تمرین سری دوم هوش مصنوعی

کیمیا اسماعیلی - 610398193

مقدمه:

در این تمرین به بررسی نحوه حل بازی ۸-پازل پرداخته شده است. این تمرین به کمک الگوریتم‌های جستجو تلاش می‌کند تا بتواند مسئله ۸-پازل را حل کند. در این تمرین از انواع مختلف روش‌های جستجو مانند DFS، BFS، IDS، UCS، A* استفاده شده است و زمان اجرا و میزان حافظه مورد استفاده در فایل اکسل ضمیمه شده در فایل زیپ به تفصیل مقایسه شده است.

توضیح Stateها:

برای انجام جستجو نیاز بود تا تعریفی از Stateهایی که میان آن‌ها جستجو را انجام می‌دهیم ارائه دهیم. محل قرار گرفتن هر کدام از اعداد در مربع ۳ در ۳ به عنوان State تعریف شده است. در پیاده‌سازی کلاس State تعریف شده است که توابع کمکی و موقعیت تمام اعداد و خانه‌ی خالی را به صورت یک آرایه دو بعدی نگه‌داری می‌کند.

توضیح Actions:

در این تمرین چهار جهت بالا، پایین، چپ و راست به عنوان actionهایی که می‌توانیم برای رسیدن به State بعدی انجام دهیم، تعریف شده است. نکته قابل توجه این است **تعریف از action به این صورت است که خانه‌ی غیر خالی که جای خانه خالی را می‌گیرد به کدام جهت حرکت کرده است.** برای مثال زمانی که یک عدد که پایین خانه‌ی خالی قرار دارد جای آن را می‌گیرد می‌گوییم که action بالا انجام شده است.

توضیح Goal State:

برای اینکه بررسی کنیم که به حالت هدف رسیده‌ایم یا خیر از شکلی که در صورت مسئله به عنوان حالت هدف ذکر شده است، استفاده شده است. زمانی که به این چینش از اعداد برسیم به حالت هدف رسیده‌ایم.

۱	۲	۳
۴	۵	۶
۷	۸	

حالت هدف

توضیح Path cost:

در این تمرین path cost به صورت تعداد Stateهایی که به صورت متوالی باید طی شوند تا به حالت هدف برسیم تعریف شده است.

توضیح Transition model:

به صورت یک تابع در کلاس State پیاده‌سازی شده است که در ادامه توضیح داده شده است. این تابع با گرفتن یک action حالت Stateی که از این State خروجی داده می‌شود را بازمیگرداند.

توضیح تابع heuristic:

از روش Manhattan distance استفاده شده است. در این روش هر عدد با محل قرارگیری نهایی خود مقایسه می‌شود که با استفاده از مجموع فاصله‌ها مقدار نهایی heuristic مشخص می‌شود.

توضیح کد:

در ابتدا کتابخانه‌های مورد نیاز را اضافه می‌کنیم:

```
import enum
import numpy as np
import tracemalloc
import time
from collections import deque
import copy
import sys
import heapq
```

در ادامه تابع read_input پیاده‌سازی شده است که یک رشته دریافت می‌کند و آرایه دوبعدی خانه‌ها را در اختیار می‌گذارد:

```
def read_input(line):
    input_2d = []
    input_array = list(map(int, line.split(",")))
    for i in range(9):
        if i%3 == 0:
            input_2d.append([input_array[i]])
        else:
            input_2d[i//3].append(input_array[i])
    return input_2d
```

برای تعریف actionها در یک محل از کلاسی به صورت Enum استفاده شده است که در آن حالت‌های مختلف حرکت تعریف شده است. همچنین با استفاده از تابع actions در این کلاس می‌توان همه‌ی actionهای موجود را دریافت کرد.

```
class Direction(enum.IntEnum):
    Up = 0
    Right = 1
    Down = 2
    Left = 3

    @staticmethod
    def actions():
        return [Direction.Right, Direction.Down, Direction.Left, Direction.Up]
```

در بخش بعد کلاس State تعریف شده است که بخش به بخش توضیح داده می‌شود. تابع init که در زیر تعریف شده است آرایه دوبعدی موقعیت‌ها را دریافت می‌کند و مقداردهی اولیه‌ای به State می‌دهد.

```
def __init__(self, positions):
    self.positions = positions
```

در ادامه دو تابع eq و hash به دلیل اینکه پایتون برای مقایسه از آن‌ها در پشت صحنه استفاده می‌کند تعریف شده است. در تابع eq، آرایه موقعیت اعداد و آرایه‌ای دیگر توسط numpy مقایسه می‌شود تا برابری مشخص شود. در تابع hash، با استفاده از هش کردن همه موقعیت‌های اعداد در این State یک عدد یکتا برای نمایندگی این State مشخص شده است.

```
def __eq__(self, other):
    return np.array_equiv(self.positions, other.positions)

def __hash__(self):
    return hash(tuple(np.reshape(self.positions, (1, 9))[0].tolist()))
```

در تابع is_goal رسیدن به حالت هدف که پیش‌تر ذکر شد بررسی می‌شود:

```
def is_goal(self):
    return np.array_equiv(self.positions, [[1, 2, 3], [4, 5, 6], [7, 8, 0]])
```

در تابع calculate_heuristic، مقدار heuristic مربوط به این State محاسبه می‌شود که همان‌طور که پیش‌تر ذکر شد به صورت Manhattan distance میباشد.

```
def calculate_heuristic(self):
    sum_manhat_dist = 0
    for i in range(3):
        for j in range(3):
            current_val = self.positions[i][j]
            if current_val == 0:
                continue
            goal_x = (current_val - 1) // 3
            goal_y = (current_val -1) % 3
            sum_manhat_dist += abs(i - goal_x) + abs(j - goal_y)
    return sum_manhat_dist
```

در تابع transition، نحوه‌ی State Transition مشخص می‌شود. ابتدا موقعیت بخش خالی یا همان عدد ۰ پیدا می‌شود. سپس بررسی می‌شود که آیا اصلا action اعلام شده امکان‌پذیر است یا خیر. در صورتی که این action امکان‌پذیر نباشد مقدار Fasle به عنوان خروجی تابع بازگردانده می‌شود. در غیر این صورت، بسته به جهت حرکت عنصر خالی با عنصر مجاور خود تعویض می‌شود.

```
def transition(self, action):
    zero_pos_x = 0
    zero_pos_y = 0
    for i in range(3):
        for j in range(3):
            if (self.positions[i][j] == 0):
                zero_pos_x = i
                zero_pos_y = j
    if (
        (zero_pos_x == 0 and action == Direction.Down) or
        (zero_pos_x == 2 and action == Direction.Up) or
        (zero_pos_y == 0 and action == Direction.Right) or
        (zero_pos_y == 2 and action == Direction.Left)
    ):
        return False
    new_positions = copy.deepcopy(self.positions)
    if action == Direction.Up:
        new_positions[zero_pos_x][zero_pos_y] = new_positions[zero_pos_x+1][zero_pos_y]
        new_positions[zero_pos_x+1][zero_pos_y] = 0
    elif action == Direction.Down:
        new_positions[zero_pos_x][zero_pos_y] = new_positions[zero_pos_x-1][zero_pos_y]
        new_positions[zero_pos_x-1][zero_pos_y] = 0
    elif action == Direction.Left:
        new_positions[zero_pos_x][zero_pos_y] = new_positions[zero_pos_x][zero_pos_y+1]
        new_positions[zero_pos_x][zero_pos_y+1] = 0
    elif action == Direction.Right:
        new_positions[zero_pos_x][zero_pos_y] = new_positions[zero_pos_x][zero_pos_y-1]
        new_positions[zero_pos_x][zero_pos_y-1] = 0
    return State(new_positions)
```

یک کلاس کمکی Node نیز تعریف شده است که بتوانیم Stateها و زنجیره‌ی طی شده را نگهداری کنیم:

```
class Node:
    def __init__(
        self, _state: State, _parent, _action, _path_cost
    ):
        self.state = _state
        self.parent = _parent
        self.action = _action
        self.path_cost = _path_cost

    def __lt__(self, other):
        return self
```

تابع __lt__ به این منظور تعریف شده است که پایتون بتواند بین Nodeها مقایسه انجام دهد.
در نهایت تابع run تعریف شده است که با دریافت یک الگوریتم آن را اجرا می‌کند و اطلاعات مورد نیاز را چاپ می‌کند. برای محاسبه زمان از کتابخانه time و برای محاسبه حافظه از tracemalloc استفاده شد.

```
def run(algorithm_func, start_node, *args, **kwargs):
    mem_used = 0
    tic = time.time()
    tracemalloc.start()
    final = algorithm_func(start_node, *args, **kwargs)
    mem_used = tracemalloc.get_traced_memory()[1]
    tracemalloc.stop()
    toc = time.time()
    if final == False:
        print("NOT POSSIBLE")
        return
    answer_actions = []
    while final is not None:
        answer_actions = [final.action] + answer_actions
        final = final.parent
    print("Actions: ", end="")
    for action in answer_actions[1:]:
        print(action.name, end=" ")
    print("Time taken: {}".format(toc - tic))
    print("Memory used: {}".format(mem_used))
```

تابع run_for_tests تابع run را برای همه‌ی تست‌های آسان، متوسط و سخت چاپ می‌کند.

```
def run_for_tests(algorithm_func, *args, **kwargs):
    i = 1
    for test in EASY_INSTANCES:
        print("EASY INSTANCE TEST NUMBER {}".format(i))
        i += 1
        start_node = Node(State(read_input(test)), None, None, 0)
        run(algorithm_func, start_node, *args, **kwargs)
        print("----------------------------------------------------------------------------------------------------")
    i = 1
    for test in MEDIUM_INSTANCES:
        print("MEDIUM INSTANCE TEST NUMBER {}".format(i))
        i += 1
        start_node = Node(State(read_input(test)), None, None, 0)
        run(algorithm_func, start_node, *args, **kwargs)
        print("----------------------------------------------------------------------------------------------------")
    i = 1
    for test in HARD_INSTANCES:
        print("HARD INSTANCE TEST NUMBER {}".format(i))
        i += 1
        start_node = Node(State(read_input(test)), None, None, 0)
        run(algorithm_func, start_node, *args, **kwargs)
        print("----------------------------------------------------------------------------------------------------")
```

الگوریتم‌های تعریف شده در ادامه ذکر شده‌اند.

الگوریتم bfs:

به این صورت عمل شده است که در هر سطح بچه‌های این State که با پیمایش همه‌ی actionها ایجاد می‌شوند، باز می‌شوند و در لیست قرار می‌گیرند. سپس مرحله به مرحله این کار تکرار می‌شود و در صورت رسیدن به حالت نهایی آن را برمیگردانیم.

```
def bfs(start_node: Node):
    states_met = 0
    distinct_states_met = 0
    if start_node.state.is_goal():
        return start_node
    frontier = deque([start_node])
    frontier_state_set = set()
    frontier_state_set.add(start_node.state)
    explored = set()
    while True:
        if not frontier:
            return False
        current = frontier.pop()
        frontier_state_set.remove(current.state)
        explored.add(current.state)
        for action in Direction.actions():
            new_state = current.state.transition(action)
            if new_state is not False:
                states_met += 1
                child = Node(new_state, current, action, current.path_cost + 1)
                if (
                    child.state not in explored
                    and child.state not in frontier_state_set
                ):
                    distinct_states_met += 1
                    if child.state.is_goal():
                        return child
                    frontier.appendleft(child)
                    frontier_state_set.add(child.state)
```

الگوریتم DFS:

این الگوریتم به صورت عمقی همه‌ی Stateها را تا عمق مشخصی (در این تمرین ۴۰ در نظر گرفته شده است) باز می‌کند. در صورتی که به پاسخ نرسیم به عمق عقب‌تر برمی‌گردیم و ادامه‌ی جستجو را انجام می‌دهیم.

```
def dfs(start_node, depth=40):
    states_met = 0
    distinct_states_met = 0
    if start_node.state.is_goal():
        return start_node
    frontier = deque([start_node])
    explored = dict()
    while True:
        if not frontier:
            return False
        current = frontier.pop()
        if current.path_cost == depth:
            continue
        explored[current.state] = current.path_cost
        for action in Direction.actions():
            new_state = current.state.transition(action)
            if new_state is not False:
                states_met += 1
                child = Node(new_state, current, action, current.path_cost + 1)
                if (
                    child.state not in explored
                    or child.path_cost < explored[child.state]
                ):
                    distinct_states_met += 1
                    if child.state.is_goal():
                        return child
                    frontier.append(child)
```

الگوریتم ids:

در این الگوریتم از تابع dfs که تعریف شد استفاده می‌شود و به صورت مرحله به مرحله عمق را اضافه می‌کنیم. به این معنی که اگر در عمق ۱ جواب پیدا نشد حداکثر عمق را ۲ قرار می‌دهیم و همینطور ادامه می‌دهیم.

```
def ids(start_node):
    for depth in range(sys.maxsize):
        current_states = 0
        node = dfs(start_node, depth)
        if node != False:
            return node
    return False
```

الگوریتم A*:

این الگوریتم عملا از یک queue استفاده می‌کند تا stateها را بررسی کند. در این الگوریتم از جمع path_cost و تابع heurisitic استفاده می‌شود تا بتوانیم میان حالت‌ها مقایسه انجام دهیم. از ساختمان

داده heap queue که نوعی priority queue است استفاده شده است تا هر بار بتوانیم کمترین هزینه را دریافت کنیم و گسترش دهیم.

```
def a_star(start_node):
    states_met = 0
    distinct_states_met = 0
    if start_node.state.is_goal():
        return start_node
    frontier = []
    heapq.heappush(
        frontier,
        (
            start_node.path_cost
            + start_node.state.calculate_heuristic(),
            start_node,
        ),
    )
    frontier_state_set = set()
    frontier_state_set.add(start_node.state)
    explored = set()
    while True:
        if not frontier:
            return False
        current = heapq.heappop(frontier)
        frontier_state_set.remove(current[1].state)
        explored.add(current[1].state)
        for action in Direction.actions():
            new_state = current[1].state.transition(action)
            if new_state is not False:
                child = Node(
                    new_state, current[1], action, current[1].path_cost + 1
                )
                states_met += 1
                if (
                    child.state not in explored
                    and child.state not in frontier_state_set
                ):
                    distinct_states_met += 1
                    if child.state.is_goal():
                        return child
                    heapq.heappush(
                        frontier,
                        (
                            child.path_cost
                            + child.state.calculate_heuristic(),
                            child,
                        ),
                    )
                    frontier_state_set.add(child.state)
```

الگوریتم UCS:

این الگوریتم مشابه A* پیاده‌سازی شده است با این تفاوت که از هزینه‌ای که تابع heuristic اعلام می‌کند استفاده نمی‌کند. صرفا path_cost در آن لحاظ شده است.

```python
def ucs(start_node):
    states_met = 0
    distinct_states_met = 0
    if start_node.state.is_goal():
        return start_node
    frontier = []
    heapq.heappush(
        frontier,
        (
            start_node.path_cost,
            start_node,
        ),
    )
    frontier_state_set = set()
    frontier_state_set.add(start_node.state)
    explored = set()
    while True:
        if not frontier:
            return False
        current = heapq.heappop(frontier)
        frontier_state_set.remove(current[1].state)
        explored.add(current[1].state)
        for action in Direction.actions():
            new_state = current[1].state.transition(action)
            if new_state is not False:
                child = Node(
                    new_state, current[1], action, current[1].path_cost + 1
                )
                states_met += 1
                if (
                    child.state not in explored
                    and child.state not in frontier_state_set
                ):
                    distinct_states_met += 1
                    if child.state.is_goal():
                        return child
                    heapq.heappush(
                        frontier,
                        (
                            child.path_cost,
                            child,
                        ),
                    )
                    frontier_state_set.add(child.state)
```

در فایل اکسل نتایج خروجی همه الگوریتم‌ها به ازای هر تست ذکر شده است. در ادامه میانگین زمان اجرا و حافظه استفاده شده برای الگوریتم‌های مختلف مقایسه شده است.

مقایسه زمان اجرا:

در جدول ادامه متوسط زمان اجرا برای الگوریتم‌های مختلف ذکر شده است. همانطور که مشاهده می‌شود الگوریتم A* بهترین عملکرد را داشته است. دلیل این موضوع استفاده‌ی آن از heuristic است که می‌تواند راحت‌تر حالت هدف را نمایان کند. پس از A* الگوریتم BFS بهترین عملکرد را داشته است و پس از آن UCS و DFS زمان اجرایشان از IDS بهتر بوده است. دلیل اینکه الگوریتم IDS

از همه الگوریتم‌ها بدتر عمل کرده است این موضوع است که در حالت‌هایی که جواب در عمق‌های زیاد وجود دارد این الگوریتم باید همه عمق‌های قبل آن را باز کند تا بتواند به آن برسد که زمان اجرای بسیار زیادی میبرد. در فایل اکسل نیز مشخص است که الگوریتم IDS برای تست‌های آسان و متوسط و بعضی تست‌های سخت عملکرد خوبی داشته است. اما در تست‌های با عمق جواب طولانی بسیار بد عمل کرده است که باعث بد شدن میانگین زمان اجرای آن شده است.

الگوریتم	BFS	DFS	IDS	A*	UCS
متوسط زمان اجرا به ثانیه	9.783594843	23.26396871	56.96086837	0.2274738832	11.04157677

مقایسه حافظه استفاده شده:

در جدول زیر حافظه استفاده شده توسط هر الگوریتم به طور متوسط ذکر شده است.

الگوریتم	BFS	DFS	IDS	A*	UCS
متوسط حافظه اجرا به بایت	25,123,298.84	22,987,011.8	13,976,243.53	758,765.6727	25,724,465.85

مجددا الگوریتم A* بهترین عملکرد را داشته است که به نظر می‌آید به دلیل بررسی کردن بهینه حالت‌ها و عدم نگهداری حالت‌های بیش از اندازه است.

سپس IDS بهترین عملکرد را داشته است. الگوریتم‌های UCS، DFS، BFS حدودا میانگین حافظه یکسانی داشته‌اند که به دلیل نحوه باز کردن فرزندان Stateها در آنها می‌تواند باشد.