درس مبانی و کاربردهای هوش مصنوعی

گزارش پروژه اول

آرش حریرپوش
9731505

آرش حریرپوش
9731505

فرموله سازی مسئله:

ابتدا یک کلاس به نام Node ایجاد می کنیم تا با کمک آن بتوانیم مسیر رسیدن به جواب را پیدا کنیم؛ به کمک این کلاس اطلاعات هر نود (info)، نود والد (parent) و عملیات انجام شده بر روی والد تا رسیدن به فرزند (action) را نگهداری می کنیم تا بعدا به کمک این اطلاعات مسیر رسیدن به جواب را پیدا کنیم. سپس برای فایل به نام Game ایجاد می کنیم تا به کمک آن شرایط مسئله را فرموله سازی کنیم. برای فرموله سازی مسئله در این فایل چندین تابع تعریف شده است که به معرفی آن ها پرداخته می شود:

- تابع **card_choices**: به کمک این تابع تمام دسته هایی که یک کارت می تواند در آن قرار بگیرد را مشخص می کند. ورودی های این تابع row_num که شماره دسته ای است که قرار است تمام جابجایی های ممکن برای آخرین کارت آن را بدست آوریم و state نیز تمام دسته های موجود را نشان می دهد تا بتوانیم به کمک آن دسته های دیگری که کارت می تواند در آن قرار بگیرد را محاسبه کنیم. خروجی این تابع یک آرایه می باشد که تمام جابجایی های ممکن بین آخرین کارت دسته عنوان شده و بقیه دسته ها را نشان می دهد.
- تابع **actions**: این تابع به کمک تابع card_choices تمام حالات ممکن برایی جابجایی کارت های یک نود را بررسی می کند و در نهایت تمام جابجایی های ممکن را بر می گرداند. ورودی این تابع تنها state می باشد که برابر دسته های موجود در یک نود است. و خروجی این نود نیز یک آرایه می باشد که تمام action هایی که آن نود می تواند انجام دهد را نشان می دهد. در آرایه خروجی هر المان یک tuple می باشد که در آن عنصر اول شماره سطری است که آخرین کارت آن برداشته می شود و عنصر دوم شماره دسته ای است که کارت برداشته شده در آن قرار می گیرد.
- تابع **do_action**: به کمک این تابع نود بچه تولید می شود. ورودی های این تابع state که تمام دسته های یک نود را نشان می دهد، i شماره سطری که قرار است آخرین کارت آن را جابجا کنیم و j شماره دسته ای که قرار است کارت برداشته شده را در آن قرار دهیم. خروجی این تابع تمام دسته های موجود در نود بچه می باشد.
- تابع **check_color**: به کمک این تابع بررسی می کنیم که آیا رنگ های هر دسته یکسان می باشد یا خیر، ورودی این تابع کارت های یک دسته می باشد و خروجی آن یک Boolean می باشد که در صورتی که همه رنگ های آن دسته یکسان باشد True و در غیر این صورت False است.
- تابع **check_order**: به کمک این تابع بررسی می کنیم که آیا همه کارت های در یک دسته به ترتیب نزولی قرار گرفته اند یا خیر، ورودی این تابع کارت های یک دسته می باشد و خروجی آن یک Boolean می باشد که در صورتی که شماره کارت ها به صورت نزولی قرار گرفته باشند True و در غیر این صورت False است.
- تابع **goal_test**: در این تابع بررسی می کنیم که آیا نود مورد نظر هدف می باشد یا خیر، ورودی این تابع state که دسته های موجود در یک نود می باشد و m که برابر تعداد رنگ ها می باشد، در اینجا به کمک دو تابع check_color و check_order بررسی می کنیم که آیا کارت ها در هر دسته هم رنگ و به ترتیب نزولی قرار

گرفته اند یا خیر، و در صورتی که همه کارت ها به ترتیب رنگ و شماره کارت به صورت نزولی قرار گرفته باشند بررسی می کنیم که آیا تعداد ستون های دارای کارت در این نود برابر تعداد رنگ های می باشد یا خیر، بنابراین خروجی این تابع یک Boolean می باشد که در صورتی که کارت های همه دسته ها هم رنگ و به صورت نزولی قرار گرفته باشند همچنین تعداد دسته های دارای کارت نیز برابر تعداد رنگ ها باشد True است در غیر این صورت False.

(فایل Node.py مربوط به کلاس Node و فایل Game.py مربوط به فرموله سازی مسئله می باشد.)

1. این مسئله را به کمک الگوریتم جستوجوی گرافی اول سطح حل کنید.

برای حل کردن این مسئله به کمک BFS ابتدا کلاس Search را ایجاد می کنیم. در این کلاس الگوریتم BFS پیاده سازی شده است و همچنین مراحل رسیدن به جواب نمایش داده می شود، برای پیاده سازی الگوریتم عنوان شده و نمایش مراحل رسیدن به مسئله چندین تابع در این کلاس تعریف شده است که به معرفی آن ها پرداخته می شود:

- تابع **BFS_GraphSearch**: در این تابع الگوریتم عنوان شده پیاده سازی شده است، بدین صورت که ابتدا بررسی می کند که نود شروع برابر حالت هدف هست یا خیر، در صورتی که نود شروع هدف باشد آن را به عنوان خروجی بر می گرداند، در غیر این صورت این نود را به صف frontier اضافه می کند، و پس از آن تا زمانی که همه نود های این صف بررسی شوند کار های زیر در یک حلقه اجرا می شود:
  - ابتدا یک نود از صف frontier برداشته می شود.
  - این نود به لیست explored اضافه می شود.
  - سپس به کمک تابع actions در فایل Game تمام جابجایی های ممکن برای این دسته بندی کارت ها محاسبه می شود.
  - پس از آن در یک حلقه دیگر تمام بچه هایی که از این نود می تواند ایجاد را بررسی می کنیم بدین صورت که به کمک تابع do_action در فایل Game به ترتیب هر یک از جابجایی ها را که باعث ایجاد یک نود جدید می شود انجام می دهیم، پس از انجام هر جابجایی در صورتی که نود ایجاد شده در صف frontier و لیست explored نباشد، ابتدا بررسی می کنیم که آیا این نود هدف می باشد یا خیر(goal_test زمانی که یک نود تولید (generate) می شود چک می شود.)، در صورتی که هدف باشد آن نود را به عنوان نتیجه جستجو بر می گردانیم، و در غیر این صورت در صورتی که این نود در صف frontier نباشد آن را به صف frontier اضافه می کنیم.

آرش حریرپوش
9731505

در صورتی که هیچ جوابی برای مسئله پیدا نشود یک آرایه تهی به عنوان خروجی جستجو برگردانده می شود.

(این جستجو بر اساس pseudo code موجود در اسلاید ها پیاده سازی شده است.)

- تابع **final_actions**: این تابع نود هدف را دریافت می کند و به کمک آن تمام والد های نود ها تا رسیدن به None را در یک لیست اضافه می کند و در نهایت آن لیست را به عنوان خروجی بر می گرداند.
- تابع **print_actions**: این تابع همه والد های بدست آمده در مرحله قبل و نود هدف را دریافت می کند و با pop کردن از لیست والد های مرحله قبل مسیر رسیدن به پاسخ را از نود شروع نشان می دهد و همچنین در هر مرحله دسته بندی کارت ها و عملیات انجام شده در هر مرحله را نشان می دهد.

سپس در نهایت با استفاده از این کلاس الگوریتم جستجو را انجام می دهیم؛ خروجی این جستجو به ازای ورودی زیر برابر است با:

ورودی:

5 3 4

5g 4g 3g 2g 1g

5r 4r 3r

5y 4y 3y 2r 1y

1r 2y

خروجی:

آرش حریرپوش
9731505

```
Please Enter n, m, k:
5 3 4
5g 4g 3g 2g 1g
5r 4r 3r
5y 4y 3y 2r 1y
1r 2y
Processing The Input.....

Step:  0
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'r'), (1, 'y')]
[(1, 'r'), (2, 'y')]
Action:  (2, 3)
--------------------------------------------------
Step:  1
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'r')]
[(1, 'r'), (2, 'y'), (1, 'y')]
Action:  (2, 1)
--------------------------------------------------
Step:  2
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y')]
[(1, 'r'), (2, 'y'), (1, 'y')]
Action:  (3, 1)
```

آرش حریرپوش
9731505

```
------------------------------------------------
Step:  3
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r'), (1, 'y')]
[(5, 'y'), (4, 'y'), (3, 'y')]
[(1, 'r'), (2, 'y')]
Action:  (3, 2)
------------------------------------------------
Step:  4
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r'), (1, 'y')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'y')]
[(1, 'r')]
Action:  (1, 2)
------------------------------------------------
Step:  5
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'y'), (1, 'y')]
[(1, 'r')]
Action:  (3, 1)
------------------------------------------------
Final Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r'), (1, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'y'), (1, 'y')]
[]
```

```
Goal Has Found In 0.04388236999511719 Seconds.
Explored Nodes:  77
Generated Nodes:  109
Depth Search:  6
```

همانطور که مشاهده می شود این الگوریتم با بررسی 77 و تولید 109 نود، نود هدف را تقریبا در 0.04 ثانیه در عمق 6 پیدا کرده است.

(کلاس Search در این سوال در فایل BFS_GraphSearch.py و دریافت ورودی و انجام جستجو در فایل main_BFS.py قرار دارد)

2. مسئله را به کمک الگوریتم جست و جوی درختی اول عمق با افزایش تدریجی عمق (عمق اولیه باید از پارامترهای قابل تغییر در برنامه شما باشد و عمق باید یکی یکی افزایش پیدا کند) حل کنید.

برای حل این مسئله نیز همانند سوال قبل یک کلاس Search ایجاد می شود که در آن چندین تابع تعریف شده است که به معرفی آن ها پرداخته می شود:

- تابع **recursive_DLS**: در این تابع با دریافت نود شروع (node) و محدودیت عمق (limit) به جستجوی فضای حالت به کمک DFS پرداخته می شود. بدین صورت که ابتدا نود بررسی می شود (goal_test زمانی که یک نود گسترش می یابد (explored) می یابد چک می شود.) و در صورتی که نود هدف باشد آن نود را به عنوان هدف بر می گردانیم و در غیر این صورت در صورتی که محدودیت عمق وارد شده برابر صفر باشد نود ورودی را به همراه عبارت cutoff به عنوان خروجی بر می گرداند، و در غیر این صورت ابتدا به کمک تابع actions در فایل Game همه جابجایی های ممکن برای یک نود را محاسبه می کنیم و سپس در حلقه زیر نود های بچه این گراف را ایجاد می کنیم، عملیات زیر در یک حلقه تکرار می شود:
  - ابتدا action مورد نظر را بر روی نود مورد نظر انجام می دهیم و سپس نود بچه را ایجاد می کنیم.
  - سپس این تابع (recursive_DLS) را به صورت بازگشتی برای نود بچه و حد یکی کمتر از مقدار ورودی فراخوانی می کنیم.
  - سپس در صورتی که نتیجه فراخوانی تابع عنوان شده برابر cutoff باشد متغیر مرتبط با رخ دادن cutoff را True می کنیم و در صورتی که نتیجه فراخوانی fail نباشد خروجی بدست آمده در فراخوانی را بر می گردانیم.
  - پس از اتمام حلقه در صورتی که cutoff رخ داده باشد نود را بر می گردانیم و در غیر این صورت یعنی عملیات موفقیت آمیز نبوده و fail برگردانده می شود.

تابع **depth_limited_search**: این تابع با دریافت محدودیت عمق (limit)، محدودیت عمق و نود شروع را به تابع recursive_DLS می دهد و جواب آن را به عنوان خروجی بر می گرداند.

تابع **iterative_deepening_search**: این تابع عمق اولیه را به عنوان ورودی دریافت می کند و سپس در یک حلقه تابع depth_limited_search را از عمق اولیه محاسبه می کند و در صورتی که جوابی پیدا نشده باشد، تا زمانی که جواب مسئله پیدا شود عمق یک واحد اضافه می گردد و تابع depth_limited_search فراخوانی می شود.

(این جستجو بر اساس pseudo code موجود در اسلاید ها پیاده سازی شده است.)

- توابع **final_actions** و **print_actions**: این توابع همانند سوال قبل می باشد.

سپس در نهایت با استفاده از این کلاس الگوریتم جستجو را انجام می دهیم؛ و عمق اولیه برابر 4 در نظر گرفته می شود، خروجی این جستجو به ازای ورودی زیر برابر است با:

ورودی:

5 3 4

5g 4g 3g 2g 1g

5r 4r 3r

5y 4y 3y 2r 1y

1r 2y

خروجی:

```
Please Enter n, m, k:
5 3 4
5g 4g 3g 2g 1g
5r 4r 3r
5y 4y 3y 2r 1y
1r 2y
Processing The Input.....

Step:  0
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'r'), (1, 'y')]
[(1, 'r'), (2, 'y')]
Action:  (2, 3)
--------------------------------------------------
Step:  1
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'r')]
[(1, 'r'), (2, 'y'), (1, 'y')]
Action:  (2, 1)
--------------------------------------------------
Step:  2
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y')]
[(1, 'r'), (2, 'y'), (1, 'y')]
Action:  (3, 1)
```

آرش حریرپوش
9731505

```
------------------------------------------------
Step:  3
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r'), (1, 'y')]
[(5, 'y'), (4, 'y'), (3, 'y')]
[(1, 'r'), (2, 'y')]
Action:  (3, 2)
------------------------------------------------
Step:  4
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r'), (1, 'y')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'y')]
[(1, 'r')]
Action:  (1, 2)
------------------------------------------------
Step:  5
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'y'), (1, 'y')]
[(1, 'r')]
Action:  (3, 1)
------------------------------------------------
Final Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r'), (1, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'y'), (1, 'y')]
[]
```

```
Goal Has Found In 0.6333494186401367 Seconds.
Explored Nodes:  2485
Generated Nodes:  10968
Depth Search:  6
```

همانطور که مشاهده می شود این الگوریتم با بررسی 2485 و تولید 10968 نود، نود هدف را تقریبا در 0.63 ثانیه در عمق 6 پیدا کرده است. با توجه به نتایج بدست آمده در این سوال در می یابیم که الگویتم BFS بسیار سریع تر از الگوریتم IDS عمل می کند که به این دلیل می باشد که در BFS هر نود یک بار گسترش پیدا می کند اما در IDS هر نود ممکن است چندین بار مشاهده شود و به همین دلیل تعداد نود های تولید شده بسیار بیشتر از BFS می باشد.

(کلاس Search در این سوال در فایل IDS_TreeSearch.py و دریافت ورودی و انجام جستجو در فایل main_IDS.py قرار دارد)

3. ابتدا یک هیوریستیک قابل قبول و غیر بدیهی برای این مسئله ارائه داده و علت قابل قبول بودن آن را در گزارش خود ذکر کنید. سپس مسئله را به کمک الگوریتم جستوجوی گرافی * A حل کنید.

در این مسئله هزینه را برابر عمق نود در نظر می گیریم چرا که هر بار با تغییر یک کارت در نود والد فرزند ایجاد می گردد، بنابراین با هر مرحله پایین رفتن در این مسئله هزینه یک واحد افزایش می یابد؛ و با توجه به اینکه برای هر دسته باید همه رنگ های کارت ها یکسان باشد بنابراین در صورتی که تعداد رنگ های مشاهده شده در یک دسته بیش از یک باشد بدین معنی است که دسته بندی درست نمی باشد و میزان بد بودن دسته بندی را با تعداد رنگ های مشاهده شده نمایش می دهیم که این مقدار را برابر different_colors در نظر می گیریم، همچنین یکی دیگر از روش های ارزشیابی برای میزان بد بودن دسته بندی تعداد کارت هایی می باشد که که از همسایه سمت چپ خود کوچک تر و از همسایه سمت راست خود بزرگ تر نیستند به عبارت دیگر یعنی از نظر شماره کارت در جای مناسبی در آن دسته بندی قرار ندارند که این مقدار را نیز برابر disordered_cards در نظر می گیریم، و همچنین می دانیم که در هر دسته بندی باید به تعداد n تا کارت قرار داشته باشد، بنابراین کمترین فاصله هر دسته با n را محاسبه و با یک دیگر جمع می کنیم، که این مقدار را برابر diff_num در نظر می گیریم؛ بنابراین هیوریستیک زیر را برای حل این مسئله در نظر می گیریم:

$$Heuristic = 0.7 * \max(disordered\_cards, different\_colors - 1) + 0.3 * diff\_num$$

همچنین هیوریستیک عنوان شده در این مسئله همواره بیشتر از هزینه اصلی نمی باشد چرا که در صورتی که بیش از یک رنگ در یک دسته بندی وجود داشته باشد حداقل با جابجایی یک کارت می توان این مشکل را برطرف کرد و در صورتی که یک کارت در مکان مناسبی نیز قرار نداشته حداقل در یک حرکت نیاز است تا به دسته ای وارد شود که از نظر مکانی برای آن کارت مناسب می باشد، در صورتی که در یک دسته بندی کارت ها از نظر ترتیب شماره کارت ها بسیار نامنظم باشند یا از نظر تنوع رنگی کارت های با رنگ های متنوعی در هر دسته قرار داشته باشد، در صورتی که تعداد کارت هایی که از نظر شماره کارت به ترتیب قرار نگرفته اند را با m و مجموع تعداد رنگ های اضافی مشاهده شده باشد را با n نشان دهیم، برای اینکه بخواهیم این مشکلات را برطرف کنیم حداقل نیاز است تا به اندازه max(m, n) کارت را جابجا کنیم تا نامنظمی که بیشتر مشاهده شده است را برطرف کنیم و ممکن است با انجام این کار رنگ های مشاهده شده اضافی در دسته ها نیز برطرف شود. اما ممکن است با انجام این جابجایی ها همچنان تعداد کارت های دسته های ما یکسان نباشد و تعداد دسته های دارای کارت مطلوب نباشد برای همین diff_num * 0.3 را نیز در نظر می گیریم تا بتوانیم مسئله را از طریق یک فاکتور متفاوت تر نیز بررسی کنیم. همچنین در صورتی که یک دسته کمتر از n کارت داشته باشد یا باید همه کارت های آن های دسته جابجا شوند یا باید تعدادی کارت به این دسته اضافه شود تا این دسته معتبر شود، با توجه به اینکه در این هیوریستیک کمترین مقدار را در نظر می گیریم به این معنی است که همواره کمترین تعداد جابجایی کارت ها را در

نظر می گیریم، یعنی در صورتی که در هر ستون نیاز باشد تا 5 کارت وجود داشته باشد در صورتی که یک دسته 3 کارت داشته باشد مقدار این هیوریستیک 2 می شود در حالی که ممکن است حتی نیاز باشد تا هر سه این کارت ها جابجا شوند به همین دلیل مقدار هزینه اصلی همواره از این هیوریستیک کمتر نمی باشد. حال برای ترکیب مقادیر عنوان شده و ایجاد یک هیوریستیک مناسب برای مسئله مقادیر عنوان شده را با در نظر گرفتن ضرایبی با یک دیگر جمع می کنیم که این ضرایب به صورت تجربی تنظیم شده اند، (مسئله به ازای ورودی های مختلف با ضرایب مختلف بررسی شده و در نهایت مناسب ترین ضرایب انتخاب شده اند). بنابراین با توجه به اینکه همه مقادیر عنوان شده همواره حد پایین فاصله نود تا هدف را از جهات مختلفی بررسی می کنند مقدار هیوریستیک نهایی این مسئله بزرگ تر از هزینه واقعی نمی باشد بنابراین این هیوریستیک قابل قبول می باشد.

برای حل این مسئله نیز همانند سوال قبل یک کلاس Search ایجاد می شود که در آن چندین تابع تعریف شده است که به معرفی آن ها پرداخته می شود:

- تابع **calculate_different_colors**: این تابع با دریافت کارت های مشاهده شده در یک دسته تعداد رنگ های اضافی مشاهده شده را محاسبه می کند و به عنوان خروجی بر می گرداند.
- تابع **calculate_disordered_cards**: این تابع با دریافت کارت های مشاهده شده در یک دسته تعداد کارت هایی را که از نظر شماره در جای درستی قرار ندارند را محاسبه می کند و به عنوان خروجی بر می گرداند.
- تابع **calculate_heuristic**: این تابع دسته های موجود در یک نود را دریافت می کند و به کمک دو تابع calculate_different_colors و calculate_disordered_cards بخشی از مقدار هیوریستیک نود و سپس با اضافه کردن مجموع کمینه تعداد کارت هایی که باید بین دسته ها جابجا شوند مقدار هیوریستیک نود را محاسبه می کند.
- تابع **find_min_node**: این تابع نود های موجود در لیست frontier را بررسی می کند و نودی را که کمترین مقدار هزینه تخمینی تا نود هدف (جمع مقدار هیوریستیک با هزینه طی شده) را دارد برای گسترش در مرحله بعد انتخاب می کند.
- تابع **AStar_GraphSearch**: این تابع تقریبا همانند تابع BFS_GraphSearch در سوال اول عمل می کند با این تفاوت که در هر مرحله به کمک تابع find_min_node نودی را گسترش می دهد که از نظر هزینه تخمینی تا نود هدف کمترین مقدار را دارد، و در هر مرحله برای نود های ایجاد مقدار هیوریستیک را به کمک تابع calculate_heuristic محاسبه می کند و با هزینه طی شده تا آن نود جمع می کند و سپس آن را به لیست frontier اضافه می کند.
- توابع **final_actions** و **print_actions**: این توابع همانند سوال های قبل می باشد.

آرش حریرپوش
9731505

سپس در نهایت با استفاده از این کلاس الگوریتم جستجو را انجام می دهیم؛ خروجی این جستجو به ازای ورودی زیر برابر است با:

ورودی:

5 3 4

5g 4g 3g 2g 1g

5r 4r 3r

5y 4y 3y 2r 1y

1r 2y

خروجی:

```
Please Enter n, m, k:
5 3 4
5g 4g 3g 2g 1g
5r 4r 3r
5y 4y 3y 2r 1y
1r 2y
Processing The Input.....

Step:  0
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'r'), (1, 'y')]
[(1, 'r'), (2, 'y')]
Action:  (2, 3)
--------------------------------------------------
Step:  1
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'r')]
[(1, 'r'), (2, 'y'), (1, 'y')]
Action:  (2, 1)
--------------------------------------------------
Step:  2
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y')]
[(1, 'r'), (2, 'y'), (1, 'y')]
Action:  (3, 1)
```

آرش حریرپوش
9731505

```
--------------------------------------------------
Step:  3
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r'), (1, 'y')]
[(5, 'y'), (4, 'y'), (3, 'y')]
[(1, 'r'), (2, 'y')]
Action:  (3, 2)
--------------------------------------------------
Step:  4
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r'), (1, 'y')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'y')]
[(1, 'r')]
Action:  (1, 2)
--------------------------------------------------
Step:  5
Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'y'), (1, 'y')]
[(1, 'r')]
Action:  (3, 1)
--------------------------------------------------
Final Node:
[(5, 'g'), (4, 'g'), (3, 'g'), (2, 'g'), (1, 'g')]
[(5, 'r'), (4, 'r'), (3, 'r'), (2, 'r'), (1, 'r')]
[(5, 'y'), (4, 'y'), (3, 'y'), (2, 'y'), (1, 'y')]
[]
```

```
Goal Has Found In 0.010987520217895508 Seconds.
Explored Nodes:  32
Generated Nodes:  76
Depth Search:  6
```

همانطور که مشاهده می شود این الگوریتم با بررسی 32 و تولید 76 نود، نود هدف را تقریبا در 0.01 ثانیه در عمق 6 پیدا کرده است. جدول زیر نتایج بدست آمده از این سه جستجو را به ازای تست انجام شده بر روی آن ها نمایش می دهد:

آرش حریرپوش
9731505

Search	Elapsed Time (Sec)	Explored Nodes	Generated Nodes	Depth Solution
BFS_Graph	0.04	77	109	6
IDS_Tree	0.63	2485	10968	6
A*_Graph	0.01	32	76	6

همانطور که در جدول بالا مشاهده می شود، الگوریتم جستجوی گرافی *A در سریع ترین زمان ممکن و با مشاهده کمترین نود توانسته است تا نود هدف را پیدا کند. و پس از آن الگوریتم جستجوی گرافی BFS توانسته است با مشاهده تعداد بیشتری نود، نود هدف را بیابد. اما در الگوریتم جستجوی درختی IDS تعداد نود های مشاهده شده به شدت افزایش می یابد و به همین دلیل برای پیدا کردن نود هدف به زمان بیشتری نیاز دارد.

به طور کلی هر سه این الگوریتم ها نود هدف را در یک عمق پیدا می کنند، اما با توجه به اینکه در الگوریتم جستجوی درختی IDS ما هر نود را چندین بار مشاهده می کنیم به همین دلیل به زمان بیشتری نسبت به دیگر الگوریتم های جستجوی عنوان شده نیاز داریم تا این الگوریتم نود هدف را پیدا کند. اما در دو الگوریتم دیگر با توجه به اینکه هر نود را تنها یک بار مشاهده می کنیم بنابراین به زمان بسیار کمتری نسبت به IDS نیاز داریم. با تعریف یک هیوریستیک مناسب در الگوریتم جستجوی گرافی *A نود ها را بر اساس نزدیک بودن به نود هدف گسترش می دهیم، بنابراین الگوریتم جستجوی گرافی *A معمولا با مشاهده نود های کمتری و در زمان کمتری نسبت به BFS نود هدف را پیدا می کند.

(کلاس Search در این سوال در فایل AStar_GraphSearch.py و دریافت ورودی و انجام جستجو در فایل main_AStar.py قرار دارد)