به نام خدا

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

گزارش پروژه اول هوش مصنوعی

نام نگارنده:

محمد امین رحیمی(۹۶۳۱۰۳۰)

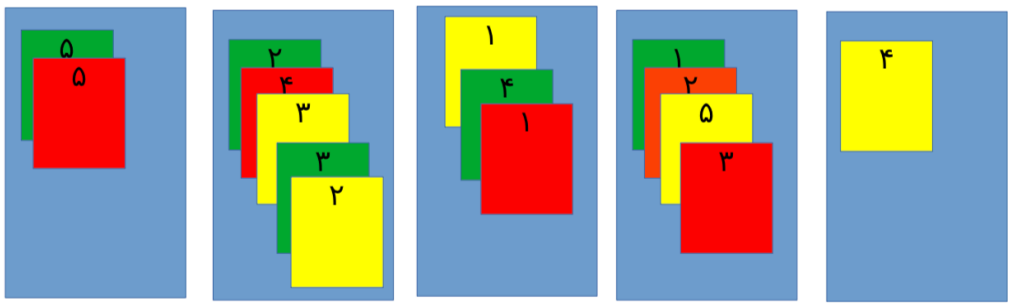
نام استاد درس:

خانم موسوی

دی ماه نود و نه

فرموله‌سازی مسئله:

حالات: هر حالت را به صورت یک nتایی مرتب نشان می‌دهیم که n نشان دهنده تعداد رنگ‌ها می‌باشد. هر یک از عناصر این nتایی مرتب خود یک mتایی مرتب می‌باشند که m نشان دهنده تعداد کارت‌های هر رنگ می‌باشد. هر یک از این عناصر این mتایی مرتب خود یک ۳تایی هستند یکی از آن ها رنگ کارت، یکی ردیفی که کارت در آن قرار دارد و دیگری نشان دهنده این است که کارت عنصر چندم در ردیفی که در آن قرار دارد می‌باشد.(** **توجه شود که نیازی به ذخیره سازی شماره کارت نیست زیرا که کارت ها به ترتیب ذخیر شده اند.)**



به طور مثال حالت بالا را به شکل زیر نشان می‌دهیم.

S=((r,3,1),(r,4,3),(r,4,1),(r,2,4),(r,1,1)), ((y,3,3),(y,2,1),(y,2,3),(y,5,1),(y,4,2)),

((g,4,4),(g,2,5),(g,2,2),(g,3,2),(g,1,2)))

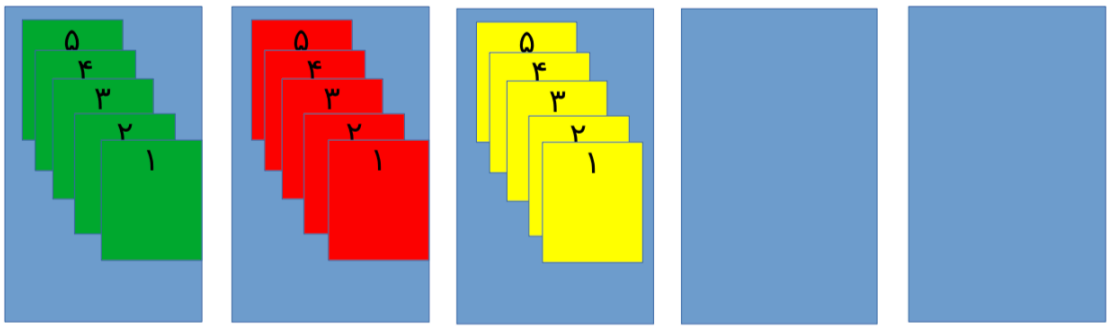
عملیات: با انجام یک عمل در یک حالت یکی از کارت های یک ردیف برداشته شده و به کارت‌های ردیف دیگر اضافه می‌شود. که باید شرایط زیر را داشته باشد.

1 - کارتی که از هر ردیف می‌خواهیم برداریم باید عنصر اول آن ردیف باشد(یعنی رو آن کارت دیگر قرار نداشته باشد). مثلا کارت (r,3,1) با توجه به این که سومین خصیصه برابر یک است پس این کارت یک کارت رویی می‌باشد.

2 - شماره این کارت باید از شماره کارت که در روی ردیف مقصد قرار دارد بیشتر باشد.

حالت اولیه: حالت اولیه را کاربر به عنوان ورودی وارد می‌کند و محدودیت خاصی ندارد. هر عنصر از فضای حالت می‌تواند باشد.

حالت هدف: حالتی که در آن همه عنصر های یک ردیف هم رنگ هستند و به ترتیب هستند و همه‌ی کارت‌های هم‌رنگه در یک ردیف قرار دارند.



به طور مثال حالت بالا یک حالت هدف می‌باشد که به شکل زیر نشان می دهیم

S=(((r,2,1),(r,2,2),(r,2,3),(r,2,4),(r,2,5)), ((y,3,1),(y,3,2),(y,3,3),(y,3,4),(y,3,5)),
((g,1,1),(g,1,2),(g,1,3),(g,1,4),(g,1,5)))

هزینه مسیر: تعداد اعمال انجام شده برای رسیدن به هدف تابع هزینه می‌باشد.

نحوه پیاده‌سازی هر الگوریتم

پیاده سازی هر الگوریتم مطابق شبه کد موجود برای هر الگوریتم در کتاب مرجع می‌باشد.

مهمترین نکات پیاده سازی چگونگی مدل کردن وضعیت‌های می‌باشد که در زیر شرح می‌دهیم.

این پروژه با زبان c پیاده سازی شده و نحوه پیاده‌سازی حالت‌ها در هر سه الگوریتم یکسان است.

بدین صورت که هر حالت را یک struct در نظر گرفته ایم. که این struct شامل یک آرایه n عضوی (که n تعداد ردیف‌ها را نشان می‌دهد) می‌باشد که هر یک از عناصر این آرایه یک اشاره به ابتدای یک لیست پیوندی می‌باشد اولین عنصر این لیست پیوندی معادل اول کارت موجود در هر ردیف می‌باشد و دومین آن هم به همین ترتیب.

شبه کد الگوریتم bfs:

```
struct Queue* frontier  = createQueue();
enQueue(frontier, initialState);
struct Queue* explored = createQueue();

while(!isEmpty(frontier)){
    struct State* state = deQueue(frontier);
    enQueue(explored, state);
    for(int i=0 ; i<numberOfRows ; i++){
        for(int j=0 ; j<numberOfRows ; j++){
            struct State* childState =  childStateCreate(state, i, j);
            if(!isInQueue(explored, childState) && !isInQueue(frontier, childState)){
                if(goalTest(childState)){
                    solution(childState);
                    return 0;
                }
                enQueue(frontier, childState);
            }
        }
    }
}
```

برای نگه داری مجوعه frontier از ساختمان داده صف استفاده شده است زیرا frontier باید بصورت lifo باشد. برای نگه داری مجموعه explored هم از صف استفاده کرده ایم(از ساختمان داده های دیگر هم میتوان استفاده کرد. برای راحتی از صف استفاده شده است).

ابتدا حالت اولیه را وارد صف frontier شده و سپس وارد حلقه می‌شویم. حلقه while تا زمانی که عنصری در frontier وجود دارد اجرا می‌شود. در ابتدای حلقه یک عنصر را از صف frontier خارج می‌کنیم و آن را به explored اضافه میکنیم. بچه های های این حالت را ایجاد می‌کنیم اگر هدف بودند برنامه متوقف شده و راه حل را بر می‌گرداند. اما اگر راه حل نبود و همچنین در صف ها وجود نداشت آنگاه به صف frontier اضافه شده و سپس حلقه از نوع شروع به اجرا می‌کند.

شبه کد الگوریتم ids:

```
int recursiveDLS(struct State* state, int limit){
    if(goalTest(state)){
        solution(state);
        return 1;
    }
    if(limit==0){
        return 0;
    }
    for(int i=0 ; i<numberOfRows ; i++){
        for(int j=0 ; j<numberOfRows ; j++){
            if(i==j)
                continue;
            struct State* childState =  childStateCreate(state, i, j);
            if(childState == NULL)
                continue;
            int result = recursiveDLS(childState, limit-1);
            if(result==1){
                freeState(childState);
                return 1;
            }
            freeState(childState);
        }
    }
    return 0;
}
```

تابع recurvieDLS دو پارامتر دارد. اولی یک حالت می‌باشد و دومی limit می‌باشد که نشان می‌دهد از این حالت چقدر می‌توانیم پیش برویم. در این تابع در ابتدا بررسی می‌شود که آیا این تابع هدف می‌باشد یا نه اگر این تابع هدف باشد کار تمام است. در غیر این صورت در ادامه بررسی می‌کند که آیا limit برابر صفر شده است یا خیر. اگر برابر صفر بود دیگر اجازه بسط نود و رفتن به گره های عمق بیشتر را نداریم. در غیر این صورت وارد حلقه شده و فرزندان این حالت را بوجود اورده و به ازای آن ها تابع را بطور بازگشتی فراخوانی می‌کند ولی limit را یکی کم کرده و بعد فراخونی انجام می‌شود.

شبه کد الگوریتم *A:

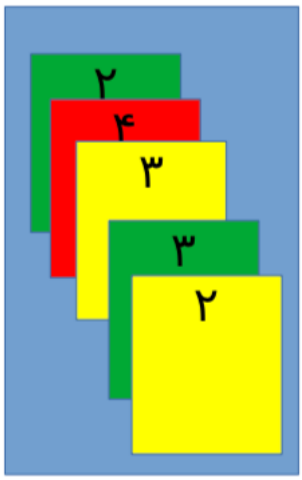
```
struct Queue* frontier  = createQueue('f');
pEnqueue(frontier, initialState);
struct Queue* explored = createQueue('p');

while(!isEmpty(frontier)){
    struct State* state = deQueue(frontier);
    enQueue(explored, state);
    for(int i=0 ; i<numberOfRows ; i++){
        for(int j=0 ; j<numberOfRows ; j++){
            if(i==j)
                continue;
            struct State* childState =  childStateCreate(state, i, j);
            if(childState == NULL)
                continue;
            if(!isInQueue(explored, childState)){
                if(!isInQueue(frontier, childState)){
                    if(goalTest(childState)){
                        solution(childState);
                        return 0;
                    }
                    pEnqueue(frontier, childState);
                }else{
                    upadateQueue(frontier, childState);
                }
            }
            else{
                freeState(childState);
            }
        }
    }
}
```

پیاده سازی این الگوریتم شباهت زیادی به bfs دارد. البته تفاوت‌های عمده‌ای هم وجود دارد. از قبیل این که مجموعه frontier بصورت صف پیاده سازی نشده بلکه بصورت صف اولویت پیاده‌سازی شده است. هر حالتی که f کمتری داشته باشد اولویت بیشتری دارد. نحوه بدست آوردن f به تابع هیوریستیک بستگی دارد که در قسمت بعد شرح داده می‌شود. تفاوت دیگر این است که عناصر موجود درfrontier امکان تغییر دارند.

تابع هیوریستیک:

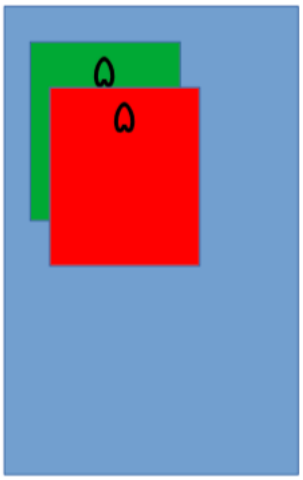
تابع هیوریستیک بدین صورت محاسبه می‌شود که یک رویه برای همه‌ی ردیف‌ها اجرا می‌شود و برای هر ردیف یک عدد برمیگرداند و با جمع این اعداد مقدار تابع هیوریستیک برای هر حالت محاسبه می‌شود. نحوه عملکرد این رویه را بر روی ردیف ها در قابل چند مثال بیان می‌کنیم.

مثال اول:



در این مثال از هر رنگ ۵ کارت وجود دارد. چون ۵ کارت از هر رنگ داریم، درونی ترین عنصر هر حالت باید برابر ۵ باشد اگر درونی ترین عنصر برابر ۵ نباشد یعنی در جایگاه اشتباهی قرار دارد و باید جابجا بشود. همان طور که ملاحظه می‌شود عنصر اخر در این مثال شماره ۲ را دارد که نشان می‌دهد که در جایگاه اشتباهی قرار دارد و باید جابجا بشود. برای جابجا شدن همه ی عناصر که بر روی آن قرار دارند باید از ردیف خارج بشوند. در شکل بالا رویه هیوریستیک برای این ردیف عدد ۵ را بر می‌گرداند.

مثال دوم:



در این مثال عنصر درونی شماره ۵ را دارد که درست می‌باشد. با مشاهده این عنصر این نتیجه را میگیریم که عنصر روی آن باید برابر با شماره۴ و رنگ سبز باشد. اگر این طور نباشد یعنی در جایگاه اشتباهی قرار دارد و باید جابجا بشود. در این مثال عنصر دوم برابر ۵قرمز می‌باشد که با توجه به اینکه عنصر زیر ان ۵سبز است میفهمیم که در جایگاه اشتباهی قرار دارد و باید جابجا بشود. برای جابجا شدن تمامی عنصرهای روی آن هم باید جابجا شوند. در شکل بالا فقط عنصر ۵قرمز است که حتما باید جابجا شود و لذا تابع هیوریستیک عدد ۱ را برمی‌گرداند.

شرح کلی: برای هر ردیف از اخرین عنصر شروع به بررسی میکند و وقتی به عنصری میرسد که در جایگاه نادرستی قرار دارد و باید جابجا شود. عددی را که رویه باز میگرداند برابر است با تعداد کل کارت هایی که روی کارت نادرست قرار دارند بعلاوه یک(زیرا خود کارت هم باید جابجا بشود).

مقایسه الگوریتم ها:

بهینه بودن: الگوریتم های bfs و ids در صورت وجود جواب در عمق محدود جواب بهینه را بازمی‌گردانند. اما الگوریتم A* گرافی به دلیل استفاده از هیوریستیک قابل قبول ممکن است جواب بهینه را بازنگرداند.

حافظه: الگوریتم ids از حافظه بطور خطی استفاده می‌کند. اما الگوریتم‌های A* و bfs بطور نمایی از حافظه استفاده می‌کنند. اما بطور میانگین A* بهتر است از bfs زیرا تعداد نودهای کمتری را تولید می‌کند برای رسیدن بجواب.

گره‌های تولیدشده و بسط داده شده: الگوریتم ids از نظر تعداد گره های تولید شده و بسط داده شده از دو الگوریتم دیگر بدتر است. هم بدلیل اینکه افزایش دادن عمق هم بدلیل پیادسازی درختی نودهای تکراری زیادی را تولید و بسط می‌دهد. الگوریتم bfs هم از نظر نودهای تولید شدهه و بسط داده شده بطور نمایی عمل می‌کند. با وجود اینکه الگوریتم A* در بدترین حالت نمایی است. اما با بهره بردن از یک تابع هیوریستیک خوب تعداد نودهای تولید شده و بسط داده شده را به شدت می‌توانیم کاهش دهیم.