مجید فریدفر

۸۱۰۱۹۹۵۶۹

گزارشکار تمرین کامپیوتری اول هوش مصنوعی (ترم بهار ۱۴۰۲)

در ابتدا، لازم بود برای نگه داری اطلاعات لازم و همین طور انجام یک سری processهای خاص، تعدادی کلاس تعریف کنیم. در ادامه هر کدام توضیح داده خواهند شد:

۱.یال:

```
class Edge:
    id: int
    nodes = set
    is_loose: bool
    wait_time: int
```

۲.راس

```
class Node:
    id: int
    edges: dict # access to edge via id of the other connected node
    type: NodeType
```

این کلاس، یک متود هم دارد:

```
def add_edge(self, u, edge: Edge):
    self.edges[u] = edge
```

همانطور که مشاهده می‌کنید، edges یک دیکشنری است که به ما اجازه می‌دهد به کامپوننتِ مربوط به یال بین دو راس دلخواه دسترسی داشته باشیم. (با استفاده از این متود می‌توانیم آن را به لیست یال‌های متصل به راس‌های دو سرش اضافه کنیم.

فیلد type هم برای این موضوع است که مشخص کنیم که نوع این راس چیست؟ روی آن یک دانشجو ایستاده؟ یک پیتزافروشی در آن واقع شده؟ یا صرفا یک راس معمولی است؟ این فیلد مقدار زیر را می‌تواند بگیرد:

```
class NodeType(Enum):
    student = 0
    pizza = 1
    normal = 2
```

۳.گراف

```
class Graph:
    num_of_nodes: int
    num_of_edges: int
    nodes: list[Node] # with n elements
    edges: list[Edge]
    student_pizza: dict # access to id of the pizza node which a node wants (by their
id)
    pizza_student: dict # access to id of the student node
    loose_edges: set[Edge]
    ziraj_first_position: int
    priorities: list # access to prority of each node
    students: set[Node]
    pizzas: set[Edge]
```

بیشتر فیلدها واضحند. در ادامه بعضی از آن‌ها را شرح می‌دهیم:

فیلد student_pizza: یک دیکشنری است که می‌توانیم با استفاده از آن بفهمیم که هر دانشجویی از کدام پیتزافروشی، پیتزا می‌خواهد.

فیلد pizza_student: یک دیکشنری است که مشخص می‌کند که از یک پیتزافروشی، چه دانشجویی پیتزا سفارش داده است.

این دو فیلد را در متود زیر پر می‌کنیم:

```
def assign_pizza_and_student(self, student, pizza):
     self.nodes[student].type = NodeType.student
     elf.students.add(student)
     self.nodes[pizza].type = NodeType.pizza
     self.pizzas.add(pizza)

     self.student_pizza[student] = pizza
     self.pizza_student[pizza] = student
```

این متود، یک سفارش را می‌گیرد، و و این دو فیلد را پر می‌کند. همچنین type راس‌های دانشجو و پیتزافروشی را درست می‌کند. و آن‌ها را در مجموعه‌های دانشجو-پیتزا قرار می‌دهد.

فیلد priorities: یک لیست است، که می‌توانیم با استفاده از آن می‌توانیم بفهمیم که که به ازای یک راس دلخواه، کدام راس‌ها نسبت به آن اولویت دارند. در متود زیر، این مقادیر را پر می‌کنیم:

```
def add_prority(self, u, v):
     self.priorities[v].append(u)
```

همچنین دوتا متود دیگر هم داریم که تقریبا کارکردشان واضح است: add_edge (اضافه کردن یک یال به لیست یال‌ها و همین طور به لیست یال‌های دو سرش) و change_edge_to_loose (با استفاده از این متود یک یال را تبدیل به یال لق می‌کنیم).

۴.استیت

```
class State:
    ziraj_current_position: int
    loose_edges_last_pass_time: dict # access to last pass time of each loose edge by its id
    students_with_pizza: set # set of id of studesnt who delivered their pizza
    delivered_pizza: set # set of id pizzas wiche are delivered
    path: list # list of id of nodes on the crrent path
    passed_time: int
    holded_pizza: int
    stay_on_node: int
```

این کلاس، تقریبا اصلی‌ترین کلاس مسئله است. یک آبجکت از این کلاس، یک لحظه از جهان را نشان می‌دهد. مواردی که برایمان در آن لحظه مهم است:

- مکان کنونی زیرج (در ziraj_current_position)
- مجموعه‌ی دانشجوهایی که پیتزایشان را دریافت کرده‌اند (در students_with_pizza)
- مجموعه‌ی پیتزاهایی که تحویل داده شده‌اند (در delivered_pizza)
- مسیری که زیرج از لحظه‌ی اول تا اینجا آمده (در path). لازم به ذکر است که اگر زیرج چندبار در راس یک راس بماند و حرکتی نکند، توی path به همان تعداد، از آن راس پشت سر هم داریم.
- پیتزایی که در دست زیرج است و آن را حمل می‌کند (در holded_pizza)، اگر پیتزایی در دستش نبود، این مقدار None است.
- زمان گذشته تا این لحظه (passed_time)
- مدت زمانی که روی راس کنونی ایستاده (در stay_on_node)
- و یک دیکشنری از یال‌های لقی که اجازه‌ی گذر از آن‌ها را نداریم (loose_edges_last_pass_time). از طریق این دیکشنری می‌توانیم بفهمیم، چند مرحله‌ی دیگر می‌توانیم از روی آن رد بشویم. لازم به ذکر است، یال‌های لقی که می‌توانیم از روی آن‌ها رد بشویم، در این دیکشنری قرار ندارند.

همچنین این کلاس تعدادی متود هم دارد:

```
def update_pass_time_of_loose_edges(self):
    for loose_edge in list(self.loose_edges_last_pass_time.keys()):
        self.loose_edges_last_pass_time[loose_edge] -= 1
        if self.loose_edges_last_pass_time[loose_edge] == 0:
            del self.loose_edges_last_pass_time[loose_edge]
```

زمانی که از این متود استفاده می‌کنیم، که یک مرحله گذشته باشد. حالا یک روز از تمام روزهای باقی مانده برای انتظارِ هر یال، کم می‌شود. همچنین اگر این مقدار برای یک یال، صفر بشود آن را از دیکشنری حذف می‌کنیم. چون حالا می‌توانیم از روی آن گذر کنیم.

```
def __eq__(self, s: object):
    return (
        self.ziraj_current_position == s.ziraj_current_position and
        self.stay_on_node == s.stay_on_node and
        self.delivered_pizza == s.delivered_pizza and
        self.holded_pizza == s.holded_pizza
    )
```

این متود، برای مقایسه‌ی دو استیت به کار می‌روند که در ادامه به آن نیاز داریم. برای این که بررسی کنیم آیا دو استیت باهم برابرند، چهار فیلد زیر را از هر کدام با هم مقایسه می‌کنیم:

- مکان کنونی زیرج در دو استیت
- دانشجوهایی که در این دو استیت، سفارشان را دریافت کرده‌اند.
- پیتزاهایی که دو استیت، دلیور شده‌اند.
- مدت زمانی که زیرج روی راس کنونی‌اش صبر کرده.

متودهای __str__ و __hash__ در حل مسئله کاربرد خاصی ندارند.

لازم به ذکر است که استیت اولیه به این شکل است:

```
init_state = State(graph.ziraj_first_position)
```

کانسراتکتور کلاس:

```
def __init__(self, ziraj_position):
    self.ziraj_current_position = ziraj_position
    self.loose_edges_last_pass_time = dict()
    self.students_with_pizza = set()
    self.delivered_pizza = set()
    self.path = [ziraj_position]
    self.passed_time = 0
    self.holded_pizza = None
    self.stay_on_node = 0
```

در اینجا set()ها و dict()ها یعنی در ابتدا، مثلا مجموعه‌ی دانشجوهایی که پیتزایشان را دریافت کرده‌اند، خالی است (یا پیتزاهایی که به دانشجوی مورد نظر رسیده است). همین طور در ابتدا، می‌توانیم از روی همه یال‌های لق رد بشویم. بقیه موارد تقریبا واضحند.

همچنین، استیتی به عنوان goal_state در نظر گرفته می‌شود، که در آن همه‌ی دانشجوها پیتزایشان را دریافت کرده باشند. یا به عبارتی دیگر (در همه جای کد، این statement، استفاده می‌شود):

```
len(State.delivered_pizza) == num_of_orders
```

حالا یک تابع می‌نویسیم تا به اطلاعات موجود در فایل‌ها را بخوانیم و گراف مدنظر را تولید کنیم:

```
def read_input(i):
    f = open("tests/Test"+str(i)+".txt", 'r', encoding='utf-8')

    n, m = [int(x) for x in f.readline().split()]

    graph = Graph(n, m)

    for i in range(m):
        u, v = [int(x) for x in f.readline().split()]
        graph.add_edge(i, v-1, u-1)

    num_of_loose_edges = int(f.readline())

    for i in range(num_of_loose_edges):
        id, xi = [int(x) for x in f.readline().split()]
        graph.change_edge_to_loose(id-1, xi)

    graph.ziraj_first_position = int(f.readline())-1
    num_of_students = int(f.readline())

    students = []

    for i in range(num_of_students):
        student, pizza = [int(x) for x in f.readline().split()]
        graph.assign_pizza_and_student(student-1, pizza-1)
        students.append(student-1)

    num_of_priority_rules = int(f.readline())

    for i in range(num_of_priority_rules):
        a, b = [int(x) for x in f.readline().split()]
        graph.add_prority(students[a-1], students[b-1])

    f.close()
    return graph
```

- دیتاها در فولدر tests/ قرار دارند و فرمت اسم آن‌ها به این شکل است: Test#Num.txt. مثلا Test1.txt.
- در اینجا یک لیست students هم تعریف کرده‌ایم. از طریق آن می‌توانیم بفهمیم که دانشجوی شماره‌ی i در چه راسی است. که می‌شود: students[i-1]. از آن برای اضافه کردن اولویت‌ها استفاده می‌کنیم. در نهایت، گراف تولید شده را برمی‌گردانیم.

```
graph = read_input(int(input("Enter number of test file: ")))
num_of_orders = len(graph.student_pizza.keys())
```

در اینجا شماره‌ی فایل را از terminal می‌خوانیم. و پس از ساخته شدن گراف، تعداد سفارشات را در num_of_orders می‌ریزیم (که با تعداد دانشجوها برابر است). از آن بعدا برای چک کردن این که یک استیت، goal state است یا استفاده می‌کنیم (همان طور که پیش‌تر گفته شد).

و سپس استیت اولیه را می‌سازیم (که خاصیت‌هایش پیش‌تر گفته شده را داراست):

```
init_state = State(graph.ziraj_first_position)
```

دقت کنید در کل برنامه، مقادیر graph و init_state به صورت جهانی در تمامی تابع‌ها استفاده می‌شوند (برای جلوگیری از کپی شدن و هدر رفت زمان).

حالا یک تابع مهم دیگر داریم: get_all_next_states. کار این تابع این است که در ورودی، یک استیت بگیرد و پس از انجام یک سری کارها، لیستی از استیت‌های دیگر که از استیت کنونی می‌توان به آن‌ها رفت را برگرمی‌داند.

در ابتدا نیاز است که actionهای مختلفی که زیرج در هر محله می‌تواند انجام بدهد را بدانیم:

- هیچ کاری نکردن! (یعنی روی همان راس می‌ماند).
- رفتن به یکی از رئوس همسایه
- اگر در راس کنونی، پیتزافروشی بود، می‌تواند از آنجا پیتزا بگیرد، یا نگیرد!
- اگر راس کنونی، دانشجو بود و زیرج داشت پیتزای او را حمل می‌کرد، پیتزا را به او تحویل می‌دهد.

الگوریتم انجام این کار:

راس فعلی زیرج را می‌گیریم و برای تمام راس‌های همسایه‌اش در گراف، این کارها را انجام می‌دهیم:

یک کپی از state کنونی می‌گیریم و در next_state می‌ریزیم. چون عملا این استیت، باید مرحله‌ی بعدی باشد، passed_timeاش را یک واحد زیاد می‌کنیم و روی آن متود update_pass_time_for_loose_edges را صدا می‌زنیم (عملکرد این متود پیشتر توضیح داده شد). حالا یالی که بین زیرج و این همسایه است را بررسی می‌کنیم. اگر یال لق بود و نمی‌توانستیم از روی آن رد بشویم، (این موضوع را با شرط زیر چک می‌کنیم:

```
if edge.is_loose:
        if edge.id in current_state.loose_edges_last_pass_time.keys():
```

)، زیرج نمی‌تواند روی این یال حرکت کند و باید صبر کند. پس این کارها را انجام می‌دهیم:

```
next_state.path.append(ziraj.id)
next_state.stay_on_node += 1
```

و next_state را به لیست اضافه کردیم و سراغ همسایه‌ی بعدی می‌رویم. (چون لازم نیست کار دیگری بکنیم).

ولی اگر لق بود و می‌توانستیم از روی آن رد بشویم، در مراحل بعدی، دیگر نمی‌توانیم از روی آن عبور کنیم، و باید صبر کنیم. برای همین این یال را به دیکشنری loose_edges_last_pass_time اضافه کرده و مقدار wait_time اولیه‌اش را در آن می‌ریزیم:

```
next_state.loose_edges_last_pass_time[edge.id] = graph.edges[edge.id].wait_time
```

(اگر یال لق نبود که هیچی، یک راست سراغ مراحل بعدی می‌رویم).

حالا مسیر و مکان کنونی زیرج را آپدیت می‌کنیم و stay_on_node زیرج را صفر می‌کنیم (چون تازه به این راس آمده):

```
next_state.ziraj_current_position = neighbor
next_state.path.append(neighbor)
next_state.stay_on_node = 0
```

حالا، اگر دست زیرج پیتزا بود، چک می‌کنیم که آیا راس کنونی یک دانشجو است یا نه؟ اگر بود، بررسی می‌کنیم که همین پیتزایی که دست زیرج هست را می‌خواهد یا نه؟ اگر می‌خواست پیتزا را به او تحویل می‌دهیم:

```
next_state.students_with_pizza.add(neighbor)
next_state.delivered_pizza.add(next_state.holded_pizza)
next_state.holded_pizza = None
```

(بررسی کردن این شرط هم به این شکل است:

```
if next_state.holded_pizza != None: # ziraj holds a pizza
    if graph.nodes[neighbor].type == NodeType.student and
graph.student_pizza[neighbor] == next_state.holded_pizza:
```

). در غیر این صورت، یعنی اگر در دست زیرج، پیتزایی نبود، بررسی می‌کنیم که آیا راس کنونی یک پیتزافروشی است یا نه؟ اگر هست آیا قبلا پیتزایش تحویل داده شده؟ اگر نشده، با توجه به مسائل اولویت می‌تواند آن را بردارد؟ اگر می‌توانست، آن را برمی‌دارد (در اینجا از خاصیت else داشتن for استفاده می‌کنیم):

```
if next_state.holded_pizza == None:
    if graph.nodes[neighbor].type == NodeType.pizza and not neighbor in
next_state.delivered_pizza:
        for student in graph.priorities[graph.pizza_student[neighbor]]:
            if not student in next_state.students_with_pizza:
                break
        else:
            next_state.holded_pizza = neighbor
```

در نهایت next_state را به لیست اضافه کرده و سراغ همسایه‌ی بعدی می‌رویم.

یک تابع ساده هم داریم برای چاپ کردن اطلاعات به اسم print_answer که از توضیح آن صرف نظر می‌کنیم.

BFS

الگوریتم:

یک مجموعه‌ی frontier داریم (که حکم صف -queue– دارد) از راس‌های مرزی، که در ابتدا، init_state در آن قرار دارد. و یک مجموعه‌ی explored از راس‌های پردازش شده که در ابتدا خالی است.

تا جایی که می‌شود، از سر صف frontier برمی‌داریم و در current_state می‌ریزیم. سپس، فرزندانش را (در صورتی که قبلا پردازش نشده باشند و همین طور در مجموعه‌ی مرزی موجود نباشند) به ته صف اضافه می‌کنیم. همچنین اگر یکی از فرزندان، goal state بود، کار به پایان می‌رسد (از statementای که پیش‌تر گفته شد، برای بررسی این موضوع استفاده می‌کنیم). در نهایت، current_state را به مجموعه‌ی explored اضافه می‌کنیم.

```
def bfs():
    frontier = [init_state] # as a queue
    explored = set()
    visited_states = 1

    while len(frontier) != 0:
        current_state = frontier.pop(0)

        for next_state in get_all_next_states(current_state):
            if (next_state in frontier) or (next_state in explored):
                continue

            visited_states += 1

            if len(next_state.delivered_pizza) == num_of_orders:
                return next_state.path, visited_states

            frontier.append(next_state)

        explored.add(current_state)

    return None, None
```

در اینجا، یک متغیر visited_states هم داریم که شمار استیت‌هایی است که پردازششان می‌کنیم. در انتها، این مقدار را هم همراه با مسیر به دست آمده از راس‌ها (برای رسیدن به goal state) برمی‌گردانیم.

در ادامه، نتایج اجرای کد را می‌بینیم (هر تست سه بار اجرا شده است و همین طور از محیط Google Colab استفاده شده است):

Test1.txt:

```
start_time = time.time()
answer, visited_states = bfs()
# answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 1
Best time: 10
Suggested Path: 6 4 10 5 3 8 9 1 7 9 2
Run Time: 0.028302669525146484
Number of Visited States: 231
```

```
start_time = time.time()
answer, visited_states = bfs()
# answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 1
Best time: 10
Suggested Path: 6 4 10 5 3 8 9 1 7 9 2
Run Time: 0.0277161598205664
Number of Visited States: 231
```

```
start_time = time.time()
answer, visited_states = bfs()
# answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 1
Best time: 10
Suggested Path: 6 4 10 5 3 8 9 1 7 9 2
Run Time: 0.02758312225341797
Number of Visited States: 231
```

مشاهده می‌شود که بهترین زمان برای رساندن تمام سفارشات، ۱۰ است. و زیرج باید مسیر گفته شده را طی بکند. همچنین زمان اجرای الگوریتم هم با محدودیت زمانی تعیین شده مطابقت دارد (کم‌تر از ۰٫۱ ثانیه است).

Test2.txt:

```
start_time = time.time()
answer, visited_states = bfs()
# answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 2
Best time: 20
Suggested Path: 14 9 4 15 12 3 12 6 14 1 14 12 3 8 6 4 10 11 8 3 13
Run Time: 0.2496178150177002
Number of Visited States: 966
```

```
start_time = time.time()
answer, visited_states = bfs()
# answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 2
Best time: 20
Suggested Path: 14 9 4 15 12 3 12 6 14 1 14 12 3 8 6 4 10 11 8 3 13
Run Time: 0.2946610450744629
Number of Visited States: 966
```

```
start_time = time.time()
answer, visited_states = bfs()
# answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 2
Best time: 20
Suggested Path: 14 9 4 15 12 3 12 6 14 1 14 12 3 8 6 4 10 11 8 3 13
Run Time: 0.24463891983032227
Number of Visited States: 966
```

بهترین زمان ممکن برای رساندن تمام سفارشات: ۲۰. مسیر در شکل‌ها موجود است و همین طور زمان اجرای الگوریتم هم با محدودیت زمانی تعیین شده مطابقت دارد (کم‌تر از ۲ ثانیه است).

Test3.txt:

```
start_time = time.time()
answer, visited_states = bfs()
# answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 3
Best time: 31
Suggested Path: 15 14 8 20 13 8 18 10 8 9 16 5 15 12 3 19 11 19 13 6 13 1 13 8 17 8 18 2 18 4 14 16
Run Time: 12.482389211654663
Number of Visited States: 18622
```

```
start_time = time.time()
answer, visited_states = bfs()
# answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 3
Best time: 31
Suggested Path: 15 14 8 20 13 8 18 10 8 9 16 5 15 12 3 19 11 19 13 6 13 1 13 8 17 8 18 2 18 4 14 16
Run Time: 12.452548027038574
Number of Visited States: 18622
```

```
start_time = time.time()
answer, visited_states = bfs()
# answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 3
Best time: 31
Suggested Path: 15 14 8 20 13 8 18 10 8 9 16 5 15 12 3 19 11 19 13 6 13 1 13 8 17 8 18 2 18 4 14 16
Run Time: 13.381192445755005
Number of Visited States: 18622
```

بهترین زمان ممکن برای رساندن تمام سفارشات: ۳۱. مسیر در شکل‌ها موجود است و همین طور زمان اجرای الگوریتم هم با محدودیت زمانی تعیین شده مطابقت دارد (کم‌تر از ۲۰ ثانیه است).

IDS

برای استفاده از این روش، ابتدا یک روش دیگر را پیاده سازی می‌کنیم: Depth Limited Search.

الگوریتم:

یک مجموعه‌ی frontier داریم (که حکم استک -stack– دارد) از راس‌های مرزی، که در ابتدا، init_state در آن قرار دارد. و یک دیکشنری به نام explored از راس‌های پردازش شده. از روی این دیکشنری، می‌توانیم بفهمیم که هر راسی، در چه عمقی، دیده و پردازش شده بود (واضح است که این مقدار برابر است با passed_time راس) که در ابتدا خالی است.

تا جایی که می‌شود، از سر استک frontier برمی‌داریم و در current_state می‌ریزیم. سپس، فرزندانش را روی استک می‌گذاریم به شرطی که:

- در مجموعه‌ی مرزی نباشند، یعنی قبل به استک اضافه نشده باشند.
- قبلا در عمق کم‌تری، پردازش نشده باشند.
- مقدار passed_timeشان کم تر از l باشد (یعنی حد تعیین شده، رعایت شده باشد).
- یک goal state نباشد. در این صورت، کار تمام است و مسیر به دست آمده را برمی‌گردانیم.

در نهایت current_path را به دیکشنری اضافه می‌کنیم و زمان دیده شدنش را برابر با مقدار depth قرار می‌دهیم. (ممکن است این نود قبلا به دیکشنری اضافه شده باشد! مشکلی نیست. در این صورت کافی است این مقدار را تغییر دهیم).

```
def depth_limited_search(l):
    frontier: list[State] = [init_state] # as a stack
    explored_at: dict[State, int] = dict()
    visited_states = 1

    while len(frontier) != 0:
        current_state = frontier.pop()
        depth = current_state.passed_time

        for next_state in get_all_next_states(current_state):
            if (next_state in frontier) or (next_state in explored_at.keys() and
explored_at[next_state] <= depth+1):
                continue

            if next_state.passed_time >= l:
                continue

            visited_states += 1

            if len(next_state.delivered_pizza) == num_of_orders:
                return next_state.path, visited_states

            frontier.append(next_state)

        explored_at[current_state] = depth

    return None, visited_states
```

در اینجا هم مثل bfs، همان متغیر visited_stets را هم داریم. (توضیحات مشابه است) که به همراه مسیر برگردانده می‌شود.

حالا تابع ids را می‌نویسیم، که از این تابعِ گفته شده استفاده می‌کند:

```
def ids():
    l = 0
    visited_states = 0

    while True:
        path, more_visited_states = depth_limited_search(l)
        visited_states += more_visited_states

        if path != None:
            return path, visited_states
        else:
            l += 1
```

همان طور که مشاهده می‌شود، این تابع، به ترتیب در عمق‌های مختلف (که قدم به قدم زیاد می‌شود)، الگوریتم depth_limited_search را اجرا می‌کند. تا جایی که برای اولین بار به goal state برسیم. این جا کار تمام است و مسیر به دست آمده را به همراه تعداد استیت‌های پردازش شده برمی‌گردانیم.

در ادامه، نتایج اجرای کد را می‌بینیم:

Test1.txt:

```
start_time = time.time()
# answer, visited_states = bfs()
answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 1
Best time: 10
Suggested Path: 6 4 10 9 1 7 9 2 8 3 8
Run Time: 0.2023026943206787
Number of Visited States: 1485
```

```
start_time = time.time()
# answer, visited_states = bfs()
answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 1
Best time: 10
Suggested Path: 6 4 10 9 1 7 9 2 8 3 8
Run Time: 0.18378996849060059
Number of Visited States: 1485
```

```
start_time = time.time()
# answer, visited_states = bfs()
answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 1
Best time: 10
Suggested Path: 6 4 10 9 1 7 9 2 8 3 8
Run Time: 0.18443918228149414
Number of Visited States: 1485
```

مشاهده می‌شود که جواب (بهترین زمان و مسیر داده شده) با جواب bfs برابر است ولی تعداد استیت‌های پردازش شده خیلی بیشتر است. این موضوع باعث شده که زمان اجرای الگوریتم هم زیادتر بشود. اما همچنان با محدودیت زمانی داده شده مطابقت دارد (کم‌تر از ۰٫۵ ثانیه).

Test2.txt:

```
start_time = time.time()
# answer, visited_states = bfs()
answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 2
Best time: 20
Suggested Path: 14 9 4 15 4 7 14 1 14 12 3 8 6 8 6 4 10 11 5 3 13
Run Time: 2.8476531505584717
Number of Visited States: 17711
```

```
start_time = time.time()
# answer, visited_states = bfs()
answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 2
Best time: 20
Suggested Path: 14 9 4 15 4 7 14 1 14 12 3 8 6 8 6 4 10 11 5 3 13
Run Time: 3.4803173542022705
Number of Visited States: 17711
```

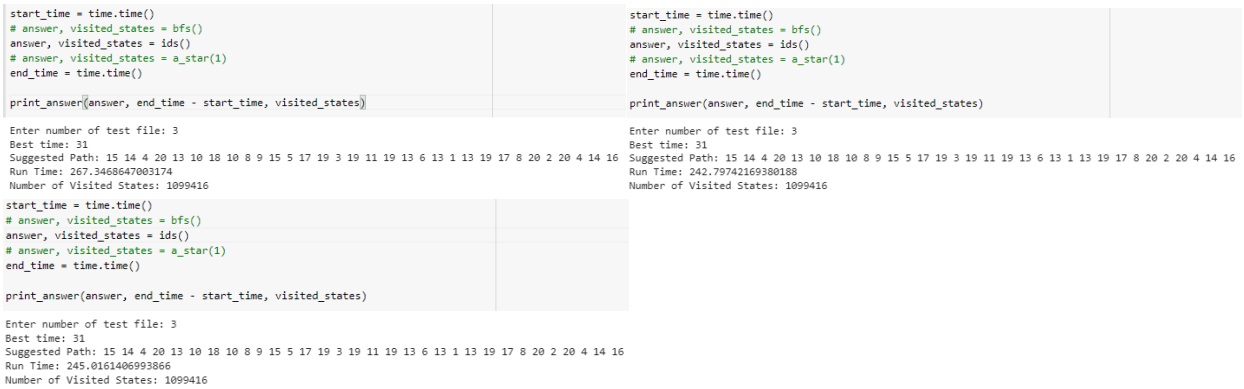
```
start_time = time.time()
# answer, visited_states = bfs()
answer, visited_states = ids()
# answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 2
Best time: 20
Suggested Path: 14 9 4 15 4 7 14 1 14 12 3 8 6 8 6 4 10 11 5 3 13
Run Time: 2.656519651412964
Number of Visited States: 17711
```

جواب داده شده مثل bfs است، ولی تعداد استیت‌های دیده شده و همچنین زمان اجرا خیلی بیشتر (ولی کم‌تر از ۶۰ ثانیه).

Test3.txt:



نتایج به دست آمده، مشابه تست یک و دو است.

A* and Weighted A*

برا این الگوریتم، ابتدا باید یک هیوریستیک در نظر بگیریم. هیوریستیکی که من در نظر گرفتم این است:

تعداد دانشجوهای بی‌غذا + **تعداد پیتزاهای تحویل داده نشده**

هم‌چنین اگر زیرج در حال حمل پیتزا بود، **یک واحد از آن کم می‌کنیم**.

```
self.heuristic = (2 * num_of_orders) - (2 * len(self.state.delivered_pizza))

    if s.holded_pizza != None:
        self.heuristic -= 1
```

نکته‌ای که اینجا وجود دارد، این است که تعداد دانشجوهای بی‌غذا با تعداد پیتزاهای تحویل داده نشده برابر است (هر داشنجو = یک پیتزا!). پس می‌توانیم این مقدار را به شکل بالا حساب کنیم (تعداد پیتزاهای داده نشده = تعداد کل پیتزاها – تعداد پیتزاهای داده شده).

این هیوریستیک، یک هیوریستیک **کانسیستنت** است. به این علت که به ازای هر اکشنی که زیرج انجام می‌دهد (که هزینه‌اش برابر با یک یا صفر است)، مقدار هیوریستیک استیت شروع، حداکثر یک واحد کم می‌شود. اکشن‌های مختلف زیرج را در نظر بگیرید:

- حرکت نکردن – برنداشتن پیتزا:
  - هزینه‌ی واقعی: ۰ – اختلاف هیوریستیک کنونی و مرحله‌ی بعدی: ۰ (تغییری نمی‌کند)
- رفتن به یال بعدی:
  - هزینه‌ی واقعی: ۱ – اختلاف هیوریستیک کنونی و مرحله‌ی بعدی: ۰
- برداشتن پیتزا:
  - هزینه‌ی واقعی: ۱ – اختلاف هیوریستیک کنونی و مرحله‌ی بعدی: ۱
- تحویل دادن پیتزا:
  - هزینه‌ی واقعی: ۱ – اختلاف هیوریستیک کنونی و مرحله‌ی بعدی: ۱-۱+۱ = ۱

پس شرط در تمام حالات برقرار است.

حالا یک کلاس دیگر تعریف می‌کنیم:

```
class HeapNode:
    state: State
    heuristic: int
    f: int
```

که سه تا فیلد دارد:

- استیت
- هیوریستیک این استیت (که به مدل گفته شده در بالا تولید می‌شود).
- مقدار $f(x) = g(x) + \alpha h(x)$. مقدار h را که در یک فیلد دیگر داریم. مقدار g هم برابر است مقدار زمان گذشته تا اینجا در استیت.

```
self.f = alpha*self.heuristic + self.state.passed_time
```

این کلاس دو تا متود هم دارد که به آن‌ها نیاز داریم:

```
def __lt__(self, s):
    return self.f < s.f

def __eq__(self, s: State):
    return self.state == s
```

برای مقایسه‌ی دو تا HeapNode یا مقایسه‌ی یک HeapNode و یک State.

همین طور که از اسم این کلاس هم مشخص است، قرار است، که از این آبجکت‌ها در Heap استفاده کنیم.

الگوریتم *A:

یک مجموعه‌ی frontier داریم (که حکم مین هیپ -min heap– دارد) از راس‌های مرزی، که در ابتدا، init_state در آن قرار دارد. و یک مجموعه به نام explored از راس‌های پردازش شده که در ابتدا خالی است.

تا جایی که می‌شود، کوچیک‌ترین عنصر frontier را برمی‌داریم و در current_state می‌ریزیم (منظور از کوچک‌ترین عنصر frontier، عنصری است که مقدار f(x) کم‌تری دارد. فلسفه‌ی متود __lt__ در HeapNode هم همین است). سپس، فرزندانش را در به شرطی که در مین هیپ و مجموعه‌ی explored نباشند، به مین هیپ اضافه می‌کنیم. البته اگر در این بین، یکی از فرزندان، goal state بود، کار تمام می‌شود و مسیر به دست آمده را برمی‌گردانیم.

در نهایت current_path را به مجموعه‌ی explored اضافه می‌کنیم و دوباره همین کارها را انجام می‌دهیم. (در این جا هم مثل دو تا متود قبلی، باید متغیر visited_states را هم در هر مرحله مقداردهی کنیم).

```python
def a_star(alpha = 1):
    frontier = [] # as a min heap
    explored = set()

    heapq.heappush(frontier, HeapNode(init_state, alpha))
    visited_states = 1

    while len(frontier) != 0:
        current_state = heapq.heappop(frontier).state

        for next_state in get_all_next_states(current_state):
            if next_state in explored or next_state in frontier:
                continue

            visited_states += 1

            if len(next_state.delivered_pizza) == num_of_orders:
                return next_state.path, visited_states

            heapq.heappush(frontier, HeapNode(next_state, alpha))

        explored.add(current_state)

    return None, None
```

- از این تابع به عنوان weighted A* هم استفاده می‌کنیم. همان طور که مشاهده می‌کنید، مقدار alpha به صورت پیش‌فرض برابر با ۱ است. این یعنی A*. اما اگر به آن مقدار بدهیم، می‌توانیم الگوریتم Weighted A* را تولید کنیم. به ازای alphaی دلخواه.

در ادامه، نتایج اجرای کد را می‌بینیم (برای alpha = 1):

Test1.txt:

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 1
Best time: 10
Suggested Path: 6 4 10 9 1 7 9 2 8 3 8
Run Time: 0.016965627670288086
Number of Visited States: 137
```

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 1
Best time: 10
Suggested Path: 6 4 10 9 1 7 9 2 8 3 8
Run Time: 0.01365208625793457
Number of Visited States: 137
```

مشاهده می‌شود جواب به دست آمده، مشابه bfs و ids است. ولی مدت زمان اجرای الگوریتم و همین طور استیت‌های مشاهده شده کاهش یافته است. (زمان کم‌تر از ۰٫۱ ثانیه است).

Test2.txt:

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 2
Best time: 20
Suggested Path: 14 9 4 15 12 14 1 14 6 12 14 11 5 2 13 3 12 6 8 6 4
Run Time: 0.12792706489562988
Number of Visited States: 818
```

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 2
Best time: 20
Suggested Path: 14 9 4 15 12 14 1 14 6 12 14 11 5 2 13 3 12 6 8 6 4
Run Time: 0.1224679946899414
Number of Visited States: 818
```

Test3.txt:

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 3
Best time: 31
Suggested Path: 15 14 4 20 13 10 18 10 8 9 17 5 16 12 3 19 11 19 13 6 13 1 13 19 17 8 20 2 18 4 14 16
Run Time: 13.493613243103027
Number of Visited States: 15406
```

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 3
Best time: 31
Suggested Path: 15 14 4 20 13 10 18 10 8 9 17 5 16 12 3 19 11 19 13 6 13 1 13 19 17 8 20 2 18 4 14 16
Run Time: 12.765375137329102
Number of Visited States: 15406
```

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(1)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 3
Best time: 31
Suggested Path: 15 14 4 20 13 10 18 10 8 9 17 5 16 12 3 19 11 19 13 6 13 1 13 19 17 8 20 2 18 4 14 16
Run Time: 13.458417415618896
Number of Visited States: 15406
```

برای آلفا برابر با ۲:

Test1.txt:

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(2)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 1
Best time: 10
Suggested Path: 6 4 10 9 1 3 8 9 7 9 2
Run Time: 0.003293752670288086
Number of Visited States: 45
```

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(2)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 1
Best time: 10
Suggested Path: 6 4 10 9 1 3 8 9 7 9 2
Run Time: 0.0020473003387451117
Number of Visited States: 45
```

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(2)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 1
Best time: 10
Suggested Path: 6 4 10 9 1 3 8 9 7 9 2
Run Time: 0.0030066967010498047
Number of Visited States: 45
```

تعداد استیت‌های پردازش شده به طرز فاحشی کاهش یافته است.

Test2.txt:

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(2)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 2
Best time: 21
Suggested Path: 14 9 4 15 12 3 12 6 8 6 4 7 14 1 14 12 3 5 11 10 4 13
Run Time: 0.046178579330444336
Number of Visited States: 396
```

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(2)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 2
Best time: 21
Suggested Path: 14 9 4 15 12 3 12 6 8 6 4 7 14 1 14 12 3 5 11 10 4 13
Run Time: 0.04797649383544922
Number of Visited States: 396
```

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(2)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 2
Best time: 21
Suggested Path: 14 9 4 15 12 3 12 6 8 6 4 7 14 1 14 12 3 5 11 10 4 13
Run Time: 0.04442286491394043
Number of Visited States: 396
```

در این تست، جواب درست نیست (با bfs و ids که جواب های درست را می‌دهند مغایرت دارد)، اما جواب در زمان خیلی کم‌تری داده شده و هم چنین تعداد استیت‌های بررسی شده خیلی کم‌تر است. این موضوع یک تریدآف است بین درستی جواب و زمان اجرای الگوریتم. البته این میزان اشتباه بودن جواب خیلی زیاد نیست (در حد یک استیت بیشتر).

Test3.txt:

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(2)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 3
Best time: 31
Suggested Path: 15 14 4 20 13 8 18 10 8 9 15 5 17 19 3 19 11 19 13 6 13 1 13 19 17 8 18 2 18 4 14 16
Run Time: 0.9547266960144043
Number of Visited States: 2757
```

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(2)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 3
Best time: 31
Suggested Path: 15 14 4 20 13 8 18 10 8 9 15 5 17 19 3 19 11 19 13 6 13 1 13 19 17 8 18 2 18 4 14 16
Run Time: 0.9505980014801025
Number of Visited States: 2757
```

```
start_time = time.time()
# answer, visited_states = bfs()
# answer, visited_states = ids()
answer, visited_states = a_star(2)
end_time = time.time()

print_answer(answer, end_time - start_time, visited_states)
```

```
Enter number of test file: 3
Best time: 31
Suggested Path: 15 14 4 20 13 8 18 10 8 9 15 5 17 19 3 19 11 19 13 6 13 1 13 19 17 8 18 2 18 4 14 16
Run Time: 0.8711438179016113
Number of Visited States: 2757
```

میزان سریع جواب دادن را می‌توان در این تست فهمید! (جواب هم درست است).

حالا با داده‌هایی که داریم، جدول را پر می‌کنیم:

Test1.txt:

	پاسخ	تعداد استیت	زمان اجرا
BFS	10	231	0.027
IDS	10	1485	0.189
A* w=1	10	137	0.015
A* w=2	10	45	0.003

Test2.txt:

	پاسخ	تعداد استیت	زمان اجرا
BFS	20	966	0.265
IDS	20	17711	2.9
A* w=1	20	818	0.124
A* w=2	21	396	0.046

Test3.txt:

	پاسخ	تعداد استیت	زمان اجرا
BFS	31	18622	12.6
IDS	31	1099416	250
A* w=1	31	15406	13.2
A* w=2	31	2757	0.9

کدام الگوریتم ها جواب بهینه می‌دهند؟ الگوریتم‌های bfs و ids و A* (با آلفای یک و هیوریستیک کانسیستنت).

این الگوریتم‌ها را از لحظا سرعت اجرا مقایسه کنید. الگوریتم bfs سریع تر از ids به جواب میرسد (چون ids مرحله به مرحله جلو میرود. اگر به ازای یک ارتفاع خاص به goal state نرسید، از اول همان کارها را انجام میدهد. به همین علت استیت های بیشتری میبیند و بیشتر طول میکشد). الگوریتم های A* و Weighted A* از هر دو الگوریتم های BFS و IDS سریع تر به جواب میرسیند (چون به آینده هم نگاه میکنند و قدم های هدفمندتری برمیدارند). اما ممکن است جوابشان اشتباه باشد (مثل تست دوم Weighted A*). همچنین با آلفای بیشتر از یک، Weighted A* سریع تر از A* است.

نکته: در مواقعی که درستی جواب به طور دقیق برایمان آنچنان اهمیتی ندارد، یعنی با مقداری خطا میتوانیم یک جواب غیر درست را بپذیریم، و میخواهیم سریعتر به جواب برسیم، بهتر است از A* با Weighted A* استفاده کنیم. اما اگر درستی جواب برایمان مهم است، بهتر است از BFS استفاده کنیم.

نکته: از معایب BFS نسبت به IDS این است که فضای بیشتری اشغال میکند. در اینجا هم ترید آف بین زمان و حافظه وجود دارد بین BFS و IDS.

نکته: همچنین هرچقدر که مقدار آلفا را بیشتر کنیم، ممکن است به goal state نرسیم. پس باید به این موضوع هم دقت کنیم که نباید خیلی زیاد کنیم مقدار آلفا را.