حسین لشگری – 9731055 گزارش پروژه اول

نحوه مدل سازی :

از آنجا که مسئله حول حالت های مختلف قرارگیری ربات و کره ها در میز شکل می گیرد در هر سه نوع پیاده سازی از یک کلاس **Table** و کلاس **Node** استفاده شده است.

- یک **Table** در بردارنده ی طول و عرض میز، جایگاه ربات، آرایه ای از جایگاه کره(ها)، شخص(ها)، موانع و هزینه ها است.
- یک **Node** در بردارنده ی حالت میز(**Table**)، گره پدر، عملی که منجر به تولید آن گره شده است و عمق گره از حالت اولیه است.

برای رسیدن به حالت هدف باید با توجه اعمالی که می توان انجام داد گره های جدیدی ساخت تا به هدف رسید؛ برای داشتن لیست فرزندان تولید شده که بسط داده شده اند یا نشده اند از دو لیست **frontier** و **explored** استفاده شده است که با توجه به نوع پیاده سازی ممکن است لیست **explored** مورد نیاز باشد یا نباشد؛ برای مثال در الگوریتم **IDS** چون گره ها پس از رسیدن به انتهای عمق حذف می شوند نیازی به **explored** نیست ولی در دو الگوریتم دیگر نیاز است.

توضیح کلی توابع مشترک :

برای ساختن فرزندان هر گره از توابعی استفاده شده است که تقریبا در هر سه نوع پیاده سازی مشابه اند که کارکرد آن ها در زیر بیان می شود :

- **isInPath(Node, Table)** : این تابع یک حالت از میز و یک گره را می گیرد و بررسی می کند که آیا از مسیر حالت اولیه تا گره و یا در لیست **frontier** این حالت موجود است یا خیر، اگر بود **True** و در غیراینصورت **False** بر می گرداند.
  اگر لیست **explored** نیز وجود داشت آن را نیز بررسی می کند.

- **canDoIt(Node, Action)** : این تابع یک عمل که یک **string** است و یک گره را می گیرد و بررسی می کند که آیا با توجه به وجود موانع، ربات می تواند آن عمل را انجام دهد یا خیر.

- **checkButters(Node, Action)** : این تابع نیز یک عمل و یک گره را می گیرد و بررسی می کند که با فرض اینکه ربات می تواند عمل را انجام دهد آیا کره ای در آن خانه که ربات قصد رفتن دارد وجود دارد و اگر وجود دارد در همان سمت کره مانعی یا کره ای دیگر وجود نداشته باشد.

- **expand(Node)** : این تابع با استفاده از توابعی که در بالا گفته شد، یک گره را می گیرد و فرزندان آن را تولید می کند و به لیست **frontier** اضافه می کند.

- **goalTest(Node)** : این تابع یک گره را می گیرد و بررسی می کند که آیا این گره در حالت هدف است یا خیر.

- **write_path(Node, write)** : این تابع گره هدف را می گیرد و اعمالی که از حالت اولیه تا رسیدن به آن گره انجام شده را توسط **write** در فایل خروجی می نویسد.

الگوریتم IDS :

مطابق کد پایین، تابع **DFS** ، حالت اولیه، **cutoff** و **write** برای نوشتن در فایل خروجی را می گیرد و یک گره از حالت اولیه می سازد و به **frontier** اضافه می کند، سپس طبق الگوریتم از انتهای **frontier** یک گره خارج می کند(برای محاسبه تعداد گره های تولید شده یک متغیر به نام **explored** تعریف شده که با هر خروج از **frontier** یک واحد به آن اضافه شود)، **goalTest** را روی آن انجام می دهد و اگر هدف بود **write_path** می کند و اگر نبود و عمقش از **cutoff** کمتر بود آن را **expand** می کند.

این کار تا زمانی ادامه می یابد که یا به هدف برسد یا **frontier** خالی شود.

تابع **IDS**، با شروع **cutoff** از صفر تا عدد کاربر، **DFS** را اجرا می کند و اگر **DFS** موفق شود **return True** می کند و اگر هیچ کدام از **DFS** ها موفق نشوند در فایل خروجی پیام خطا می نویسد و **return False** می کند.

```
def DFS(init: Table, cutoff: int, write):
    global frontier
    global explored
    init_node = Node(copy.deepcopy(init))
    frontier.append(init_node)
    while len(frontier):
        pop = frontier.pop()
        explored += 1
        if goalTest(pop):
            write_path(pop, write)
            write.write("\n" + str(pop.depth) + "\n" + str(pop.depth))
            return True
        if pop.depth < cutoff:
            expand(pop)
    return False


def IDS(init: Table, max_cutoff: int, write):
    for cutoff in range(0, max_cutoff):
        if DFS(init, cutoff, write):
            return True
    write.write("can't pass the butter")
    return False
```

الگوریتم Bidirectional BFS :

در این الگوریتم علاوه بر لیست **frontier** ، چون جستجوی گرافی انجام می دهیم از لیست **explored** نیز استفاده شده است. همچنین چون جستجو دوطرفه است پس از دو لیست **frontier** و دو لیست **explored** استفاده می کنیم.

برای جستجوی مستقیم در این الگوریتم از توابع کلی که در ابتدا توضیح داده شد استفاده می شود اما برای جستجوی رو به عقب از توابعی دارای کارایی مشابه مانند **pCheckButters** ، **pCanDoIt** و **pExpand** استفاده شده است که نحوه کارکرد آن ها در زیر بیان می شود :

- **pCanDoIt(Node, Action)** : مانند **canDoIt** است با این تفاوت که اگر خانه ای که ربات میخواهد به آن برود علاوه بر موانع، کره هم باشد باز **False** برمی گرداند.
- **pCheckButters(Node, Action)** : تنها بررسی می کند که آیا در خانه ای که با عمل داده شده ربات جا به جا می شود آیا در خانه ای با عمل مخالف آن کره وجود دارد یا خیر.
- **pExpand(Node)** : از توابع بالا استفاده می کند و فرزندانی که می تواند را تولید می کند.

تفاوت این تابع با **expand** معمولی در این است که اگر بخواهیم در جهت معکوس حرکت کنیم هنگام تولید فرزندان،

اولا باید **last_action** فرزندان را مخالف عملی که گره انجام می دهد قرار دهیم مثلا اگر هنگام تولید فرزند به سمت چپ می رویم **last_action** فرزند را راست بگذاریم به این معنی که در جهت مستقیم با عمل راست به پدرش می رسد و نه چپ!

دوما اگر در جهت مخالف حرکت گره، کره ای وجود داشت دو حالت برای فرزند پیش می آید یا کره در همان محل بوده و ربات آن را جا به جا نکرده تا به حالت پدر رسیده! یا کره در مکان رباتِ حالت پدر بوده و رباتِ حالت فرزند آن را هل داده و جا به جا کرده!

بنابراین برخلاف **expand** ، در اینجا هر پدر حداکثر 8 فرزند می تواند تولید کند.(کد **up** در پایین)

```
def pExpand(child: Node):
    global frontier2
    if pCanDoIt(child, "up"):
        up_table = copy.deepcopy(child.state)
        up_table.robot[0] -= 1
        if not isInPath(up_table, 1):
            up_node = Node(up_table, child, "D")
            frontier2.append(up_node)
        if pCheckButters(child, "down"):
            up_table2 = copy.deepcopy(up_table)
            up_table2.butters.remove([child.state.robot[0]+1, child.state.robot[1]])
            up_table2.butters.append([child.state.robot[0], child.state.robot[1]])
            if not isInPath(up_table2, 1):
                up_node2 = Node(up_table2, child, "D")
                frontier2.append(up_node2)
```

- **goalTest()** : در اینجا با صدا زدن تابع، دو لیست **frontier** بررسی می شوند و اگر دو گره دارای حالت یکسان پیدا شد آن را بر می گرداند.
- **write_path(Node1, Node2, write)** : این تابع دو گره می گیرد و از گره اول تا حالت اولیه و از گره دوم تا حالت هدف می رود و اعمالی که انجام شده به همراه هزینه و عمق هدف را در فایل خروجی می نویسد.

از آنجایی که حالت هدفی که مشخص کرده ایم دقیقا حالت ایده آل نیست و ممکن است از آن عبور کند و اعمال اضافه ای انجام دهد تنها اعمالی را که تا هدف اصلی انجام شده است چاپ می کنیم و بقیه را نادیده می گیریم و در محاسبه هزینه و عمق به حساب نمی آوریم.

- **find_goal(Table, robot_place)** : برای مشخص کردن هدف از این تابع استفاده می کنیم. ابتدا حالت اولیه را میگیریم و مکان کره ها را برابر مکان اشخاص قرار می دهیم. سپس ورودی **robot_place** را مکان ربات در حالت هدف قرار می دهیم.

```python
def BDirectBFS(init: Table, cutoff: int, robot_place: list):
    global frontier1
    global frontier2
    global explored1
    global explored2
    init_node = Node(copy.deepcopy(init))
    goal = find_goal(init, robot_place)
    goal_node = Node(copy.deepcopy(goal))
    frontier1.append(init_node)
    frontier2.append(goal_node)
    while len(frontier1) or len(frontier2):
        flag, node1, node2 = goalTest()
        if flag:
            return True, node1, node2
        if len(frontier1):
            pop1 = frontier1[0]
            depth1 = pop1.depth
            while len(frontier1) and depth1 == pop1.depth:
                frontier1.pop(0)
                explored1.append(pop1)
                if pop1.depth < cutoff:
                    expand(pop1)
                if len(frontier1):
                    pop1 = frontier1[0]
        flag, node1, node2 = goalTest()
        if flag:
            return True, node1, node2
        if len(frontier2):
            pop2 = frontier2[0]
            depth2 = pop2.depth
            while len(frontier2) and depth2 == pop2.depth:
                frontier2.pop(0)
                explored2.append(pop2)
                if pop2.depth < cutoff:
                    pExpand(pop2)
                if len(frontier2):
                    pop2 = frontier2[0]
    return False, None, None
```

تابع **BDirectBFS** :

حالت اولیه، **cutoff** و مکان ربات در حالت هدف را می گیرد و گره اولیه و گره هدف را با تابع **find_goal** می سازد و به لیست **frontier** متناظرشان اضافه می کند.

سپس در هر مرحله ابتدا **goalTest** انجام می شود سپس از لیست **frontier1** یک سطح کامل به عمق اضافه می کنیم و دوباره **goalTest** انجام

می دهیم و سپس یک سطح کامل به عمق **frontier2** اضافه می کنیم تا زمانی که یا دو لیست گره ای با حالت یکسان پیدا کنند یا لیست ها خالی شود.

```
def Best_BDirectBFS(init: Table, cutoff: int, write):
    global frontier1
    global frontier2
    global explored1
    global explored2
    global generate_nodes
    global expanded_nodes
    best_node1 = None
    best_node2 = None
    best_dis = 10000
    for person in init.persons:
        flag, temp1, temp2 = BDirectBFS(init, cutoff, person)
        if flag:
            temp = copy.deepcopy(temp2)
            main_goal = copy.deepcopy(temp)
            while temp and temp.state == main_goal.state:
                for butter in temp.state.butters:
                    if temp.state.persons.count(butter) == 0:
                        main_goal = copy.deepcopy(temp.parent)
                        break
                temp = temp.parent
            temp_dis = temp1.depth+temp2.depth-main_goal.depth
            if temp_dis < best_dis:
                best_node1 = temp1
                best_node2 = temp2
                best_dis = temp_dis
        generate_nodes += len(frontier1)+len(frontier2)+len(explored1)+len(explored2)
        expanded_nodes += len(explored1)+len(explored2)
        frontier1 = []
        frontier2 = []
        explored1 = []
        explored2 = []
    if best_node1 and best_node2:
        write_path(best_node1, best_node2, write)
        return True
    write.write("can't pass the butter")
    return False
```

تابع Best_BDirectBFS :

طبق مطلبی که بیان شد براساس اینکه مکان ربات در حالت هدفی که می سازیم کنار کدام شخص باشد ممکن است جواب های مختلفی پیدا کنیم و همه جواب ها ایده آل و دارای کوتاه ترین مسیر نیستند.

پس در این تابع به تعداد اشخاص تابع **BDirectBFS** را صدا می زنیم و جواب دارای کوتاه ترین عمق را به عنوان بهترین مسیر انتخاب می کنیم.

الگوریتم A* :

در این الگوریتم چون گره ها براساس هزینه ها ارزش گذاری می شود برای هر گره یک مقدار **g** ، **h** و **f** که **f** حاصل جمع **g** و**h** است، در نظر گرفته می شود.

مقدار **g** ، حاصل جمع مقدار **g** پدر گره و هزینه خانه ی ربات است و اگر گره پدر نداشت مقدار **g** گره برابر صفر است.

مقدار **h**، حاصل جمع کمترین فاصله ربات تا یکی از کره ها و مقدار **manhattan_distance** است که توضیح داده می شود، است.

تابع شهودی برای الگوریتم :

manhattan_distance(Node) : این تابع ابتدا تمام جایگشت های کره ها در آرایه شان را به دست می آورد سپس برای هر جایگشت فاصله **manhattan_distance** تک تک کره ها در آرایه را با عضو متناظر در آرایه اشخاص به دست می آورد و با هم جمع می کند. سپس کمترین مقداری که توسط یکی از جایگشت ها تولید می شود انتخاب می کند و بر می گرداند.

```
def manhattan_distance(node: Node):
    temp_table = copy.deepcopy(node.state)
    perm_butters = list(itertools.permutations(temp_table.butters))
    sum_dis = []
    for perm in perm_butters:
        temp_dis = 0
        for i in range(0, len(perm)):
            temp_dis += abs(perm[i][0]-temp_table.persons[i][0])+abs(perm[i][1]-temp_table.persons[i][1])
        sum_dis.append(temp_dis)
    return min(sum_dis)
```

بررسی قابل قبول بودن **h** :

- در واقع حداقل هزینه ای که باید انجام شود تا تمام کره ها به اشخاص برسد این است که ابتدا ربات به یک کره برسد برای اینکار حداقل هزینه ای که باید بکند این است که نزدیک ترین کره را انتخاب کند و فاصله ی **manhattan_distance** را بدون در نظر گرفتن هیچ مانعی! و احتساب هزینه 1 برای همه خانه ها ! برود.
- سپس حتما باید فاصله ی بین هر دو کره و شخصی را که انتخاب می کند برود برای اینکار نیز با فرض های بالا کمترین مقدار مسافتی که باید طی کند در تابع **manhattan_distance** محاسبه می شود.

علاوه بر موارد بالا خیلی از هزینه ها مانند جا به جا شدن برای هل دادن و فاصله ای که بعد از رساندن یک کره به هدف تا کره بعدی باید برود و ... در نظر گرفته نمی شود.

پس جمع دو مقدار قبل حتما از هزینه واقعی که باید متحمل شود کمتر است و **h** قابل قبول است.

توابع A_star و min_cost_finder :

برای عملکرد درست الگوریتم برای حالت اولیه یک گره می سازیم و داخل لیست **frontier** قرار می دهیم، سپس گره ای که دارای **f** حداقل است توسط تابع **min_cost_finder** برای **expand** شدن انتخاب می شود سپس **goalTest** روی آن اجرا می شود و در صورت هدف نبودن آن گره **expand** می شود. در انتها یا به هدف می رسیم یا لیست **frontier** خالی می شود که در این صورت پیام خطا در فایل خروجی چاپ می شود.

```
def min_cost_finder():
    global frontier
    global explored
    best = frontier[0]
    for node in frontier:
        if node.f < best.f:
            best = node
    frontier.remove(best)
    explored.append(best)
    return best

def A_star(init: Table, cutoff: int, write):
    global frontier
    global explored
    init_node = Node(copy.deepcopy(init))
    frontier.append(init_node)
    while len(frontier):
        pop = min_cost_finder()
        if goalTest(pop):
            write_path(pop, write)
            write.write("\n" + str(pop.f) + "\n" + str(pop.depth))
            return True
        if pop.depth < cutoff:
            expand(pop)
    write.write("can't pass the butter")
    return False
```

مقایسه 3 الگوریتم پیاده سازی شده :

چون **test case** شماره سوم دارای پیچیدگی بیشتری نسبت به بقیه موارد است، این مورد برای مقایسه بررسی می شود.

دو الگوریتم **IDS** و **BFS** دوطرفه هزینه های واقعی را معیار قرار نمی دهند و تمام هزینه ها را 1 فرض می کنند بنابراین مقایسه این 2 با الگوریتم ***A*** که هزینه های واقعی را در نظر می گیرد درست به نظر نمی رسد.

بازدهی الگوریتم IDS :

```
exec time: 8 minutes and 43.07 seconds
#generated_nodes: 158799 , #expanded_nodes: 158786
```

بازدهی الگوریتم BFS دو طرفه :

```
exec time: 0 minutes and 3.09 seconds
#generated_nodes: 1688 , #expanded_nodes: 1062
```

بازدهی الگوریتم A* :

```
exec time: 0 minutes and 8.96 seconds
#generated_nodes: 1973 , #expanded_nodes: 1244
depth of deepest node: 17 , depth of goal: 16
```

به دلیل اینکه دو الگوریتم اول عمق هدف و بیشترین عمق یکسانی دارند مقدار آن تنها در خروجی چاپ شده است. ولی الگوریتم A* می تواند تا عمق بیشتری رفته باشد و برگردد و هدف را در عمق کمتری بیابد.

همانطور که مشاهده می شود زمان اجرا و تعداد گره های تولید شده الگوریتم BFS دوطرفه به طور قابل توجهی از الگوریتم IDS کمتر است.

زمان اجرای الگوریتم A* نسبت به BFS بیشتر است و این به دلیل معیار قرار دادن هزینه های واقعی در محاسبات است.

خروجی الگوریتم A* :

```
L L U L D R R R U R D D D L D R
17
16
```

خروجی دو الگوریتم IDS و BFS دوطرفه :

```
L L U L D R R R R U R D D D
14
14
```

همانطور که مشاهده می شود هر دو الگوریتم IDS و BFS جواب بهینه با عمق 14 را خروجی می دهند.

الگوریتم A* هم جواب بهینه با هزینه 17 را می دهد ولی با عمق 16.

پیچیدگی زمانی الگوریتم ها:

الگوریتم IDS دارای پیچیدگی $O(b^d)$ که b بیشترین branching factor و d عمق حالت هدف است.

الگوریتم BFS دوطرفه در حالت ایده آل که مقدار branching factor هر دوطرف یکی باشد دارای پیچیدگی $O(b^{d/2})$ است. مثلا در مسئله این پروژه b یک طرف 4 و طرف دیگر 8 بود.

الگوریتم A* براساس تابع شهودی که انتخاب می کنیم دارای پیچیدگی های مختلفی می تواند باشد.