گزارش پروژه

عرفان رفیعی اسکویی - 98243027

مقدمه

برای پیاده‌سازی الگوریتم‌های خواسته شده، ابتدا باید فرضیات انجام شده و توابع پایه را توضیح داده و سپس با استفاده از آن‌ها و داده‌ساختارهای مناسب الگوریتم‌های جستجو را پیاده‌کنیم. به این منظور با توجه به تعریف مسئله، وضعیت هزارتو همواره ثابت بوده و تنها موقعیت عامل تغییر می‌کند. در نتیجه هر حالت جستجو تنها با داشتن موقعیت عامل مشخص می‌شود. در نتیجه تابع زیر را پیاده‌سازی می‌کنیم که با گرفتن یک حالت (موقعیت عامل) حالت‌های همسایه را که در واقع خانه‌های قابل دسترسی توسط عامل به فاصله یک هستند را بر می‌گرداند.

```
def get_neighbours(self, state):
    states = []
    for direction in [(-1, 0), (+1, 0), (0, -1), (0, +1)]:
        x, y = state[0] + direction[0], state[1] + direction[1]
        if x < 0 or x >= len(self.current_state):
            continue
        if y < 0 or y >= len(self.current_state[x]):
            continue
        tile = self.current_state[x][y]
        if not tile.is_blocked():
            states.append((x, y))
    return states
```

سپس برای مشخص کردن حالت‌های جستجو شده توسط هر الگوریتم تابع زیر پیاده‌سازی شده است که خانه متناسب را رنگ می‌زند.

```
def expand(self, pos):
    tile = self.current_state[pos[0]][pos[1]]
    if tile.isGoal or tile.isStart:
        return
    tile.set_color(colors.selectionColor)
```

الگوریتم‌های جستجو باید با دریافت board یک لیست شامل مسیر رسیدن به هدف برگردانند که با توجه به این لیست، مسیر روی هزارتو رنگ زده می‌شود. صدا زدن هر الگوریتم باید در constructor کلاس به شکل زیر اتفاق بیافتد.

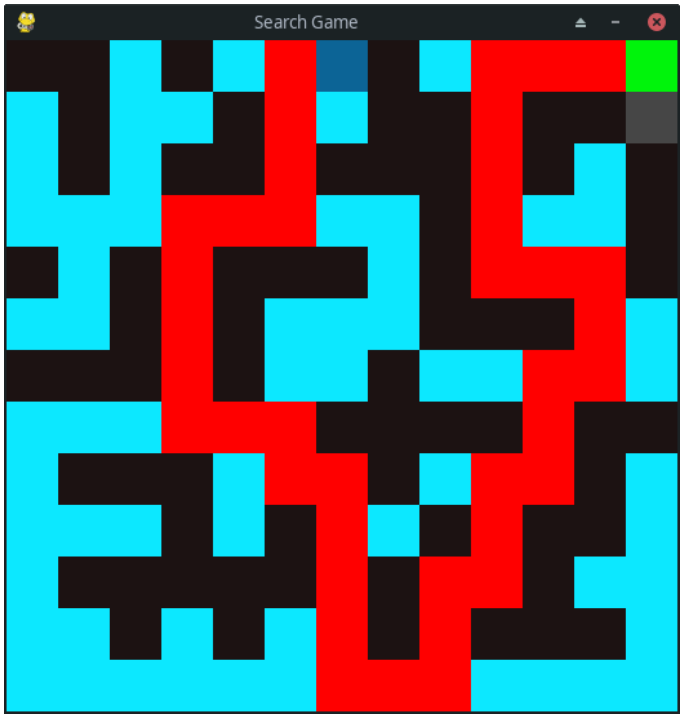
```
def __init__(self, board):
    self.position = board.get_agent_pos()
    self.current_state = board.get_current_state()
    for action in self.bfs(board):
        board.colorize(action[0], action[1])
```

الگوریتم bfs

در این الگوریتم با استفاده از داده ساختار صف، جستجو را پیاده‌سازی می‌کنیم. به این صورت که در ابتدا گره‌ی آغازین را به صف اضافه کرده و سپس تا زمانی که صف خالی نشده، گره بعدی را انتخاب کرده و همسایه‌های آن را به صف اضافه می‌کنیم. در هر بار انتخاب گره هدف بودن آن نیز مورد بررسی قرار می‌گیرد. همچنین مسیر رسیدن به هر گره نیز به همراه آن در صف قرار داده می‌شود تا مسیر نهایی به دست آید.

```
def bfs(self, environment):
        self.percept(environment)
        queue = []
        visited = set()
        agent_pos = self.get_position()
        queue.append((agent_pos, []))
        visited.add(agent_pos)
        goal_path = None
        while len(queue) > 0:
            state, path = queue.pop(0)
            self.expand(state)
            if self.current_state[state[0]][state[1]].isGoal:
                goal_path = path
                break
            for next_state in self.get_neighbours(state):
                if not next_state in visited:
                    queue.append((next_state, path + [state]))
                    visited.add(next_state)
        return goal_path[1:]
```

نتیجه‌ی اجرا:



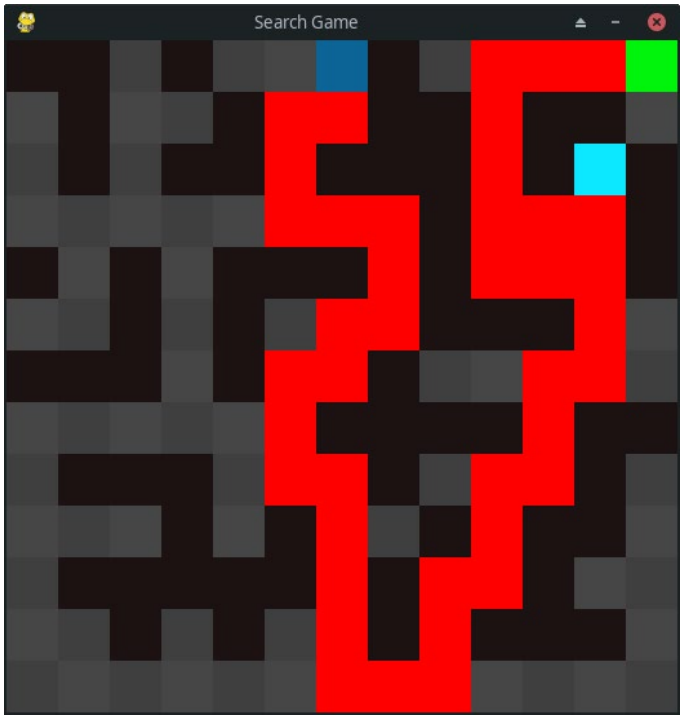
خانه‌های قرمز نمایانگر مسیر و خانه‌های آبی نمایانگر گره‌های پیمایش شده هستند.
همانطور که مشخص است در این الگوریتم تعداد پیمایش‌ها زیاد بوده ولی مسیر بهینه پیدا می‌شود.
پیچیدگی زمانی این الگوریتم برابر با $O(V+E)$ بوده و با توجه به اینکه در این سؤال $O(E) = O(V)$ است پیچیدگی آن برابر با $O(V)$ است.

الگوریتم dfs

در این الگوریتم کافی است داده ساختار مورد استفاده را به پشته تغییر دهیم. در نتیجه به جای جستجوی سطح اول، جستجوی عمق اول خواهیم داشت.

```
def dfs(self, environment):
        self.percept(environment)
        stack = []
        visited = set()
        agent_pos = self.get_position()
        stack.insert(0, (agent_pos, []))
        goal_path = None
        while len(stack) > 0:
            state, path = stack.pop(0)
            visited.add(state)
            self.expand(state)
            if self.current_state[state[0]][state[1]].isGoal:
                goal_path = path
                break
            for next_state in self.get_neighbours(state):
                if not next_state in visited:
                    stack.insert(0, (next_state, path + [state]))
        return goal_path[1:]
```

نتیجه‌ی اجرا:



همانطور که مشخص است در این الگوریتم تعداد پیمایش‌ها به مراتب کمتر بوده ولی مسیر بهینه لزوماً پیدا نمی‌شود.

پیچیدگی زمانی این الگوریتم برابر با $O(V+E)$ بوده و با توجه به اینکه در این سؤال $O(E) = O(V)$ است پیچیدگی آن برابر با $O(V)$ است.

الگوریتم a_star

ابتدا تابع هیوریستک که فاصله‌ی منهتن تا هدف را حساب می‌کند، تعریف می‌کنیم.

```
def get_goal(self):
        for i in range(len(self.current_state)):
            for j in range(len(self.current_state[i])):
                if self.current_state[i][j].isGoal:
                    return i, j

def heuristic(self, state):
        goal_pos = self.get_goal()
        return abs(state[0] - goal_pos[0]) + abs(state[1] - goal_pos[1])
```

سپس داده ساختار صف اولویت را با استفاده از داده ساختار heap که در پایتون به صورت کتابخانه قابل استفاده است به شکل زیر پیاده‌سازی می‌کنیم.

```
class PriorityQueue:
    def __init__(self):
        self.heap = []

    def push(self, item, priority):
        entry = (priority, item)
        heapq.heappush(self.heap, entry)

    def pop(self):
        (_, item) = heapq.heappop(self.heap)
        return item

    def isEmpty(self):
        return len(self.heap) == 0

    def update(self, item, priority):
        for index, (p, i) in enumerate(self.heap):
            if i == item:
                if p <= priority:
                    break
                del self.heap[index]
                self.heap.append((priority, item))
                heapq.heapify(self.heap)
                break
        else:
            self.push(item, priority)
```

سپس با استفاده از این داده ساختار و تابع تخمین خود، الگوریتم را به شکل زیر پیاده می‌کنیم.

```
def a_star(self, environment):
        self.percept(environment)
        pqueue = Agent.PriorityQueue()
        visited = set()
        agent_pos = self.get_position()
        pqueue.push((agent_pos, [], 0), 0)
        goal_path = []
        while not pqueue.isEmpty():
            (state, path, g) = pqueue.pop()
            if state in visited:
                continue
            self.expand(state)
            visited.add(state)
            if self.current_state[state[0]][state[1]].isGoal:
                goal_path = path
                break
            for next_state in self.get_neighbours(state):
                if not next_state in visited:
                    pqueue.update(
                        (next_state, path + [state], g + 1), g + 1 + self.heuristic(next_state))
        return goal_path[1:]
```

نتیجه‌ی اجرا:

مشهود است که در این الگوریتم جواب بهینه با پیمایش گره‌های کمتری نسبت به bfs پیدا شده است.