سوال۱) الف:

نوع عامل	معیار کارایی	محیط	محرک ها	حسگر ها
سیستم پرواز خودکار در هواپیما	امنیت، سرعت، قانون، سهولت سفر، حداکثر بهره وری، انتخاب مسیر درست و بهینه	فرودگاه، خطوط هوایی مسافران	هدایت کننده، شتاب دهنده،ترمز، سیگنال،نمایشگر	دوربین ها،سرعت سنج، صفحه کلید،کلومترشمار، شتاب سنج،ارتفاع سنج،GPS
ربات جابه‌جا کنندهی قطعات	جابجایی و قرار دادن قطعات در مکان های درست و سالم نگه داشتن قطعات	کارخانه ها، اپراتورها	هدایت کننده، چرخ اهرام تکان دهنده قطعات	دوربین ها،سنسور سوخت، ارتفاع سنج، فشار سنج(برای بررسی فشار لازم برای حرکت دادن قطعات)، صفحه کلید

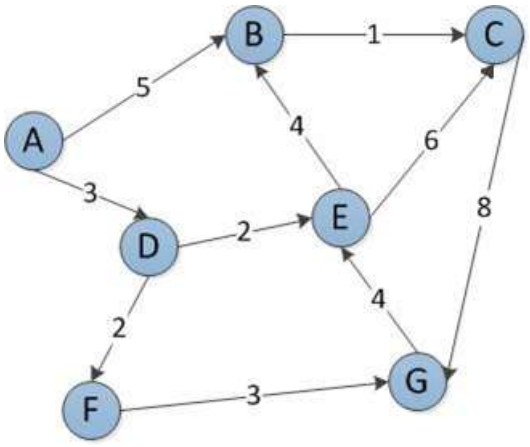
ب:

	ربات انجام دهندهی بازی دوز	شطرنج زماندار
کاملا قابل مشاهده	✓	✓
قطعی	✓	✓
رویدادی	✗	✗
ایستا	✓	✓
گسسته	✗	✓
تک عاملی	✗	✗

سوال۲) آ)این گزاره درست است.دو عامل ناشناخته بودن و پویا بودن و مشاهده پذیری اثری برهم ندارند پس می تواند محیط ناشناخته باشد ولی مشاهده پذیر.

ب)این گزاره نادرست است.عامل مبتنی بر مدل برای حالات و اعمال پیوسته میتوان استفاده کرد. برای مثال راننده تاکسی که در مورد آن در جزوه گفته شده است محیط پیوسته دارد و هر اتفاقی در حین رانندگی میوفته و محیط آن پیوسته است.

ج)این گزاره درست است. اگر یک حالت خاص را در نظر بگیریم که h در آن برابر با صفر باشد در اینصورت می توان گفت که UCS حالت خاصی از A^* است. زیرا در A^* داریم $f = g + h$ و UCS داریم $f = g$.



د)این گزاره نادرست است. در این گراف برای مثال برای رسیدن از E به C ، ۲ مسیر داریم اولی: مسیر مستقیم E به C که هزینه آن برابر است با ۶ . و مسیر دوم از E به B و از B به C که مجموع هزینه آن برابر است با ۵ . خب مسیر دوم مسیر مسیر بهینه است.حال به ازای هر $C > 1$ دیگر مسیر بهینه مسیر اول است.

ه) این گزاره درست است. اگر در A* بدترین حالت را بگیریم یعنی زمانی که h برابر صفر شود.دراین صورت دیگر تفاوتی بین تعداد گره های گسترش با UCS ندارد پس حال اگر بدترین حالت را در نظر نگیریم یعنی h صفر نیست و در حالت می دانیم که h باعث کاهش تعداد گره های گسترش میشه پس، الگوریتم A* تعداد حالات مساوی ویا کمتر نسبت به الگوریتم UCS گسترش می دهد.

و)این گزاره نادرست است. در IDA* اگربا گراف های وزن دار مواجه شویم تکرارها از نظر زمانی ما را دچار مشکل میکند. و هرچقدر تنوع وزن ها در گراف فضای حالت بیشتر باشد، تعداد تکرارها نیز بیشتر خواهد شد و این می تواند آنقدر مسئله را کند کند که دیگر کارا نباشد. و این روش تنها مشکل پیچیدگی فضایی A* را برطرف کرده.

سوال۳) باقی‌مانده ۴۵ بر ۵ برابر است با صفر پس قسمت الف :

طبق فرض داریم:

$$Cost(N,M) \geq x(N) - x(M)$$

$$Cost(N,M) \geq y(N) - y(M)$$

حال باید نشان دهیم در میانگین آن نیز صدق میکند :

$$Cost(N,M) + Cost(N,M) \geq x(N) - x(M) + y(N) - y(M)$$

$$2* Cost(N,M) \geq x(N) - x(M) + y(N) - y(M)$$

$$Cost(N,M) \geq 1/2* (x(N) +y(N)) -(x(M)+ y(M))$$

گزاره درست است.

سوال۴) بخش اول:

آ) فضای حالت این مسئله یک زوج مرتب n تایی است که هر درایه آن عددی هست از ۱ تا m.

ب) اندازه ی فضای حالت برابر است با m^n .

ج) از آنجایی که هر پک‌من در بیشترین حالت ۵ حرکت دارد در هر مرحله که حرکت به چهار جهت بالا، پایین، چپ، راست است و همچنین ماندن در موقعیت فعلیش و خب n پک‌من داریم که این کران بالا را با 5^n نشان می‌دهیم.

د)این کران را میتوان به صورت b^d نوشت که عوامل این کران b برابر است با branching factor که در قسمت ج آن را نوشتیم که برابر است با 5^n و d هم برابر است با بیشترین عمق که در این جا فرض میکنیم دو پک‌من در دورترین حالت ممکن از هم هستند با توجه به داده های مسئله ین فاصله برابر است با m که چون هر دو پک من به سمت همدیگر حرکت میکنند m/2 بیشترین تعداد حرکت است.حال d و b را جایگذاری میکنیم و به $(5^n)^{m/2}$ میرسیم که برابر است با $5^{nm/2}$.

ه) Consistentاست. اگر حالتی رو در نظر بگیریم که هیچ محدودیتی نداریم (حرکت پک‌من ها تنها محدود به بالا پایین چپ راست نباشد و بتواند به صورتی قطری هم حرکت کند و زمین آن دیواری نداشته باشد) . Consistent است چون بیشترین تغییری که در قدر مطلق های h میتواند در هر مرحله رخ بدهد ۲ است که پس یعنی در بیشترین حالت h ۱ واحد کاهش پیدا میکند و هزینه یک حرکت یک است.

گزارش کد پیاده سازی

برنامه ما به ۲ کلاس تقسیم میشه اول کلاس Node و کلاس puzzle .

کلاس اول Node این کلاس برای ساخت درخت ماست که هر گره حالت های مختلف مارو تشکیل میده و در کلاس بعدی سعی داریم که بهترین حالت را در نظر بگیریم تا حالت بهینه را پیدا و انتخاب کنیم.

```
def __init__(self, data, depth, f):
    self.data = data
    self.depth = depth
    self.f = f
```

در این قسمت یک شی میسازیم و برای هر Node یک بخش data که ارایه داده هایی که از پازل کاربر داده هست، depth عمق Nodeامون در درخت و

f همان f-score = h-score + g-score اختصاص میدهیم.

تابع move این تابع برای ساخت لیست لیست و جا به جا کردن خانه خالی (در این برنامه ❋) است.

یک لیست اولیه تعریف میکنیم temp_puz و دانه دانه درایه های ماتریس ورودی را در این لیست میریزیم. حال برای جا بجایی درایه یک متغیر تعریف میکنیم درایه (x_2 , y_2) را در این متغیر میریزیم سپس درایه (x_1 , y_1) را در (x_2 , y_2) میریزیم وسپس متغیر را در درایه (x_1 , y_1) میریزیم.

```
def find(self, puz, x):
    for i in range(0, len(self.data)):
        for j in range(0, len(self.data)):
            if puz[i][j] == x:
                return i, j
```

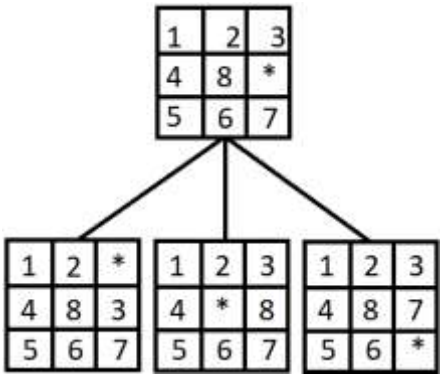
تابع find از این تابع برای پیدا کردن مکان

خانه خالی است در مختصات x وy است.که با استفاده از دو حلقه تو در تو for است تا تمام درایه های ماتریس ورودی تابع بررسی کند.

```
def g_child(self):
    x, y = self.find(self.data, '*')
    list = [[x, y - 1], [x, y + 1], [x - 1, y], [x + 1, y]]
    children = []
    for i in list:
        child = self.move(self.data, x, y, i[0], i[1])
        if child is not None:
            child_node = Node(child, self.depth + 1, 0)
            children.append(child_node)
    return children
```

تابع g_child این تابع برای ساخت Node های جدید است. این Node های جدید فرزندان گره ای که از آن خارج شده اند است.این Node های جدید حالت هایی هایی هستند که خانه خالی میتواند حرکت کند . به این شکل که در هر مرحله با یک if چک میکنیم آیا جهت های حرکتی که در لیست list آورده ایم مجز هست ویا نه بعد یک Node میسازیم و آن را در لیست جدید children میریزیم.

برای مثال :



برای این گره اولیه ۳ حالت داریم که خانه خالی به بالا، پایین، و چپ هست که هر کدام یک Node هستند و در این مرحله عمق درخت را افزایش می‌دهیم.

لیست list از حالت های ممکن حرکت خانه خالی داریم و یک لیست children تعریف میکنیم و با شرط if چک بررسی میکنیم که آیا این حالت ممکن هست یا نه.

(مثلا در مثال بالا خانه خالی نمیتواند به سمت راست بره)

حال وارد کلاس دوم می‌شویم.

```
def __init__(self, size):
    self.n = size
    self.open = []
    self.closed = []
```

در ابتدا یک شی میسازیم که برای هر Node یک بخش n که سایز ماتریس مربعی وارد شده است، open ماتریس موقعیتی است که داریم بررسی میکنیم و close ماتریس موقعیتی است که بررسی ما تمام شده است.

```
def input_m(self):
    puz = []
    for i in range(0, self.n):
        temp = input().split(" ")
        puz.append(temp)
    return puz
```

تابع input_m این تابع تابعی است که از کاربر داده رو میگیره و در یک ارایه میریزیم و این درایه ها را با space جدا میکنیم. با استفاده از split.

```
def h_score(self, start, goal):
    temp = 0
    for i in range(0, self.n):
        for j in range(0, self.n):
            if start[i][j] != goal[i][j] and start[i][j] != '*':
                temp += 1
    return temp
```

تابع h_score برای بدست آوردن این عدد میایم بررسی میکنیم تمام درایه های ارایه اولیه و تمام درایه های ارایه دوم و میبینیم که چه تعداد از این ارایه ها در جای خود نیستند(تعداد درایه هایی که جایشان در اولین ارایه و خروجی خواسته شده یکی نیست) که برای این کار از ۲ حلقه تو در تو for استفاده میکنیم.

```
def f_score(self, start, goal):
    return self.h_score(start.data, goal) + start.depth
```

تابع f_score. این تابع برای محاسبه

f-score = h-score + g-score

که g-score همان عمق node درختمان است.که برای محاسبه از همان start.depth استفاده میکنیم.(start حالتی است که در آن لحظه بررسی میکنیم و goal خروجی ای هست که باید به آن برسیم) و برای h-score هم از تابع بالا استفاده میکنیم.

```
def generate(self):
    print("Enter the start state matrix (using * as the blank block in the puzzle) \n")
    start = self.input_m()
    print("Enter the goal state matrix (using * as the blank block in the puzzle) \n")
    goal = self.input_m()
    start = Node(start, 0, 0)
    start.f = self.f_score(start, goal)
    self.open.append(start)
    ste = 1
    while True:
        cur = self.open[0]
        print("----- step ", ste, "-----")
        ste +=1

        for i in cur.data:
            for j in i:
                print(j, end=" ")
            print("")
        for i in cur.g_child():
            i.f = self.f_score(i, goal)
            self.open.append(i)
        self.closed.append(cur)
        del self.open[0]
        self.open.sort(key=lambda x: x.f, reverse=False)
        print("f --> ", self.f_score(cur,goal))
        print("h --> ", self.h_score(cur.data,goal))
        print("g --> ",cur.depth)
        if (self.h_score(cur.data, goal) == 0):
            break
```

تابع generate. در این تابع است که تمام اتفاقات پیاده سازی می شوند. در ابتدا از کاربر ماتریس مورد نظر کار بر را میگیرد و همچنین ماتریسی که هدف ما قرار هست باشد. این ماتریس ها را با همان تابع input که در بالاتر آن را تعریف کردیم میگیریم.

میایم node اول start را بر میداریم و در لیست open میگذاریم.

سپس وارد حلقه while میشویم.از دو حلقه for برای چاپ ارایه فعلی که بدست امده است در هر مرحله استفاده میکنیم.

سپس اولین Node را حذف کرده و حال برگ های درختمان را sort میکنیم تا در مرحله بعد کوچکترین دا بدست آوریم.

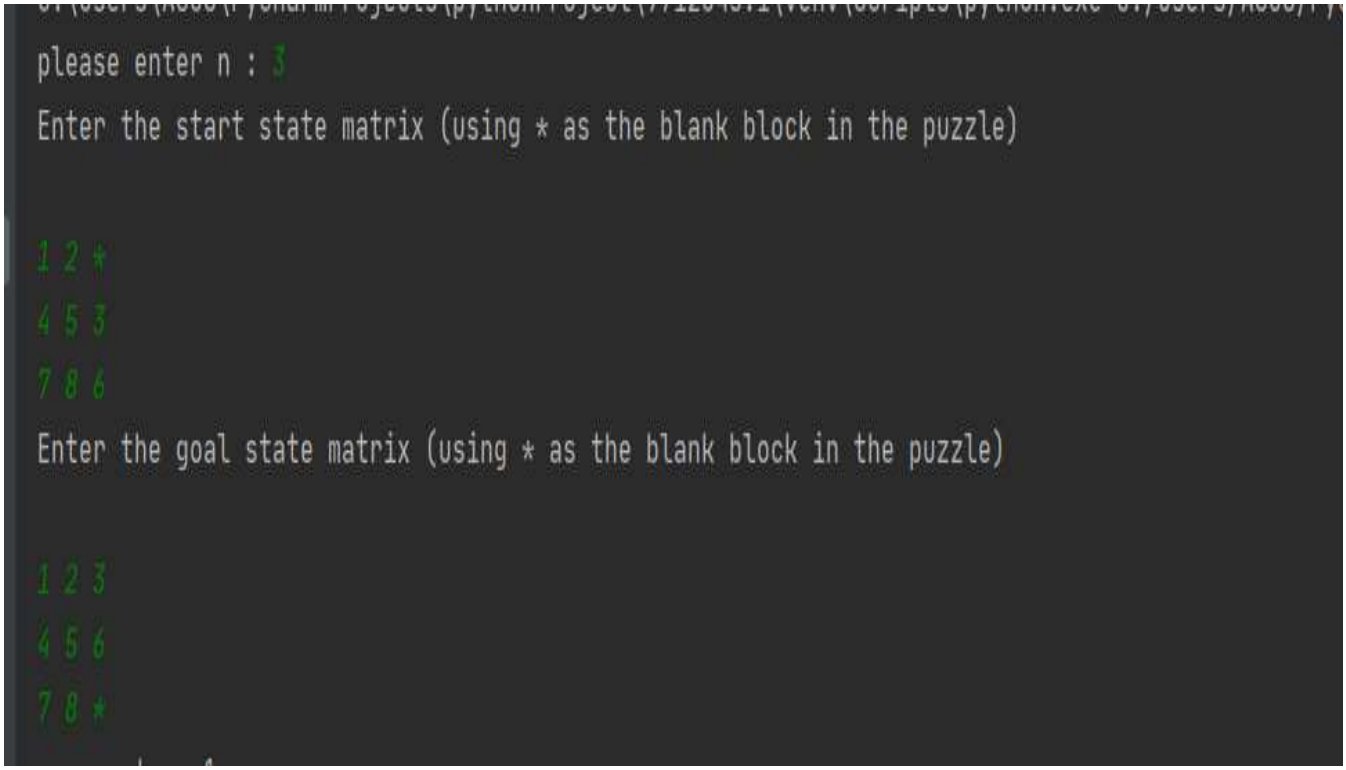
در این حلقه اول بررسی میکنیم که اگرh-score برابر صفر باشد پس ما به ماتریس هدف خود رسیدیم و break میزنیم و از حلقه خارج شده.

سه print هم برای چاپ مقادیر f-score و h-score و g-score است.

```
n = input("please enter n : ")
puz = Puzzle(int(n))
puz.generate()
```

در انتها نیز اول از کاربر n گرفته میشه که اندازه پازل ماست و سپس آن را یک شی از کلاس پازل کرده و تابع اصلیمان را که تابع generate هست را فراخوانی میکنیم.

نمونه ای از ورودی :



و خروجی آن:

```
----- step  1 -----
1 2 *
4 5 3
7 8 6
f -->  2
h -->  2
g -->  0
----- step  2 -----
1 2 3
4 5 *
7 8 6
f -->  2
h -->  1
g -->  1
----- step  3 -----
1 2 3
4 5 6
7 8 *
f -->  2
h -->  0
g -->  2
```