به نام خالق زیبایی ها

پروژه اول

درس مبانی هوش مصنوعی و کاربرد ها

امیرفاضل کوزه گر کالجی

9931099

(0

searchProblems class:

getStartState:

حالت اولیه بازی را به ما میدهد

isGoalState:

چک میکند که آیا حالت فعلی همان حالت نهایی است یا نه

getSuccessors:

به ما استیت بعدی، اکشن برای رسیدن به استیت بعدی، و هزینه رسیدن به این استیت را می دهد

getCostOfActions:

هزینه اعمال یک سری از اکشن ها به که متد می دهیم را به ما می دهد.

Game.py classes:

Agent:

وضعیت حالت بازی را می گیرد و یک اکشن برای آن تولید میکند(بر اساس directions)

Directions:

جهت ها بر اساس هر سمتی که عامل قرار دارد در این کلاس تعیین شده است.

Configuration:

مختصات یک کارکتر و جهت آن را در بر دارد.

AgentState:

وضعیت یک عامل را در بر می گیرد(مختصات، جهت، سرعت،حالت ترسیده بودن)

Grid:

نقشه مربوط به بازی را در بر دارد که با استفاده از یک آرایه 2 بعدی آن را مدیریت می کند

Actions:

این کلاس شامل متدهایی است که توسط آن ها می توان تغییر جهت بازی را مدیریت کرد.

(1

|V| = # vertices

|E| = #edges

Time complexity = O(|V|+|E|)

Space complexity = O(|V|)

از آنجایی که این الگوریتم بر اساس عمق، پیمایش می کند یعنی تمام عمق ها را تا اخرین گره می پیماید تا به گره هدف برسد و از نظر زمانی هزینه زیادی می طلبد. اگر گره هدف در لایه های کم عمقی باشد، این الگوریتم مناسب تر است.

IDS algorithm:

```
IDS(root, goal, depthLimit){
   for depth = 0 to depthLimit
      if (DFS(root, goal, depth))
         return true
   return false
}

DFS(root, depth){
   if root == goal
      return true
   if depth == 0
      return false
   for each child in root.children
      if (DFS(child, goal, depth - 1))
         return true
   return false
}
```

این الگوریتم ترکیبی از dfs و bfs می باشد و در مسائلی که عمق جست و جو نامشخص یا بسیار بزرگ است، کاربرد دارد. بدین صورت عمل می کند که هر بار عمق مشخصی را در نظر می گیرد و جست و جو را در حد آن عمق انجام می دهد. دوباره در پیمایش بعدی یک لایه عمیق تر می شود و به همین ترتیب ادامه می دهد تا به گره هدف برسد.

اگر گره هدف در لایه های عمیق و پایینی قرار داشته باشد، dfs می تواند عملکرد بهتری نسبت به ids ارائه دهد.

(2

بله پازل بدون تغییر حل می شود

```
Press return for the next state...
After 5 moves: up
-------------
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
After 6 moves: left
-------------
| 3 | 1 | 2 |
-------------
|   | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
After 7 moves: up
-------------
|   | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
```

این مسئله را به طور گرافی مدل می کنیم هر گره در لبه گراف یک حالت از پازل است.
برای حل مسئله از الگوریتم bfs بهره میگیریم. گره ها را پیمایش میکنیم تا زمانی که به گره هدف برسیم.

Bbfs و bfs هردو الگوریتم هایی برای پیمایش گراف می باشند.

Bfs یک گراف را به طور سطح به سطح پیمایش میکند و در هر سطح گره ها را بررسی کرده و به دنبال گره هدف می گردد.

Bbfs نیز همانند bfs کار میکند با این تفاوت که در دو جهت است (bidirectional) این بدینگونه است که گراف را همزمان هم از گره آخر هم از گره اول پیمایش می کند.

BFS:

Time complexity: O(V+E)

Space complexity: O(V+E)

اگر گره هدف به ریشه نزدیک باشد یعنی در اعماق سطحی تری باشد، bfs بهتر است اما اگر گره هدف در اعماق پایینتر باشد، بهتر است که از dfs بهره ببریم.

(3

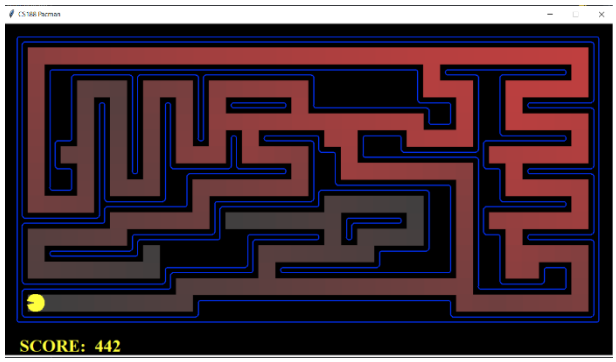
بله این امکان وجود دارد. برای رسیدن به bfs، باید تمامی هزینه ها را برابر در نظر بگیریم. در bfs همه هزینه ها یکسان در نظر گرفته می شود اما در UCS هزینه ها متفاوت است پس با یکی کردن هزینه ها، به bfs میرسیم. برای صورت گرفتن این کار نیز صرفا باید مقدار هزینه همه گره ها را یکسان تنظیم کنیم.

با اینحال نمی توان به dfs رسید زیرا لبه dfs پشته بوده ولی در bfs, ucs از صف استفاده می شود. اگرچه اگر تمام هزینه ها یکسان و برابر 0 در نظر گرفته شوند، میتوان dfs را تقلید کرد.

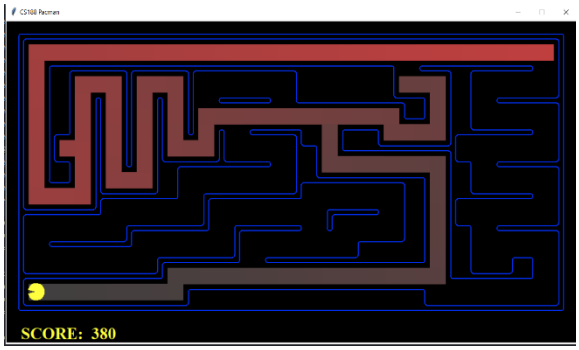
Ucs الگوریتمی ست که جواب بهینه را به ما می دهد و یکی از برتری های آن نسبت به bfs , dfs دانستن هزینه هایی راجع محیط است که باعث می شود انتخاب های بهتری بکند اما bfs , dfs نا اگاهانه کل لبه را پیمایش می کنند تا به گره هدف برسند.

با اینحال چون از صف اولویت برای لبه خود استفاده می کند، حجم زیادی از حافظه را به خود اختصاص می دهد.

(4

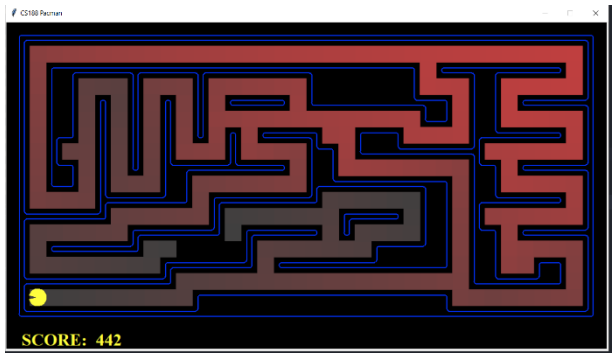


BFS



DFS

A*



UCS

الگوریتم هایی مثل bfs و dfs آگاهی ای از گره هدف ندارند و آنقدر پیمایش می کنند تا گره هدف را پیدا کنند. اما UCS و A* اطلاعاتی را راجع گره هدف دارند برای مثال می دانند انتخاب برخی مسیر ها هزینه زیادی برایشان دارد پس آن مسیر ها را انتخاب نمیکنند همچنین اطلاعات راجع گره هدف و هزینه ها در تابعی به نام تابع هزینه پیاده سازی می شود که دو مورد از آن ها تابع منهتن و اقلیدسی می باشد.

الگوریتم دایکسترا دارای چیزی به اسم تابع هیوریستیک نمی باشد و تمام گره ها را پیمایش می کند تا به جواب برسد. با اینحال در A* توابع هیوریستیک مختلفی می توان تعرف کرد که در انتخاب یک مسیر بهینه به الگوریتم کمک می کند.

5)

6)

ابتدا گوشه هایی که هنوز مورد بازدید قرار نگرفته را، در یک لیست ذخیره می کنیم. سپس از گره ای که در آنیم، هیوریستیک را شروع می کنیم. گوشه ای که کمترین فاصله را دارد پیدا میکنیم، فاصله اش را با متغیری به نام sum جمع می زنیم. و تا آن گوشه پیمایش میکنیم. وقتی گوشه پیمایش شد از لیست بازدید نشده ها حذف می شود و گوشه های دیگر مورد بررسی قرار میگیرند و این امر را آنقدر ادامه می دهیم تا تمام گوشه ها بازدید شوند.

7)

foodGrid را از وضعیت داده شده به هیوریستیک میگیریم. و به ازای تمام غذاهای داخل آن پیمایش می کنیم. اگر درون heuristicInfo مسئله، مختصات غذا وجود داشت، فاصله عامل با آن غذا را در لیستی به نام dists ذخیره می کنیم.

و اگر وجود نداشت از تابع mazeDistance بهره می گیریم تا کمترین فاصله عامل و غذا را داشته باشیم و باز همچنین در لیست dists ذخیره می کنیم.

سازگاری این هیوریستیک را می توان این گونه بیان کرد که، در هر بار گشتن برای یک غذا داریم کمترین فاصله را نسبت به یک غذا پیدا میکنیم.

*A آگاهانه است زیرا که هیوریستیک هایی دارد که اطلاعاتی درباره گره هدف به آن می دهند برای مثال مشخص می کنند برخی مسیر ها چگونه باعث افزایش هزینه و دورتر شدن از گره هدف می شوند که باعث می شود الگوریتم، هوشمندانه تر تصمیم بگیرد.

8)در حین اجرای قطعه کد خود، مشاهده شد که عامل مکرارا به یک نقطه نزدیک می شود اما از روی آن نمی گذرد تا اینکه تمامی نقاط بلعیده می شوند و آن نقطه به عنوان آخرین نقطه بلعیده می شود اما هدف مسئله که خوردن تمامی نقاط است، به ثمر می رسد دقیقا همانگونه که هیوریستیک را برنامه ریزی کرده بودیم