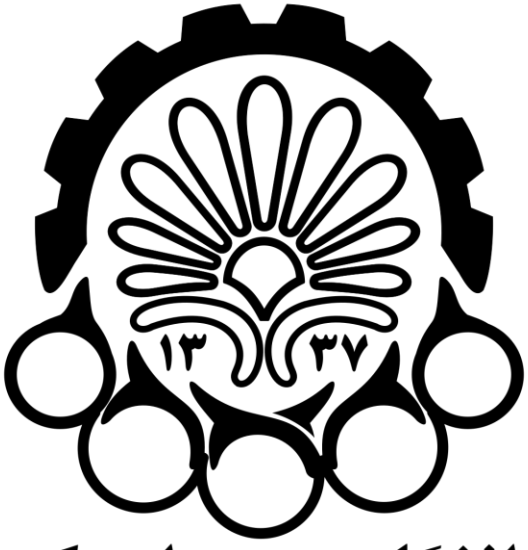


دانشگاه صنعتی امیرکبیر

(پلی تکنیک تهران)

گزارش پروژه‌ی اول درس مبانی هوش مصنوعی

استاد روشن‌فکر

آرمین ذوالفقاری ۹۷۳۱۰۸۲ و امیرحسین رجب‌پور ۹۷۳۱۰۸۵

توضیحات اولیه:

در فایل Node.py ساختار Node و initial_node طراحی شده‌اند که در الگوریتم‌ها برای ذخیره‌سازی حالت‌های مسئله کاربرد دارند. کلاس Node دارای یک محیط (محیط مسئله در آن حالت)، مختصات ربات در آن حالت، عمق آن حالت، حرکتی که ربات در آن محیط انجام داده‌است (و هزینه‌ی f و g گره که در الگوریتم A* مورد نیاز است) و در آخر والد آن حالت می‌باشد.
همچنین کلاس initial_node نیز وجود دارد که برای حالتی می‌باشد که نیاز به یک حالت اولیه داریم. زیرا حالت اولیه والد ندارد ساختار این کلاس متفاوت است و فقط دارای محیط و عمق می‌باشد.
همچنین برای برخی جزئیات در کد، کامنت‌گذاری نیز شده‌است.

توابع کمکی در فایل **AdditionalFunctions**:

یکسری تابع در این فایل هستند که به دلیل مشترک بودن در الگوریتم‌ها آن‌ها را به صورت جداگانه در این فایل قرار دادیم. در ادامه توضیحات این توابع آورده شده‌است:

- تابع **read_file**: این تابع نام فایلی که محیط اولیه‌مان در آن قرار دارد را می‌گیرد و محیط را به صورت آرایه (محیط با هزینه‌ها و محیط بدون هزینه‌ها) و هزینه‌ها را نیز به صورت آرایه، تعداد کره‌ها و مختصات اولیه‌ی ربات را برمی‌گرداند.
- تابع **:move_forward** این تابع محیط ، حرکت بعدی ربات و مختصات ربات را می‌گیرد و اگر حرکت بعدی ربات، باعث خروج ربات از محیط نشود، آن را انجام داده و مختصات جدید ربات را برمی‌گرداند.

- تابع **check_next_move:** این تابع بررسی می‌کند که آیا حرکت بعدی ربات مجاز می‌باشد یا خیر و در صورت مجاز بودن True و در غیر این صورت False بر می‌گرداند.
- تابع **get_all_permitted_movements:** این تابع با کمک تابع check_next_move تمام حرکاتی که برای ربات مجاز می‌باشد را برمی‌گرداند.
- تابع **update_environment:** این تابع محیط، مختصات ربات و حرکت بعدی ربات را می‌گیرد و محیط جدید را تشکیل داده و به همراه مختصات جدید ربات برمی‌گرداند.
- تابع **find_plates_coordinates:** این تابع مختصات تمام خانه‌های هدف را برمی‌گرداند.
- تابع **find_butters_coordinates**: این تابع مختصات تمام خانه‌هایی که کره در آن قرار دارند را برمی‌گرداند.
- تابع **generate_all_goal_environment:** این تابع تمامی حالاتی که می‌توانند حالت نهایی باشند را ایجاد کرده و برمی‌گرداند.
- همچنین چند تابع کمکی دیگر برای پرینت کردن محیط در هر مرحله در ترمینال و درآوردن لیست حرکات نهایی ربات و همچنین نوشتن نتایج در یک فایل می‌باشد.

الگوریتم **IDS**:

در این الگوریتم (با توجه به عمق که از یک شروع می‌شود و یکی یکی زیاد می‌شود تا به ماکسیمم عمق مشخص شده برسیم) فرزندان یک گره ایجاد می‌شوند و در عمق پیش می‌رویم و بررسی می‌کنیم که آیا به یکی از حالات هدف رسیده‌ایم یا خیر. در ادامه توضیحات توابع مربوطه آورده شده است:

- تابع **:create_child** این تابع یک گره را گرفته و فرزندان آن را می‌سازد.
- تابع **:dls** این تابع الگوریتم dls را اجرا می‌کند (جستجو را انجام می‌دهد و بعد هر مرحله عمق را یکی افزایش می‌دهد تا به ماکسیسم عمقی که در ورودی تابع‌اش آمده برسد) و اگر به یکی از حالات هدف برسد آن را برمی‌گرداند.
- تابع **:ids** این تابع، تابع dls را به صورت recursive اجرا می‌کند.
- تابع **:start_ids_algorithm** این تابع فایل تست کیس و ماکسیمم عمق را گرفته و الگوریتم IDS را اجرا می‌کند.

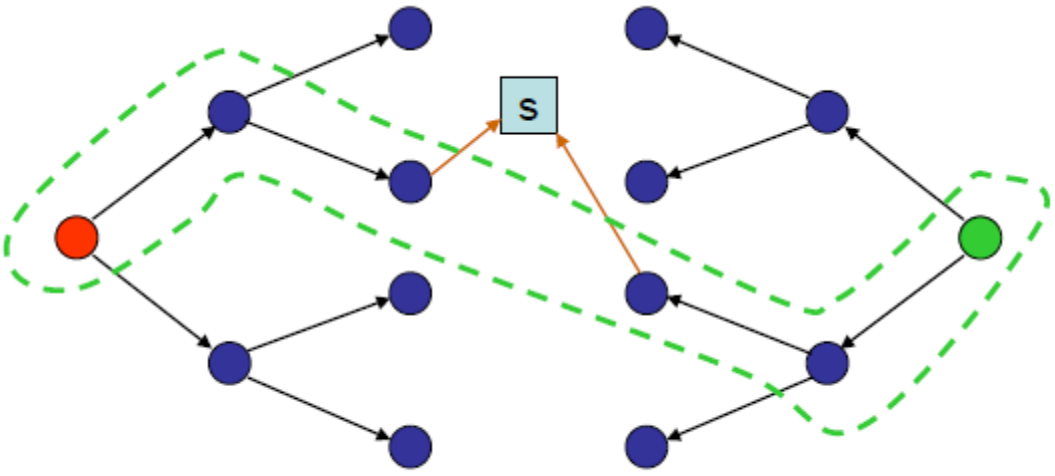
الگوریتم **BBFS**:

در فایل BBFS.py قرار دارد. در ابتدا تابع BBFS نام فایل را گرفته و آرایه‌های اولیه را با کمک تابع read_file تشکیل می‌دهد. سپس دو صف frontier را ایجاد می‌کنیم یکی برای اجرای BFS به صورت forward و دیگری برای backward. این دو صف را با حالت‌های اولیه‌شان مقداردهی اولیه می‌کنیم. در صف forward تنها حالت اولیه مسئله را داریم اما در حالت backward می‌توان چندین حالت پایان برای الگوریتم در نظر گرفت در نتیجه تمام این حالات را با کمک توابع generate_all_goal_environment و سپس create_final_nodes ساخته و به لیست اضافه می‌کنیم. حال در یک حلقه‌ی بی‌نهایت عنصر اول هر دو صف را گرفته و تمام حالات فرزندان موجود برای آن حالت را ایجاد می‌کنیم و به انتهای لیست اضافه می‌کنیم. حالت forward و backward به صورت جداگانه.
برای این کار از تابع bfs کمک می‌گیریم. این تابع یک حالت مسئه را به همراه صف frontier و این که آیا باید به صورت forward اجرا شود یا backward را گرفته و فرزندان آن حالت

را ایجاد می‌کند و در صورتی که فرزند تکراری نباشد و در صف موجود نباشد، آن را به صف اضافه می‌کند و در انتها صف جدید را برمی‌گرداند.

در حالتی که الگوریتم به صورت forward اجرا می‌شود از تابع update_environment استفاده می‌شود (که کارکرد آن توضیح داده شده‌است) اما در صورتی که به صورت backward اجرا شود تابع update_environment کمی تغییر می‌کند و به جای آن تابع update_environment_backward اجرا می‌شود که این تابع نیز حالت جدید محیط و مختصات جدید ربات را برمی‌گرداند. در این تابع حرکت ربات به صورت معکوس در نظر گرفته شده‌است و تابع reverse_movement برای چک کردن این است که آیا در راستای حرکت معکوس ربات کره هست یا خیر که در صورت وجود کره باید به صورت معکوس حرکت داده شود و محیط بروز شود.

بعد از هر بار اجرای این مراحل end_checker صف‌های forward و backward را گرفته و چک می‌کند که آیا گره مشترکی در این دو صف قرار دارد یا خیر. اگر گره مشترکی در این دو صف بود یعنی اینکه به جواب رسیده‌ایم و این گره مشترک و والد این گره از صف backward برگردانده می‌شوند. سپس تابع find_path گره مشترک و والد گره از طرف backward و محیط‌های هدف را گرفته و سپس والدها را دنبال می‌کند تا به حالات اولیه و هدف برسد و در نهایت این مسیر را به صورت آرایه‌ای از گره‌ها از حالت اولیه تا هدف برمی‌گرداند (مسیر سبز در شکل زیر).



در نهایت تابع BBFS این آرایه (و اینکه آیا مسئله جواب دارد یا خیر) را برمی‌گرداند و سپس این آرایه توسط تابع print_path در فایل AdditionalFunctions.py به صورت مرحله مرحله چاپ می‌شود. در صورتی که جمع عمق‌های آخرین گره‌های تولید شده در صف‌های forward و backward از تعداد خانه‌های محیط بیشتر شود به این معنا می‌باشد که تمام خانه‌های محیط را بررسی کرده‌ایم اما جوابی پیدا نشده پس پیغام there is no answer in this environment! در خروجی چاپ می‌شود.
تابع write_to_file در فایل AdditionalFunctions.py نیز برای نوشتن خروجی در فایل می‌باشد.

الگوریتم **A***:

توضیح تابع هیوریستیک: تابع هیوریستیک به این صورت می‌باشد که فاصله‌ی (منهتن) ربات تا هدف‌ها (ظرف‌ها) را محاسبه می‌کند و ترتیبی که ربات باید به هدف‌ها برود را مشخص می‌کند.(به نزدیک ترین خانه ی بشقاب هدف می رود و سپس به نزدیک ترین بشقاب بعدی می رود و ...)
توضیح توابع:

- تابع **calculate_manhattan_distance:** این تابع فاصله‌ی منهتن بین دو نقطه را برمی‌گرداند.
- تابع **find_closest_plate:** این تابع نزدیکترین ظرف به ربات را پیدا می‌کند.
- تابع **calculate_distance_point_to_all_plates:** این تابع از مختصات ربات به ترتیب به نزدیک‌ترین ظرف‌ها می‌رود.
- تابع **calculate_heuristic_environment:** این تابع برای هر خانه‌ی جدول مقدار تابع هیوریستیک آن را حساب می‌کند.
- تابع **sort_frontier_by_cost:** این تابع گره‌های در frontier را بر اساس هزینه‌ی f به صورت صعودی سورت می‌کند.

- تابع **calculate_cost_f_node:** این تابع هزینه‌ی f را برای یک خانه برمی‌گرداند.
- تابع **calculate_cost_g_node:** مقدار تابع هزینه‌ی g را برای یک خانه‌ی جدول برمی‌گرداند.
- تابع **is_new_node_in_frontier:** این تابع یک گره و لیست frontier را می‌گیرد و بررسی می‌کند که آیا این گره در لیست از قبل وجود داشته یا خیر.
- تابع **is_new_node_in_explored:** این تابع یک گره و لیست explored را می‌گیرد و بررسی می‌کند که آیا این گره در این لیست از قبل وجود داشته یا خیر که در صورت وجود دیگر آن گره را بررسی نمی‌کنیم.
- تابع **update_frontier_explored:** این تابع بررسی می‌کند که اگر گره از قبل در لیست frontier بود گره‌ای که هزینه‌ی کمتری دارد در لیست قرار گیرد.
- تابع **generate_node:** گره‌های فرزند یک گره را تشکیل می‌دهد.
- تابع **expanding_node:** فرزندان یک گره را با کمک تابع generate_node ایجاد می‌کند و آن گره را به لیست explored اضافه می‌کند.
- تابع **a_star_algorithm:** این تابع الگوریتم را اجرا می‌کند. این تابع از گره اولیه شروع می‌کند و فرزندانش را ایجاد می‌کند و بررسی می‌کند که آیا فرزندانش در یکی از حالات پایانی قرار دارند یا خیر و در صورت حالت پایانی نبودن به تولید فرزندان گره‌ها با اولویت هزینه‌ی گره‌ها می‌پردازد تا به یکی از حالات نهایی برسد. و اگر عمق گره از حداکثر عمقی که برای الگوریتم مشخص شده بیشتر شود الگوریتم را خاتمه می‌دهد.
- تابع **start_a_star_algorithm:** فایل تست و حداکثر عمق را می‌گیرد و الگوریتم A* را اجرا می‌کند.

مقایسه کلی:

در زیر خروجی سه الگوریتم برای تست کیس ۱ قابل مشاهده است:

برای الگوریتم **IDS** :

```
Which Algorithm do you want to use?
1) IDS
2) BBFS
3) A*
1
number of created nodes are: 103127
number of expanded nodes are: 103126
duration is :  3.7857465744018555
path costs :  11
depth of goal :  10
path is:  ['d', 'r', 'r', 'u', 'r', 'd', 'd', 'r', 'd', 'l']
**********************************************************
```

برای الگوریتم **BBFS** :

```
Which Algorithm do you want to use?
1) IDS
2) BBFS
3) A*
2
number of created nodes are: 392
number of expanded nodes are: 126
duration is :  0.024280071258544922
depth of goal :  10
```

برای الگوریتم **A*** :

```
Which Algorithm do you want to use?
1) IDS
2) BBFS
3) A*
3
number of created nodes are: 455
number of expanded nodes are: 147
duration is :  0.029184818267822266
path costs :  15
depth of goal :  10
path is:  ['r', 'd', 'r', 'u', 'r', 'd', 'd', 'r', 'd', 'l']
*****************************************************
```

Criterion	Breadth-First	Uniform-Cost	Depth-Frist	Depth-Limited	Iterative-Deepening	Bidirectional (If applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+C^*/\varepsilon})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+C^*/\varepsilon})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Superscript caveats are as follows:
[a] complete if b is finite
[b] complete if step costs ≥ ε for positive ε
[c] optimal if step costs are all identical
[d] if both directions use breadth-first search

زمان صرف شده : با توجه به جدول بالا و مقایسه ی الگوریتم ها می توان فهمید که درست پیاده سازی شده است چون زمان صرف شده در IDS از AStar و BBFS خیلی بیشتر است.

پیچیدگی زمانی: در جدول بالا ذکر شده است.

تعداد گره های تولید شده و گسترش داده شده: در IDS خیلی بیشتر از دو الگوریتم دیگر است زیرا برای هر cutoff میاید تمام گره‌های تا آن عمق را تولید و گسترش داده می شود

(برای فهم بهتر می توان مثال زیر را گفت: تفاوت جمع اعداد 1 تا 50 و خود عدد 50). در الگوریتم AStar جون هوشمندی به کار رفته است عموما باید تعداد نود های گسترش داده شده و تولید شده کم‌تر از BBFS باشد ولی این همیشه صدق نمی کند.

عمق راه حل: عمق جواب برای IDS و BBFS یکی هستند چون هزینه هر خانه را یک در نظر گرفته اند ولی برای AStar ممکن است بیشتر باشد زیرا شاید مسیری پیدا شود که طول آن بیشتر باشد ولی هزینه ی آن کم‌تر باشد(بهینه تر باشد).دقت شود در این تست کیس برای همه ی الگوریتم ها یکسان و برابر 10 است.(فرض خانه ی شروع ربات عمق صفر را دارد.)