کمتر است

حال الگریتم ها را برای هر ۳ تست کیس استفاده می‌کنیم و داریم:

تست ۱:

```
DFS unlimited :
max node explored :  52  max node expanded :  52  answer depth :  42
Elapsed Time is:  0.05611824989318848
BFS answer :
max node explored :  48  max node expanded :  25  answer depth :  23
Elapsed Time is:  0.009853124618530273
DFS growing depth answer :
max node explored :  52  max node expanded :  52  answer depth :  42
Elapsed Time is:  0.05479764938354492
A* answer with admissible heuristic:
max node explored :  22  max node expanded :  16  answer depth :  16
Elapsed Time is:  0.004567146301269531
A* answer second heuristic:
max node explored :  45  max node expanded :  25  answer depth :  24
Elapsed Time is:  0.01067805290222168
```

تست ۲:

```
DFS unlimited :
max node explored :  69  max node expanded :  69  answer depth :  51
Elapsed Time is:  0.09352493286132812
BFS answer :
max node explored :  65  max node expanded :  43  answer depth :  33
Elapsed Time is:  0.021203041076660156
DFS growing depth answer :
max node explored :  69  max node expanded :  69  answer depth :  51
Elapsed Time is:  0.09264206886291504
A* answer with admissible heuristic:
max node explored :  63  max node expanded :  42  answer depth :  33
Elapsed Time is:  0.02806687355041504
A* answer second heuristic:
max node explored :  61  max node expanded :  43  answer depth :  34
Elapsed Time is:  0.02277207374572754
```

تست ۳:

```
DFS unlimited :
max node explored :  86  max node expanded :  86  answer depth :  79
Elapsed Time is:  0.23938703536987305
BFS answer :
max node explored :  146  max node expanded :  80  answer depth :  46
Elapsed Time is:  0.08669900894165039
DFS growing depth answer :
max node explored :  98  max node expanded :  98  answer depth :  59
Elapsed Time is:  0.24646592140197754
A* answer with admissible heuristic:
max node explored :  109  max node expanded :  71  answer depth :  47
Elapsed Time is:  0.07442688941955566
A* answer second heuristic:
max node explored :  139  max node expanded :  80  answer depth :  47
Elapsed Time is:  0.08898711204528809
```

که در همه ی آن‌ها اول Astar admissible consistent سپس bfs و بعد Astar و در آخر ids تمام می‌کند

برای جدول داریم:

	فاصله جواب	تعداد استیت دیده شده	تعداد استیت دیده شده مجزا	زمان اجرا
BFS	34	86.33	51.33	0.038
IDS	50.66	73	73	0.130
Astar First	32	64.66	43	0.035
Astar Second	35	82.66	49.33	0.04