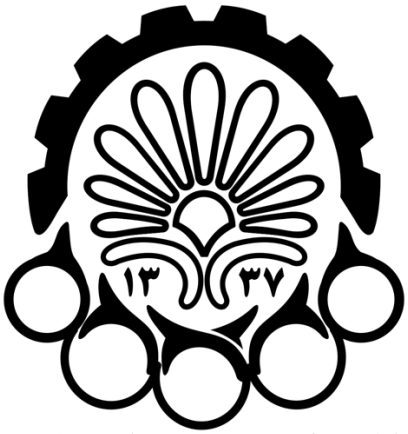
**به نام خدا**




**دانشگاه صنعتی امیرکبیر**

**( پلی تکنیک تهران )**

# پروژه‌ی اول درس هوش مصنوعی و کاربردهای آن: جستجو

**استاد درس: دکتر بهنام روشن‌فکر**

دانیال حمدی – ۹۷۳۱۱۱۱

بهار ۱۴۰۰

## ۱. نحوه‌ی مدل‌سازی مسئله برای جستجو

برای مدل‌سازی وضعیت محیط در هر مرحله از جستجو، یک کلاس گره (Node) در نظر گرفته شده است. در این کلاس، وضعیت خانه‌های جدول (اشیاء درونشان)، مختصات ربات، عمق جستجو، حرکت انجام شده در آن مرحله، شئ پدر این مرحله را نگه می‌داریم.

```python
class Node:
    def __init__(self, objects, robot_loc, depth, movement, parent, cost_g=None, cost_f=None):
        self.objects = objects
        self.robot_loc = robot_loc
        self.depth = depth
        self.movement = movement
        self.parent = parent
```

به کمک این کلاس، برای حل مسئله کافی‌ست درختواره‌ی جستجو را تشکیل دهیم، و آن را گسترش دهیم تا به هدف برسیم. بنا بر این از گره ریشه شروع کرده و فرزندان آن را تولید می‌کنیم. فرزندان یک گره، گره‌های حاصل از حرکت ربات گره مبدأ به جهات بالا، پایین، چپ و راست خواهند بود. اما باید توجه کنیم که حرکت به تمام این جهات لزوماً ممکن نیست؛ بلکه در شرایطی، حرکت به بعضی جهات غیر مجاز خواهند بود، مثال‌هایی از این موارد «حرکت به خارج از جدول»، «حرکت به مانع» و «حرکت در جهت هل دادن دو کره‌ی متوالی» هستند.
بنا بر این به کمک توابع زیر، حرکات مجاز را به دست آورده ، و به ازای هر یک از آن حرکات، گره فرزند تولید می‌کنیم.

```python
def get_valid_movements(objects, robot_loc):
    movements = ["u", "r", "d", "l"]
    valid_movements = []
    for movement in movements:
        if is_move_valid(objects, robot_loc, movement):
            valid_movements.append(movement)

    return valid_movements
```

```python
def is_move_valid(objects, robot_loc, movement):
    if not one_layer_check(objects, robot_loc, movement):
        return False

    robot_new_loc = calc_new_loc(robot_loc, movement)
    robot_new_x, robot_new_y = robot_new_loc

    if objects[robot_new_x][robot_new_y] == "b":
        butter_loc = robot_new_x, robot_new_y

        if not one_layer_check(objects, butter_loc, movement):
            return False

        butter_new_loc = calc_new_loc(butter_loc, movement)
        butter_new_x, butter_new_y = butter_new_loc
        if objects[butter_new_x][butter_new_y] == "b":
            return False

    return True
```

به کمک توابع بالا، فرزندان گره را تولید می‌کنیم.

```python
def generate_children(node):
    children = []

    objects, robot_loc, depth = node.objects, node.robot_loc, node.depth
    valid_movements = get_valid_movements(objects, robot_loc)
    for movement in valid_movements:
        new_objects, new_robot_loc = perform_move(objects, robot_loc, movement)
        new_depth = depth + 1

        child_node = Node(new_objects, new_robot_loc, new_depth, movement, node)

        children = [child_node] + children

    return children
```

در نهایت نیاز داریم به معیاری برای چک کردن آن که هر یک از گره‌های تولید شده شرط هدف را برآورده می‌کنند یا خیر. این معیار را به صورت تابع زیر پیاده می‌کنیم.
این تابع بررسی می‌کند که آیا خانه‌ای شامل کره‌ی تنها در جدول وجود دارد یا خیر. در صورت نبود چنین خانه‌ای، خواسته‌ی مسئله برآورده شده و به هدف رسیدیم و در غیر این صورت باید به جستجو ادامه دعیم.

```python
def is_in_goal(objects):
    for row in objects:
        if any(cell == "b" for cell in row):
            return False

    return True
```

حال که مسئله را به یک مدل انتزاعی تبدیل کردیم، و امکان تولید گره‌های فرزند برای هر گره را هم داریم، و شرط پایان را هم مشخص کردیم، می‌توانیم بنا بر الگوریتم انتخابی، درختواره‌ی جستجو را تشکیل داده و تا به گره هدف برسیم.

در این جا به عنوان مثال، پیاده‌سازی الگوریتم IDS را (به علت سادگی بیش‌تر) بررسی می‌کنیم. الگوریتم IDS تابع DLS (Depth Limited Search) را با عمق بیشینه‌های افزایشی و پله‌پله، صدا زده می‌زند. این الگوریتم این کار را تا زمانی انجام می‌دهد که تابع DLS جواب مسئله را پیدا کند، و یا عمق بیشینه، از مقدار مرزی تعیین شده بیش‌تر شود.

```python
def ids(starting_node, max_depth):
    for depth in range(max_depth):
        nodes = _dls(starting_node, depth)
        if len(nodes):
            return nodes

    return []
```

تابع DLS، الگوریتم DFS را با یک حداکثر عمق مشخص اجرا می‌کند، و در صورت یافتن مسیر به جواب آن را برمی‌گرداند. در این جا برای رعایت اختصار، از توضیح الگوریتم DFS خودداری می‌کنیم.

```python
def _dls(cur_node, limit):
    if is_in_goal(cur_node.objects):
        return [cur_node]

    if limit <= 0:
        return []

    children = generate_children(cur_node)
    for child in children:
        nodes = _dls(child, limit - 1)
        if len(nodes) > 0:
            return nodes + [child]

    return []
```

در این قسمت، مدل‌سازی مسئله را توضیح دادیم، روند کلی جستجو (ایجاد گره اولیه، ایجاد فرزندان، تشکیل درختواره و چک کردن شرط هدف) را بررسی کردیم و در نهایت روش الگوریتم IDS را شرح دادیم.

## ۲. تابع شهودی انتخاب شده و بررسی قابل قبول بودن آن

تابع شهودی پیشنهادی، به صورت زیر پیاده‌سازی شده است.

```python
def heuristic_1(point, plates_locs):
    h = 0

    plates_locs_cpy = deepcopy(plates_locs)
    while plates_locs_cpy:
        closest_plate, closest_plate_distance = get_closest_plate(point, plates_locs_cpy)

        plates_locs_cpy.remove(closest_plate)
        h += closest_plate_distance

        point = closest_plate

    return h
```

این تابع برای هر خانه «با شروع از آن خانه، مجموع مسافت منهتنی طی شده، برای پیمایش تمام خانه‌های هدف جدول» را محاسبه می‌کند. به عبارت دیگر، از خانه‌ی مبدأ شروع کرده، و در هر مرحله به نزدیک‌ترین خانه‌ی هدف (بشقاب) بعدی می‌رود، و مجموع فواصل منهتنی طی شده را گزارش می‌کند.
برای قابل قبول بودن، تابع شهودی هیچ‌گاه نباید مقدار هزینه‌ی مورد نیاز برای حل مسئله را بیش از مقدار واقعی تخمین بزند.
در این جا می‌دانیم که هزینه‌ی تمام خانه‌ها مساوی یا بیش‌تر از ۱ هستند، بنا بر این در بهترین حالت، و در صورتی که تمام خانه‌های مسیر هم هزینه‌ی یک داشته باشند، باز هم هزینه‌ی مورد نیاز بزرگ‌تر یا مساوی مقدار تابع شهودی خواهد بود.
چرا که ربات باید در هر مرحله، ابتدا به کنار کره رفته، و سپس کره را به خانه‌های هدف منتقل کند.

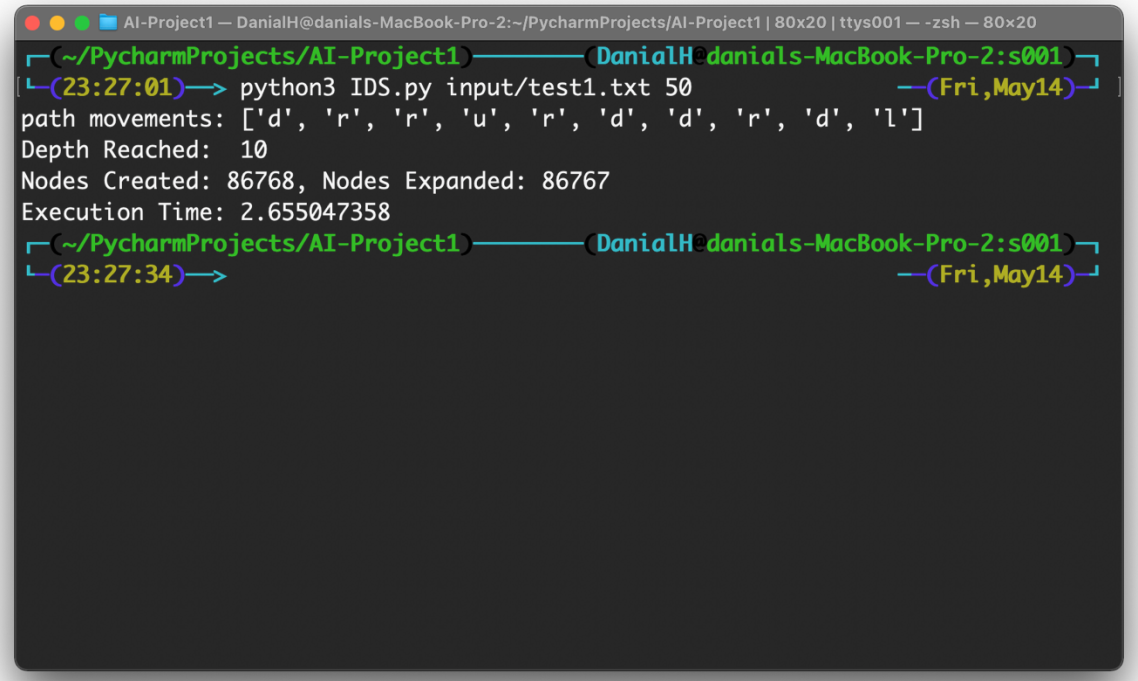
## ۳. توضیح کلی توابع و کلاس‌های تعریف شده در کد

تمامی کدها به طور کامل مستندسازی شده‌اند. همچنین در قسمت ۱، توابع اصلی مشترک، و توابع الگوریتم IDS را بررسی کردیم.

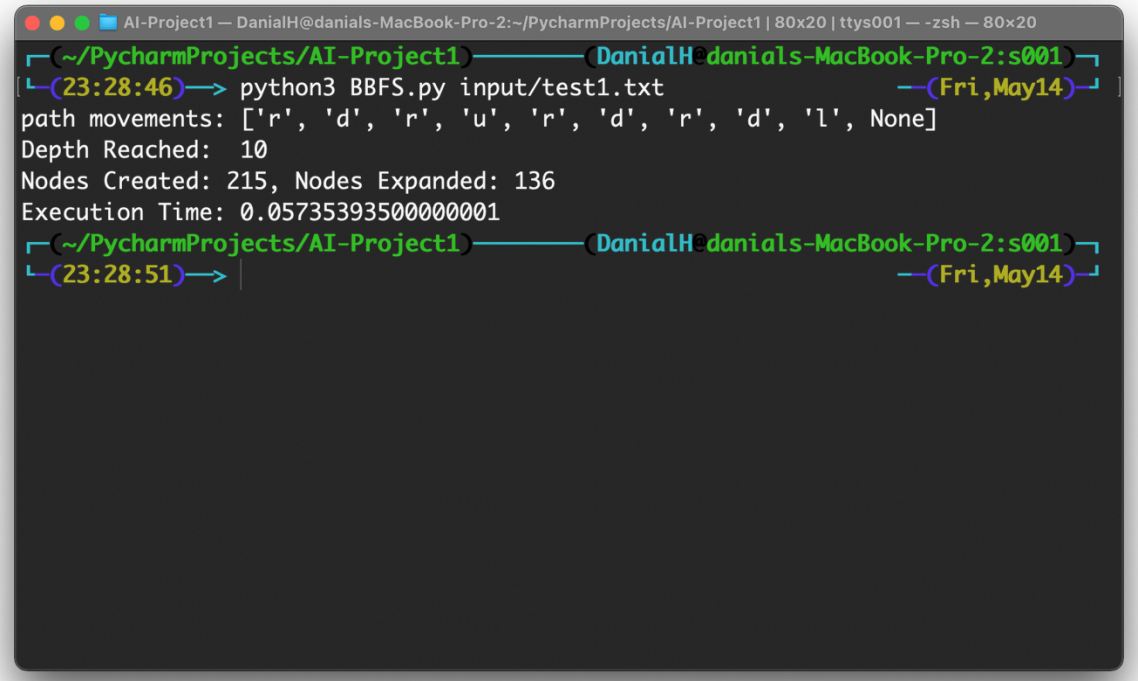
## ٤. مقایسه‌ی روش‌های پیاده‌سازی شده

در ادامه خروجی هر سه الگوریتم را برای تست‌کیس اول داده شده مقایسه می‌کنیم.

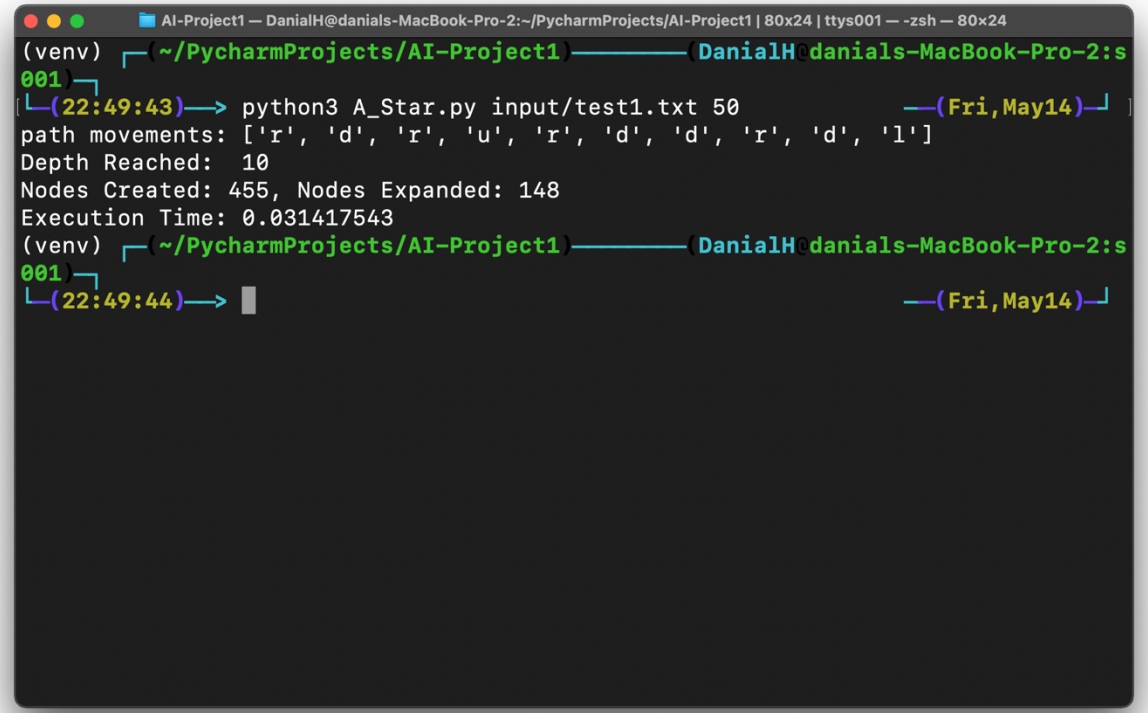
برای الگوریتم IDS داریم:




و برای الگوریتم BBFS:



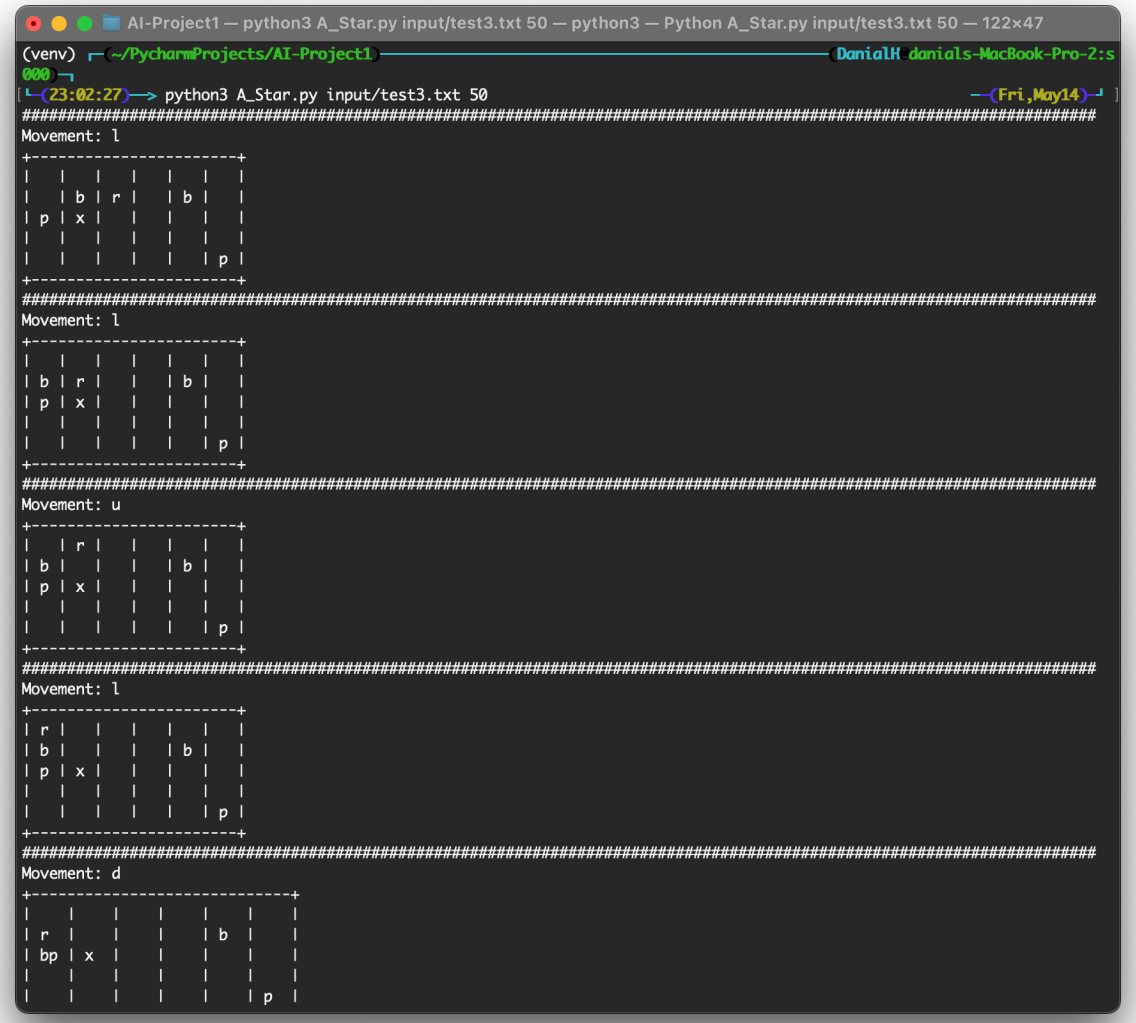

و در نهایت الگوریتم A*:

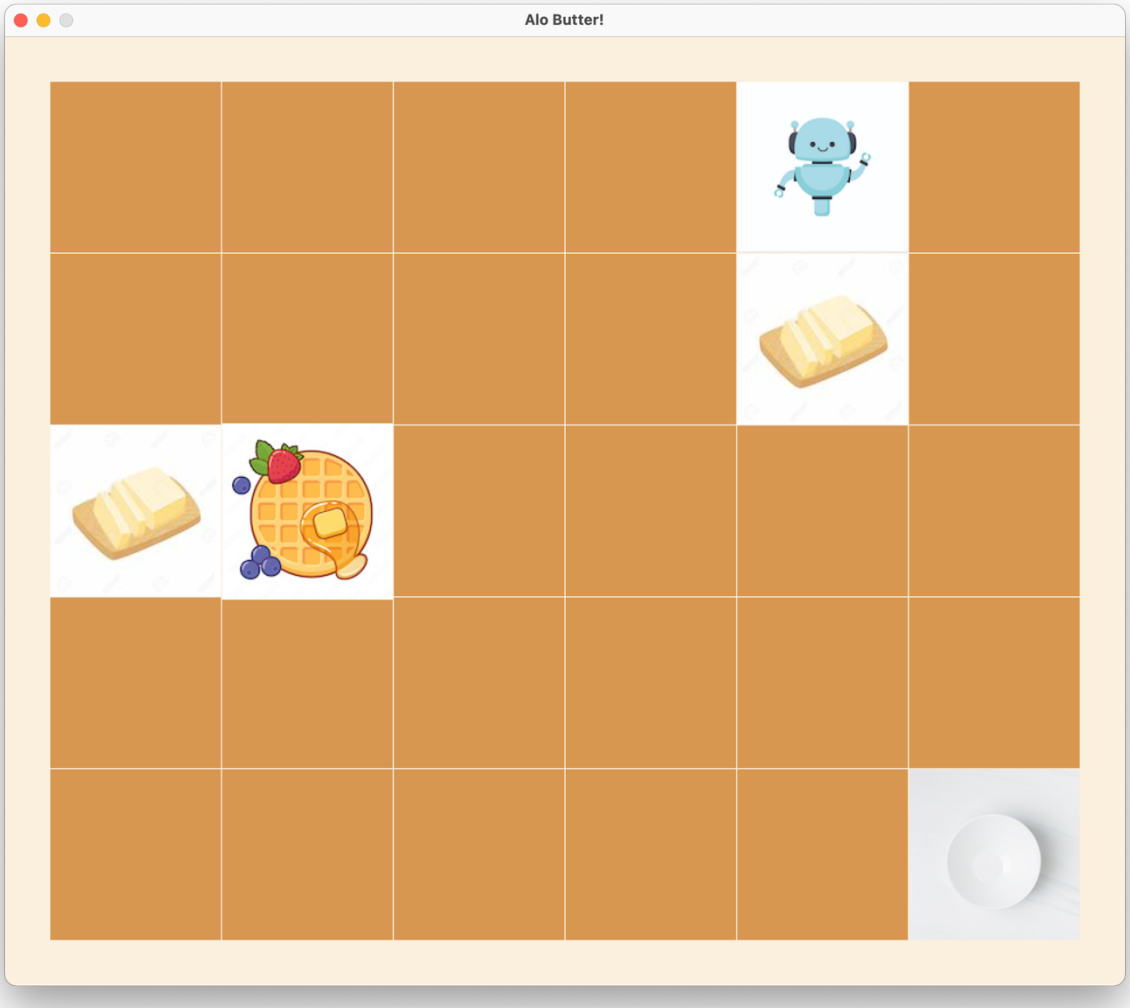


مطابق انتظار، زمان اجرا و تعداد گره‌های تولید شده در الگوریتم IDS بسیار بیش‌تر از الگوریتم‌های BBFS و A* است.
برای پیچیدگی زمانی این الگوریتم‌ها داریم:

$$IDS: O(b^d)$$
$$BBFS: O(b^{\frac{d}{2}})$$
$$A*: depends\ on\ the\ heuristic$$

## ۵. نمونه‌ی خروجی‌های واسط خط فرمان و واسط گرافیکی کاربری




Alo Butter!