"به نام یزدان پاک"

گزارش کار پروژه اول هوش

نام استاد:

استاد روشن فکر

اعضای گروه:

کیانا آقا کثیری 9831006

سارا تاجرنیا 9831016

تاریخ آزمایش : 1400/2/23

1) IDS Search (عمق اول با افزایش تدریجی عمق)

در این الگوریتم با توجه به ضریب انشعاب (k) از الگوریتم DFS استفاده کنیم به طوری که ابتدا k را برابر 0 میگذاریم و به تدریج آن را زیاد میکنیم(... k=0, 1,) و هر دفعه تا عمق k , DFS را صدا میزنیم تا هدف را پیدا کنیم. خوبی این الگوریتم این است که هم حافظه کمی مثل DFS را اشغال میکند هم Optimal است.

مدل سازی← ابتدا cut off را برابر 0 میگذارد و هر دفعه اگر هدف را پیدا نکرد آن را ++ میکند تا دفعه بعد با عمق بیشتری سرچ کند. و الگوریتم DFS الگوریتمی بازگشتی است که هر دفعه فرزندان خود را گسترش میدهد تا به cut off برسد و پس از آن همه ی مسیر ها با این طول را چک میکند و cut off را زیاد میکند.

کد این الگوریتم شامل 3 کلاس node, environment, agent است که هر یک به شرح زیر هستند:

```
class node():
    def __init__(self, parent, robot, butter, action_from_par):
        self.parent = parent
        self.robot = robot
        self.butter = butter
        self.action_from_par = action_from_par
```

کلاس node برای نگهداری مختصات رباط و کره موجود در نقشه و parent هر node و action که با استفاده از آن به Node رسیده مثلا اگر از parent به سمت پایین اومده تا به node برسه 'D' را ذخیره میکند.

```
class for map and finding robot and butters and plates and
obstacles position
and check the available actions for both normal and reversed
movements
'''
class environment():
    def __init__(self, map_file):
        self.rows, self.cols = map(int,
map_file.readline().split())
        self.table = [map_file.readline().split() for j in
range(self.rows)]
        self.find_people()
        self.find_butter()
```

به طور کلی کلاس environment برای ذخیره کلی نقشه است به طوری که با وجود table و تعداد سطر ها و اطلاعات کره و people به عنوان مقصد مسیرهای ممکن را پیدا میکند سپس action ها را چک میکند تا قابل دسترسی باشند و هر دفعه نقشه را آپدیت میکند.

```
'''
to have the plates and butters position
'''
def map(self):
    return self.people_list, self.butter_list
```

تابع map در کلاس environment برای نگهداری اطلاعات نقشه به صورت list است به طوری که در توابع دیگر به راحتی بتوان به اطلاعات people و butter دسترسی داشت.

```
'''
to find plates(people) position
'''def find_people(self):
    self.people_list = []
    for i in range(self.rows):
        for j in range(self.cols):
            if 'p' in self.table[i][j]:
                self.people_list.append((i, j))
```

تابع find people برای پیدا کردن p (های) موجود در نقشه به عنوان مقصد کره

```
'''
to find robot position
'''
def find_robot(self):
    for i in range(self.rows):
        for j in range(self.cols):
            if 'r' in self.table[i][j]:
                return i, j
```

تابع find robot برای پیدا کردن r موجود در نقشه به عنوان نقطه شروع

```
'''
to find the butters position
'''
def find_butter(self):
    self.butter_list = []
    for i in range(self.rows):
        for j in range(self.cols):
            if 'b' in self.table[i][j]:
                self.butter_list.append((i, j))
```

تابع find butter برای پیدا کردن b (های) موجود در نقشه

```
def find_obstacles(self):
    self.obstacle_list = []
    for i in range(self.rows):
        for j in range(self.cols):
            if 'x' in self.table[i][j]:
                self.obstacle_list.append((i, j))
```

تابع find obstacles برای پیدا کردن x (های) موجود در نقشه

```
check all the available actions for robot and return them
'''
def available_actions(self, robot, butter):
    actions = []
    if robot[0] - 1 >= 0 and self.table[robot[0] - 1][robot[1]]
!= "x":
        if butter == (robot[0] - 1, robot[1]):
            if robot[0] - 2 >= 0 and self.table[robot[0] -
2][robot[1]] != "x":
                actions.append("U")
        else:
            actions.append("U")

    if robot[0] + 1 < self.rows and self.table[robot[0] +
1][robot[1]] != "x":
        if butter == (robot[0] + 1, robot[1]):
            if robot[0] + 2 < self.rows and self.table[robot[0]
+ 2][robot[1]] != "x":
                actions.append("D")
        else:
            actions.append("D")

    if robot[1] - 1 >= 0 and self.table[robot[0]][robot[1] - 1]
!= "x":
        if butter == (robot[0], robot[1] - 1):
            if robot[1] - 2 >= 0 and
self.table[robot[0]][robot[1] - 2] != "x":
                actions.append("L")
        else:
            actions.append("L")
```

```
    if robot[1] + 1 < self.cols and
self.table[robot[0]][robot[1] + 1] != "x":
        if butter == (robot[0], robot[1] + 1):
            if robot[1] + 2 < self.cols and
self.table[robot[0]][robot[1] + 2] != "x":
                actions.append("R")
        else:
            actions.append("R")
    return actions
```

تابع available action با وجود داشتن مختصات رباط و کره چک میکند که اول رباط در action که میخواهد انجام دهد از نقشه خارج نشود و همچنین action که انتخاب میکند به x ختم نشود در این صورت چک میکند که کره ای در اطراف رباط بود چک کند در صورتی بتواند به آن سمت برود که حتما وقتی کره را هول میدهد راه آن باز باشد و x در آنجا نباشد.

```
'''
check the action given with all available actions
if allowed then it moves the robot and if needed the butter
'''
def step(self, robot, butter, action):
    # action : "U", "D", "R", "L"
    # robot: tuple (x, y)
    if action not in self.available_actions(robot, butter):
        raise ValueError("action is not available")

    if action == "U":
        next_robot = robot[0] - 1, robot[1]
        cost = self.table[next_robot[0]][next_robot[1]]
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        if next_robot == butter:
            butter = butter[0] - 1, butter[1]
        return next_robot, butter, int(cost)

    elif action == "D":
        next_robot = robot[0] + 1, robot[1]
```

```
        cost = self.table[next_robot[0]][next_robot[1]]
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        if next_robot == butter:
            butter = butter[0] + 1, butter[1]
        return next_robot, butter, int(cost)

    elif action == "L":
        next_robot = robot[0], robot[1] - 1
        cost = self.table[next_robot[0]][next_robot[1]]
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        if next_robot == butter:
            butter = butter[0], butter[1] - 1
        return next_robot, butter, int(cost)

    elif action == "R":
        next_robot = robot[0], robot[1] + 1
        cost = self.table[next_robot[0]][next_robot[1]]
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        if next_robot == butter:
            butter = butter[0], butter[1] + 1
        return next_robot, butter, int(cost)

    else:
        raise ValueError("action is wrong")
```

تابع step با وجود ورودی رباط و کره و action که میخواهیم روی آن پیاده کنیم و وجود تابع available action چک میکند که همچین حرکتی با این مختصات برای رباط ممکن است یا خیر و اگر ممکن بود هزینه وارد شده برای این حرکت را بدست میاورد.

```
class agent():
    def __init__(self, env):
        self.env = env
        self.butter = env.find_butter()
        self.robot = env.find_robot()
        self.people = env.find_people()
```

کلاس agent که در این کلاس از کلاس و توابع دیگر استفاده میکنیم که شامل تابع ids و dfs است این کلاس شامل environment یعنی کل اطلاعات نقشه و همچنین اطلاعات node های کره و رباط و آدم که مقصد است.

```
def ids(self,people_list, butter, robot):
    root = node(None, robot, butter, None)
    queue_temp=[]
    depth_limit = 20
    i = 0
    while (i<=depth_limit):
        front=[]
        res=self.dfs(root,front,i, people_list)
        if(res is not None):
            if   res[0] is not None and res[0] is not False:
                res[0].insert(0, root)
                return res[0],res[1],res[2],i
        i+=1
```

کار تابع ids این است که هر در حلقه cut off را تعیین میکند که تا depth limit میتواند پیش برود و هر دفعه برای آن تابع dfs را صدا میزند.

```
recursive dfs algorithm
'''
def dfs(self,Node:node,path,depth, people_list):
    for person in people_list:
        if Node.butter == person:
            path, action_path, robot_last = self.show_path(Node)
            return path, action_path, robot_last
    if(depth<=0):
        return None,None,None
```

```
    next=[]
    actions = env.available_actions(Node.robot, Node.butter)
    for a in actions:
        next_robot, next_butter, _ = env.step(Node.robot,
Node.butter, a)
        child = node(Node, next_robot, next_butter, a)
        next.append(child)
    for childs in next:
        res = self.dfs(childs,path,depth-1, people_list)
        if(res is not None):
            if res[0] is not None:
                path.insert(0, childs)
                return path,res[1],res[2]
```

تابع dfs به این صورت کار میکند که هر گره تا رسیدن به برگ تعیین شده (تعداد شاخه ها مشخص است) را گسترش میدهد و تابعی بازگشتی است تا بتواند به صورت عمقی پیش برود.

```
def show_path(self, final_node):
    path = []
    action_path = []
    n = final_node
    print("****")
    while n.parent != None:
        path.append(n.robot)
        action_path.append(n.action_from_par)
        n = n.parent
    path.append(n.robot)
    path.reverse()
    action_path.reverse()
    print("path:", path)
    print("actions:", action_path)
    print("cost:", len(action_path))
    return action_path, path, final_node.robot
```

برای تابع show path با استفاده از node نهایی و این که parent آن را ذخیره کرده ایم به صورت عقب گرد parent. را صدا میکنیم تا بتوانیم path مورد نظر را پیدا کنیم.

```
if __name__ == "__main__":
    with open("test3.txt", "r") as file:
        env = environment(file)
    people_list, butter_list = env.map()
    robot = env.find_robot()
    test_agent = agent(env)
    for i in range(len(butter_list)):
        res = test_agent.ids(people_list, butter_list[i], robot)
        robot = res[2]
        if res[3] == 0:
            print("Impossible!")
            print("path:", [])
            print("actions:", [])
            print("cost:", res[3])
        print("goal depth:", res[3])
```

در تابع main هم با کمک توابع و کلاس ها به ازای همه ی کره ها ids را صدا میکنیم و بهینه ترین راه را تا بشقاب بدست میاوریم.

2) Bidirectional BFS (دوطرفه سطح اول گرافی)

برای حل این الگوریتم به روش BFS ابتدا ۲ نقطه را به عنوان شروع و مقصد در نظر میگیریم سپس هر نقطه را گسترش میدهیم از دو طرف و هر یک از آنها را چک میکنیم که بهم رسیده اند یا خیر اگر نبود دوباره فرزندان را گسترش میدهیم تا در نهایت در یک نقطه با هم به اشتراک برسند.

مدل سازی ← BFS که خود سطر به سطر node های گسترش یافته را چک میکند که با استفاده از یک صف و ذخیره شدن node های visited است اما برای ۲ طرفه بود آن یک بار ما از مسیر به سمت هدف BFS میزنیم و یک بار هم فرض میکنیم که کره روی بشقاب قرار دارد و حال به صورت reverse تلاش میکنیم تا به نقطه شروع برسیم برای این کار هم به یک صف و visited نیاز داریم تا بتوانیم بفهمیم این دو کجا به یکدیگر میرسند.

کد این الگوریتم شامل 3 کلاس node, environment, agent است که هر یک به شرح زیر هستند:

```
'''
class for nodes
we save the butter and robot for each node
and also parent and th action from parent for path
'''
class node():
    def __init__(self, parent, robot, butter, action_from_par):
        self.parent = parent
        self.robot = robot
        self.butter = butter
        self.action_from_par = action_from_par
```

کلاس node برای نگهداری مختصات رباط و کره موجود در نقشه و parent هر node و action که با استفاده از آن به Node رسیده مثلا اگر از parent به سمت پایین اومده تا به node برسه ’D‘ را ذخیره میکند.

```
class environment():
    def __init__(self, map_file):
        self.rows, self.cols = map(int,
map_file.readline().split())
        self.table = [map_file.readline().split() for j in
range(self.rows)]
        self.find_people()
        self.find_butter()
```

به طور کلی کلاس environment برای ذخیره کلی نقشه است به طوری که با وجود table و تعداد سطر ها و اطلاعات کره و people به عنوان مقصد مسیرهای ممکن را پیدا میکند سپس action ها را چک میکند تا قابل دسترسی باشند و هر دفعه نقشه را آپدیت میکند.

```
to have the plates and butters position
'''
def map(self):
    return self.people_list, self.butter_list
```

تابع map در کلاس environment برای نگهداری اطلاعات نقشه به صورت list است به طوری که در توابع دیگر به راحتی بتوان به اطلاعات people و butter دسترسی داشت.

```
'''
to find plates(people) position
'''
def find_people(self):
    self.people_list = []
    for i in range(self.rows):
        for j in range(self.cols):
            if 'p' in self.table[i][j]:
                self.people_list.append((i, j))
```

تابع find people برای پیدا کردن p (های) موجود در نقشه به عنوان مقصد کره

```
'''
to find robot position
'''
def find_robot(self):
    for i in range(self.rows):
        for j in range(self.cols):
            if 'r' in self.table[i][j]:
                return i, j
```

تابع find robot برای پیدا کردن r موجود در نقشه به عنوان نقطه شروع

```
'''
to find the butters position
'''
def find_butter(self):
    self.butter_list = []
    for i in range(self.rows):
        for j in range(self.cols):
            if 'b' in self.table[i][j]:
                self.butter_list.append((i, j))
```

تابع find butter برای پیدا کردن b (های) موجود در نقشه

```
def find_obstacles(self):
    self.obstacle_list = []
    for i in range(self.rows):
        for j in range(self.cols):
            if 'x' in self.table[i][j]:
                self.obstacle_list.append((i, j))
```

تابع find obstacles برای پیدا کردن x (های) موجود در نقشه

```
'''
check all the available actions for robot and return them
'''
def available_actions(self, robot, butter):
    actions = []
    if robot[0] - 1 >= 0 and self.table[robot[0] - 1][robot[1]]
!= "x":
        if butter == (robot[0] - 1, robot[1]):
            if robot[0] - 2 >= 0 and self.table[robot[0] -
2][robot[1]] != "x":
                actions.append("U")
        else:
            actions.append("U")

    if robot[0] + 1 < self.rows and self.table[robot[0] +
1][robot[1]] != "x":
        if butter == (robot[0] + 1, robot[1]):
            if robot[0] + 2 < self.rows and self.table[robot[0]
+ 2][robot[1]] != "x":
                actions.append("D")
        else:
            actions.append("D")

    if robot[1] - 1 >= 0 and self.table[robot[0]][robot[1] - 1]
!= "x":
        if butter == (robot[0], robot[1] - 1):
            if robot[1] - 2 >= 0 and
self.table[robot[0]][robot[1] - 2] != "x":
                actions.append("L")
        else:
```

```
            actions.append("L")

    if robot[1] + 1 < self.cols and
self.table[robot[0]][robot[1] + 1] != "x":
        if butter == (robot[0], robot[1] + 1):
            if robot[1] + 2 < self.cols and
self.table[robot[0]][robot[1] + 2] != "x":
                actions.append("R")
        else:
            actions.append("R")
    return actions
```

تابع available action با وجود داشتن مختصات رباط و کره چک میکند که اول رباط در action که میخواهد انجام دهد از نقشه خارج نشود و همچنین action که انتخاب میکند به x ختم نشود در این صورت چک میکند که کره ای در اطراف رباط بود چک کند در صورتی بتواند به آن سمت برود که حتما وقتی کره را هول میدهد راه آن باز باشد و x در آنجا نباشد.

```
'''
check the action given with all available actions
if allowed then it moves the robot and if needed the butter
'''
def step(self, robot, butter, action):
    # action : "U", "D", "R", "L"
    # robot: tuple (x, y)
    if action not in self.available_actions(robot, butter):
        raise ValueError("action is not available")

    if action == "U":
        next_robot = robot[0] - 1, robot[1]
        cost = self.table[next_robot[0]][next_robot[1]]
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        if next_robot == butter:
            butter = butter[0] - 1, butter[1]
        return next_robot, butter, int(cost)
```

```
    elif action == "D":
        next_robot = robot[0] + 1, robot[1]
        cost = self.table[next_robot[0]][next_robot[1]]
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        if next_robot == butter:
            butter = butter[0] + 1, butter[1]
        return next_robot, butter, int(cost)

    elif action == "L":
        next_robot = robot[0], robot[1] - 1
        cost = self.table[next_robot[0]][next_robot[1]]
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        if next_robot == butter:
            butter = butter[0], butter[1] - 1
        return next_robot, butter, int(cost)

    elif action == "R":
        next_robot = robot[0], robot[1] + 1
        cost = self.table[next_robot[0]][next_robot[1]]
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        if next_robot == butter:
            butter = butter[0], butter[1] + 1
        return next_robot, butter, int(cost)

    else:
        raise ValueError("action is wrong")
```

تابع step با وجود ورودی رباط و کره و action که میخواهیم روی آن پیاده کنیم و وجود تابع available action چک میکند که همچین حرکتی با این مختصات برای رباط ممکن است یا خیر و اگر ممکن بود هزینه وارد شده برای این حرکت را بدست میاورد.

```
'''
check all the available actions for robot in reversed state and
return them
'''
def available_actions_reverse(self, robot, butter):
    actions = []
    if robot[0] - 1 >= 0 and self.table[robot[0] - 1][robot[1]]
!= "x":
        if butter != (robot[0] - 1, robot[1]):
            actions.append("U")

    if robot[0] + 1 < self.rows and self.table[robot[0] +
1][robot[1]] != "x":
        if butter != (robot[0] + 1, robot[1]):
            actions.append("D")

    if robot[1] - 1 >= 0 and self.table[robot[0]][robot[1] - 1]
!= "x":
        if butter != (robot[0], robot[1] - 1):
            actions.append("L")

    if robot[1] + 1 < self.cols and
self.table[robot[0]][robot[1] + 1] != "x":
        if butter != (robot[0], robot[1] + 1):
            actions.append("R")

    return actions
```

در کد های قبل طرفی را بررسی میکردیم که از طرف کره هر node گسترش پیدا میکرد حال طرفی را در نظر میگیریم که از طرف goal یعنی p مسیر میخواهد گسترش پیدا کند حال فرض میکنیم در ابتدا کره در مقصد یعنی مختصات p قرار دارم و رباط در یکی از خانه های اطراف آن است و همه چیز را به طور برعکس چک میکنیم گویا رباط بجای هول دادن کره آن را کشیده است.

حال علاوه بر اینکه چک میکنیم رباط از نقشه بیرون نرود یا به مانع برخورد نکند چک میکنیم سمتی که رباط میخواهد برود کره نباشد زیرا همچین چیزی در حالت reverse درعمل ممکن نیست انگار که کره رباط را هل داده است.

```
def step_inverse(self, robot, butter, action):
    if action not in self.available_actions_reverse(robot,
butter):
        raise ValueError("action is not available")

    if action == "U":
        next_robot = robot[0] - 1, robot[1]
        cost = self.table[next_robot[0]][next_robot[1]]
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        robot_D = robot[0] + 1, robot[1]
        if butter == robot_D:
            butter = robot
        return next_robot, butter, int(cost)

    elif action == "D":
        next_robot = robot[0] + 1, robot[1]
        cost = self.table[next_robot[0]][next_robot[1]]
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        robot_U = robot[0] - 1, robot[1]
        if butter == robot_U:
            butter = robot
        return next_robot, butter, int(cost)

    elif action == "L":
        next_robot = robot[0], robot[1] - 1
        cost = self.table[next_robot[0]][next_robot[1]]
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        robot_R = robot[0], robot[1] + 1
        if butter == robot_R:
            butter = robot
        return next_robot, butter, int(cost)

    elif action == "R":
        next_robot = robot[0], robot[1] + 1
        cost = self.table[next_robot[0]][next_robot[1]]
```

```
        cost = cost.replace("b", "")
        cost = cost.replace("r", "")
        cost = cost.replace("p", "")
        robot_L = robot[0], robot[1] - 1
        if butter == robot_L:
            butter = robot
        return next_robot, butter, int(cost)

    else:
        raise ValueError("action is wrong")
```

تابع step inverse هم مانند تابع step برای یافتن هزینه مسیر است با این تفاوت که در آن تابع چک میکردیم که کره در خانه بعدی رباط باشد اما الان چک میکنیم که کره بتواند به خانه ای که رباط در حال حاضر در آن قرار دارد برود.

```
'''
class for bidirectional method and returning the path and
actions
'''

class agent():
    def __init__(self, env):
        self.env = env
        self.butter = env.find_butter()
        self.robot = env.find_robot()
        self.people = env.find_people()
```

کلاس agent که در این کلاس از کلاس و توابع دیگر استفاده میکنیم که شامل search bidirectional BFS هم هست که ورودی های این کلاس شامل environment یعنی کل اطلاعات نقشه و همچنین اطلاعات node های کره و رباط و آدم که مقصد است.

```
'''
The Main Method!!!
for the normal walking from the start node its as same as bfs
for the reversed walking the goal is butter in plate and robot
next to them
so for reversed we start from all the states that robot is next
to the plates(people)
returns the robot position for the other movements
returns the path and actions
'''
def bidirectional_bfs(self, people_list, butter, robot):
    queue_r = []
    queue_p = []
    is_visited_r = {}
    is_visited_p = {}
    root_r = node(None, robot, butter, None)
    queue_r.append(root_r)

    for i in range(len(people_list)):
        actions = env.available_actions(people_list[i],
people_list[i])
        for a in actions:
            next_robot, next_butter, _ =
env.step(people_list[i], people_list[i], a)
            child = node(None, robot=next_robot,
butter=people_list[i], action_from_par=None)
            queue_p.append(child)

    while len(queue_p) > 0 and len(queue_r) > 0:
        node_r = queue_r.pop(0)
        node_p = queue_p.pop(0)
        is_visited_r[node_r.robot + node_r.butter] = node_r
        is_visited_p[node_p.robot + node_p.butter] = node_p
        ##the robots see each other:)) and the butters are in
the same position
        if node_r.robot + node_r.butter in is_visited_p.keys():
            action_path, path =
self.show_bidirectional_path(node_r, is_visited_p[node_r.robot +
node_r.butter])
            n = is_visited_p[node_r.robot + node_r.butter]
            while n.parent != None:
```

```
                n = n.parent
            return n.robot, action_path, path
        if node_p.robot + node_p.butter in is_visited_r.keys():
            action_path, path =
self.show_bidirectional_path(is_visited_r[node_p.robot +
node_p.butter], node_p)
            n = node_p
            while n.parent != None:
                n = n.parent
            return n.robot, action_path, path
        actions = env.available_actions(node_r.robot,
node_r.butter)
        for a in actions:
            next_robot, next_butter, _ = env.step(node_r.robot,
node_r.butter, a)
            if next_robot + next_butter not in
is_visited_r.keys():
                child = node(node_r, next_robot, next_butter, a)
                queue_r.append(child)

        actions =
env.available_actions_reverse(robot=node_p.robot,
butter=node_p.butter)
        for a in actions:
            next_robot, next_butter, _ =
env.step_inverse(robot=node_p.robot, butter=node_p.butter,
action=a)
            if next_robot + next_butter not in
is_visited_p.keys():
                child = node(node_p, next_robot, next_butter, a)
                queue_p.append(child)
    return [], [], []
```

در تابع bidirectional BFS ابتدا ۲ صف را برای r و p و node های visited شده را در نظر میگیریم .حالت ابتدایی برای صف r همان node داده شده r است اما برای صف p باید همه ی همسایه هایی ممکن که رباط در اطراف کره باشد را در نظر میگیریم که تعداد این حالات از ۰ تا ۴ میتواند برای هر بشقاب متغیر باشد.

از داخل هر یک از این صف ها به نوبت یکی از node های را حذف میکنیم تا هر دفعه به هم نزدیک تر شوند و برای add کردن به صف node هایی را اضافه میکنیم که available باشند و visited نباشند و سپس آنها را در is visited اضافه میکنیم. برای چک کردن این که این مسیر دو طرفه به هم رسیده اند یا خیر چک میکنیم که visitedها باید مشترک باشند و اگر مسیری پیدا نکرد مقدار تهی را return میکنیم که بدانیم مسیری وجود ندارد.

```
building the path correctly and returns it
also returns the actions the same way
'''
def show_bidirectional_path(self, final_node_r, final_node_p):
    path = []
    action_path = []
    n = final_node_r
    print("****")
    while n.parent != None:
        path.append(n.robot)
        action_path.append(n.action_from_par)
        n = n.parent
    path.append(n.robot)
    path.reverse()
    action_path.reverse()
    n = final_node_p
    while n.parent != None:
        action_path.append(self.inverse(n.action_from_par))
        n = n.parent
        path.append(n.robot)
    return action_path, path
```

برای تابع show bidirectional با استفاده از node نهایی و این که parent آن را ذخیره کرده ایم به صورت عقب گرد parent. را صدا میکنیم تا بتوانیم path مورد نظر را پیدا کنیم.

```
just for inversing the actions for the reversed state
'''
def inverse(self, action):
    if action == "U":
        return "D"
    if action == "D":
        return "U"
    if action == "R":
        return "L"
    if action == "L":
        return "R"
```

در تابع inverse از آنجایی که در یک طرف BFS فرض کردیم که رباط کره را میکشد تا به جواب برسیم حال action های بدست آمده را برعکس میکنیم.

```
def terminal(table, cols, path, action, butter):
    print()
    beauty(table, cols)
    print('----------------------------------------')
    xButter, yButter = butter[0], butter[1]
    check = False
    for i in range (1, len(action)+1):
        print(table[path[i][0]][path[i][1]])
        table[path[i][0]][path[i][1]] += 'r'
        if path[i][0] == xButter and path[i][1] == yButter:
            check = True
            table[path[i][0]][path[i][1]] =
table[path[i][0]][path[i][1]].replace('b', '')
        if action[i-1] == 'U':
            table[path[i][0]+1][path[i][1]] =
table[path[i][0]+1][path[i][1]].replace('r', '')
            if check:
                table[path[i][0]-1][path[i][1]] += 'b'
                xButter -= 1
        if action[i-1] == 'D':
            table[path[i][0]-1][path[i][1]] = table[path[i][0]-
1][path[i][1]].replace('r', '')
            if check:
                table[path[i][0]+1][path[i][1]] += 'b'
                xButter +=1
        if action[i-1] == 'R':
            table[path[i][0]][path[i][1]-1] =
table[path[i][0]][path[i][1]-1].replace('r', '')
            if check:
                table[path[i][0]][path[i][1] + 1] += 'b'
                yButter +=1
        if action[i-1] == 'L':
            table[path[i][0]][path[i][1]+1] =
table[path[i][0]][path[i][1]+1].replace('r', '')
            if check:
                table[path[i][0]][path[i][1] - 1] += 'b'
                yButter -= 1
        check = False
        beauty(table, cols)
```

تابع terminal برای کشیدن نقشه مرحله به مرحله با وجود آپدیت کردن مختصات کره و رباط است.

3) A*گرافی

برای حل این الگوریتم باید heuristic خانه های جدول را بدست میاوریم سپس از f(n) = h(n)+g(n) هزینه پیش بینی شده ی هر خانه تا هدف را میتوانیم بدست آوریم همچنین با استفاده از h خانه های همسایه تصمیم میگیرم به کدام خانه برویم تا کمترین هزینه را تا رسیدن به هدف طی کنیم.

مدل سازی← مانند توضیحی که داده شد از نقطه مبدا یعنی رباط را گسترش داده و برای همسایه های آن مقدار f, h, g را بدست میاورد و بهترین مسیر با کمترین هزینه را انتخاب میکند تا به کره برسد سپس برای راه کره تا بشقاب هم همین کار را انجام میدهد با وجود اینکه هر دفعه به پشت کره برود تا بتواند آن را هل دهد.

تابع شهودی انتخاب شده و بررسی قابل قبول بودن آن ← از تابع Manhattan استفاده میکنیم به اندازه فاصله هر گره تا هدف است که از آنجایی که هم admissible است هم consistence قابل قبول است.

کد این الگوریتم شامل 3 کلاس environment, agent و تابع های A*, A*again, main است که هر یک به شرح زیر هستند:

```
a class for node
we save the f,g,h and parent and position for each node
'''
class Node():

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
        self.robot_path = []
        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position
```

کلاس node برای نگهداری مختصات رباط و parent آن برای یافتن مسیر و path رباط و g به عنوان هزینه که تا الان مصرف شده که در ابتدای کار برابر 0 است و h به عنوان حداقل هزینه پیش بینی شده برای رسیدن به هدف از آن node و f به عنوان جمع h و g به عنوان حداقل هزینه برای کل مسیر تا رسیدن به هدف

```
class for map and finding robot and butters and plates and obstacles
position
and check the available actions for both normal and reversed movements
'''
class environment():
    def __init__(self, map_file):
        self.rows, self.cols = map(int, map_file.readline().split())
        self.table = [map_file.readline().split() for j in
range(self.rows)]

        self.find_people()
        self.find_butter()
        self.robot = self.find_robot()
```

به طور کلی کلاس environment برای ذخیره کلی نقشه است به طوری که با وجود table و تعداد سطر ها و اطلاعات رباط و کره و people به عنوان مقصد مسیر های ممکن را پیدا میکند.

```
'''
to have the plates and butters and position and the map and columns
'''
def map(self):
    return self.cols, self.table, self.people_list, self.butter_list,
self.robot
```

تابع map در کلاس environment برای نگهداری اطلاعات نقشه به صورت list است به طوری که در توابع دیگر به راحتی بتوان به اطلاعات people و butter دسترسی داشت.

```
'''
to find plates(people) position
'''

def find_people(self):
    self.people_list = []
    for i in range(self.rows):
        for j in range(self.cols):
            if 'p' in self.table[i][j]:
                self.people_list.append((i, j))
```

تابع find people برای پیدا کردن p (های) موجود در نقشه به عنوان مقصد کره

```
'''
to find robot position
'''

def find_robot(self):
    for i in range(self.rows):
        for j in range(self.cols):
            if 'r' in self.table[i][j]:
                return i, j
```

تابع find robot برای پیدا کردن r موجود در نقشه به عنوان نقطه شروع

```
'''
to find the butters position
'''

def find_butter(self):
    self.butter_list = []
    for i in range(self.rows):
        for j in range(self.cols):
            if 'b' in self.table[i][j]:
                self.butter_list.append((i, j))
```

تابع find butter برای پیدا کردن b (های) موجود در نقشه

```
'''
to find the obstacles position
'''

def find_obstacles(self):
    self.obstacle_list = []
    for i in range(self.rows):
        for j in range(self.cols):
            if 'x' in self.table[i][j]:
                self.obstacle_list.append((i, j))
```

تابع find obstacles برای پیدا کردن x (های) موجود در نقشه

```
'''
find the correct path for butter
we call the A_star_again so that it gives path for robot
'''


def A_star(start, end, table, cols, robot, robot_paths, parents,
for_robot=False):
    maze = table
    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0
    open_list = []
    closed_list = []
    open_list.append(start_node)

    while len(open_list) > 0:
        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append((current.position, current.robot_path))
                current = current.parent
            return path[::-1]  # Return reversed path

        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
            # Get node position
            node_position = (current_node.position[0] +
new_position[0], current_node.position[1] + new_position[1])
```

```
            # check if the movement is available
            if node_position[0] > (len(maze) - 1) or node_position[0]
< 0 or node_position[1] > (
                    len(maze[len(maze) - 1]) - 1) or node_position[1]
< 0:
                continue

            # check if it goes into obstacle
            if maze[node_position[0]][node_position[1]] == 'x':
                continue
            new_node = Node(current_node, node_position)
            parents[new_node.position] = current_node.position
            children.append(new_node)

        for child in children:
            # Child is on the closed list
            if child in closed_list:
                continue

            # Create the f, g, and h values
            if 'r' in
maze[int(child.position[0])][int(child.position[1])]:
                child.g =
int(maze[int(child.position[0])][int(child.position[1])].replace('r',
''))
            elif 'b' in
maze[int(child.position[0])][int(child.position[1])]:
                child.g =
int(maze[int(child.position[0])][int(child.position[1])].replace('b',
''))
            elif 'p' in
maze[int(child.position[0])][int(child.position[1])]:
                child.g =
int(maze[int(child.position[0])][int(child.position[1])].replace('p',
''))
            else:
                child.g = current_node.g +
int(maze[int(child.position[0])][int(child.position[1])])

            if for_robot is False:
                current = current_node.position  # Where the butter is
                next = child.position  # Where the butter is gonna go
                dx = next[0] - current[0]
                dy = next[1] - current[1]
```

```
                previous = (
                current[0] - dx, current[1] - dy)  # Where we want the
robot to go, so it can push the butter
                parent = parents[child.position]
                robot_loc = None
                try:
                    robot_loc = robot_paths[parent][-1][0]
                except:
                    robot_loc = robot

                # Check obstacles and if it is in range
                if 0 <= previous[0] < len(table) and 0 <= previous[1]
< len(table[1]) and table[previous[0]][
                    previous[1]] != 'x':
                    table_new = copy.deepcopy(table)
                    table_new[current[0]][current[1]] = 'x'
                    robot_path = A_star_again(robot_loc, previous,
table_new)

                    robot_path += [(current_node.position, [])]  #
push

                    child.robot_path = robot_path
                    robot_paths[child.position] = robot_path

            child.h = ((child.position[0] - end_node.position[0]) **
2) + (
                       (child.position[1] - end_node.position[1]) **
2)
            child.f = child.g + child.h

            # Child is already in the open list
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue

            # Add the child to the open list
            if for_robot or (child.position in robot_paths.keys()):
                open_list.append(child)

    return []
```

تابع A* که برای حرکت کره است به این روش کار میکند که هر node که میخواهد گسترش دهد از نقشه بیرون نزند و مانع نباشد سپس برای هر یک از آنها مقدار g, h, f را بدست میاورد تا بتواند مسیر با کمترین هزینه را به ما برگرداند.

```
'''
find the correct path for robot based on the path for butter
'''


def A_star_again(start, end, table):
    maze = table
    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0
    open_list = []
    closed_list = []
    open_list.append(start_node)
    # Loop until you find the end
    while len(open_list) > 0:
        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index
        open_list.pop(current_index)
        closed_list.append(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = Node()
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1]  # Return reversed path

        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0)]:  #
Adjacent squares
            # Get node position
            node_position = (current_node.position[0] +
new_position[0], current_node.position[1] + new_position[1])
```

```
            # check if it is within range
            if node_position[0] > (len(maze) - 1) or node_position[0]
< 0 or node_position[1] > (
                    len(maze[len(maze) - 1]) - 1) or node_position[1]
< 0:
                continue

            # check if it doesnt go to obstacle
            if maze[node_position[0]][node_position[1]] == 'x':
                continue
            new_node = Node(current_node, node_position)
            children.append(new_node)

        for child in children:
            # Child is on the closed list
            if child in closed_list:
                continue

            # Create the f, g, and h values
            if 'r' in
maze[int(child.position[0])][int(child.position[1])]:
                child.g =
int(maze[int(child.position[0])][int(child.position[1])].replace('r',
''))
            elif 'b' in
maze[int(child.position[0])][int(child.position[1])]:
                child.g =
int(maze[int(child.position[0])][int(child.position[1])].replace('b',
''))
            elif 'p' in
maze[int(child.position[0])][int(child.position[1])]:
                child.g =
int(maze[int(child.position[0])][int(child.position[1])].replace('p',
''))
            else:
                child.g = current_node.g +
int(maze[int(child.position[0])][int(child.position[1])])

            child.h = ((child.position[0] - end_node.position[0]) **
2) + (
                    (child.position[1] - end_node.position[1]) ** 2)
            child.f = child.g + child.h

            # Child is already in the open list
```

```
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue

            # Add the child to the open list
            open_list.append(child)
```

پس از حرکت کره و اینکه رباط حتما بتواند کره را به این صورت هل دهد تابع A* again را داریم به این صورت که مسیر بهینه را از مختصات که رباط در حال حاضر قرار دارد را تا جایی که رباط بتواند پشت کره قرار گیرد تا بتواند آن را هول دهد و آن را در robot path ذخیره میکند . همه ی کار های تابع A* را برای چک کردن رباط تا پشت کره را انجام میدهد.

```
'''
gets the butters and plates positions
we give the first butter from list to the first plate on list and
second to second and ...
call the A star and gets the path and calculate the cost based on the
path
print path and actions and cost and goal depth
'''

def main():
    with open("test3.txt", "r") as file:
        env = environment(file)
    cols, table, people, butters, robot = env.map()
    all_path = []
    for i in range(len(butters)):
        actions = []
        butter = butters[i]
        person = people[i]
        cheat_path = []
        robot_paths = {}
        parents = {}
        cost = 0

        path = A_star(butter, person, table, cols, robot, robot_paths,
parents)
        all_path.append(path)
        ##if there is no way !
        if all_path == [[]]:
            print("NO WAY!")
            continue

        robot = path[-1][-1][-1][0]  ##update the robot location
        print(f'Butter From {butter} to {person} goes like:')
        last = None
        for sec in path:
            for i in range(len(sec[1]) - 1):
                compare_x = sec[1][i][0]
                compare_y = sec[1][i][1]
                if i == len(sec[1]) - 2:
                    compared_x = sec[1][i + 1][0][0]
                    compared_y = sec[1][i + 1][0][1]
                else:
                    compared_x = sec[1][i + 1][0]
                    compared_y = sec[1][i + 1][1]
                if last != (compare_x, compare_y):
```

```
                    cheat_path.append((compare_x, compare_y))
                if i == len(sec[1]) - 2:
                    cheat_path.append((compared_x, compared_y))
                if compare_x - compared_x == 0:
                    if compare_y - compared_y == 1:
                        actions.append("L")
                    elif compare_y - compared_y == -1:
                        actions.append("R")
                elif compare_y - compared_y == 0:
                    if compare_x - compared_x == 1:
                        actions.append("U")
                    elif compare_x - compared_x == -1:
                        actions.append("D")
                last = cheat_path[-1]
        print("path:", cheat_path)
        print(actions)
        print("goal depth:", len(actions))
        for j in range(len(cheat_path)):
            if 'r' in table[cheat_path[j][0]][cheat_path[j][1]]:
                cost +=
int(table[cheat_path[j][0]][cheat_path[j][1]].replace('r', ''))
            elif 'b' in table[cheat_path[j][0]][cheat_path[j][1]]:
                cost +=
int(table[cheat_path[j][0]][cheat_path[j][1]].replace('b', ''))
            elif 'p' in table[cheat_path[j][0]][cheat_path[j][1]]:
                cost +=
int(table[cheat_path[j][0]][cheat_path[j][1]].replace('p', ''))
            else:
                cost += int(table[cheat_path[j][0]][cheat_path[j][1]])
        print("cost:", cost)
```

در تابع main به طور کلی با کمک گرفتن از کلاس ها و تابع های بالا میتواند با پیدا کردن path ها مسیر را به صورت U,D,L,R برگرداند به طوری روی کره ها حلقه میزند تا همه ی آنها را به بشقاب برساند همچنین چک میکند که اگر مسیری وجود نداشت بگوید.

مقایسه روش های پیاده شده در موارد زیر:

زمان صرف شده :

IDS ← از آنجایی که به صورت DFS پیش میرود $T(n)=O(m+n)$

Bidirectional BFS ← از آنجایی که به صورت BFS پیش میرود $T(n)= O(V+E)$

A* ← زمان به صورت نمایی زیاد میشود پس $T(n) = 2^{(poly(n))}$

زمان مصرف شده در IDS و Bidirectional BFS برابر و A* > = هستند.

پیچیدگی زمانی :

IDS ← $O(b^d)$

Bidirectional BFS ← $O(b^{(d/2)})$

A* ← به heuristic وابسته است به طوری کلی به صورت زیر است که در بدترین حالت برابر $O(bd)$ است.

$|h(x) - h^*(x)| = O(\log(h^*(x))$

A* بهتر از Bidirectional BFS بهتر از IDS

تعداد گره های تولید شده:

IDS ← تعداد کل گره های جدول تولید میشود تا بتواند مسیر بهینه را برگرداند.

Bidirectional BFS ← تعداد کل گره های جدول تولید میشود تا بتواند مسیر بهینه را برگرداند.

A* ← اما در A* نیازی به تولید تمام گره ها نیست و بعد از تولید شدن یک گره تنها فرزند(ان) بهینه آن را دوباره گسترش میدهیم پس ممکن است همه ی گره ها تولید نشوند.

تعداد گره های تولید شده در IDS و Bidirectional BFS برابر و A* = < هستند.

تعداد گره های گسترش شده:

IDS ← از آنجایی که ضریب انشعاب (k) هر دفعه محدود است به صورت عمقی هر گره را گسترش میدهد اما هر دفعه که k تغییر میکند همه ی آنها از حافظه حذف میشوند بخاطر همین در لحظه تعداد گره های گسترش شده زیاد نیست.

Bidirectional BFS ← همه ی گره هایی که به عنوان فرزند تولید میشوند گسترش پیدا میکنند تا در آخر گره مشترک پیدا شود.

A* ← تعداد گره های گسترش داده در A* به عمق آن وابسته است به طوری که به اندازه وجه مثلث متقارن با زاویه راس b چقدر باشد که اگر به نمای d برسد عدد n را نشان دهد.

که min حالت n برابر ۱ است.(بهترین حالت) $n = 1 + b^* + (b^*)^2 + ... + (b^*)^d$.

Bidirectional search < IDS و مقدار آن در A* به h وابسته است.

عمق راه حل:

IDS ← BFS از آنجایی که cost هر مرحله برابر ۱ است و الگوریتم ها Optimal هستند کمترین عمق ممکن با بهینه ترین راه را برمیگرداند.

Bidirectional BFS ← BFS از آنجایی که cost هر مرحله برابر ۱ است و الگوریتم ها Optimal هستند کمترین عمق ممکن با بهینه ترین راه را برمیگرداند که به اندازه طول مسیر رباط است.

A* ← به دلیل وجود cost امکان دارد مسیری را برگرداند که در عمق بیشتری است اما cost کمتری دارد. که در کل به اندازه طول کل مسیری که رباط طی کرده.

عمق IDS و Bidirectional BFS برابر و =< A* هستند.