# پروژه اول درس مبانی هوش  مصنوعی

**(ربات، کره و میز غذا)**


**اعضای گروه:**

**امید ماهیار / شماره: ۹۷۳۱۱۰۰**

**پویان حسابی/ شماره: ۹۷۳۱۱۲۲**


تاریخ: ۲۲ اردیبهشت ۱۴۰۰

# نحوه مدلسازی مسئله برای جستجو:

این پروژه این درس برای یادگیری و استفاده از الگوریتم های جستجوی گراف و درخت می باشد. مسئله به این شکل مدل سازی شده که هر استیت یک نود حساب می شود، هر نود شامل لوکیشن ربات، لوکیشن کره ها، آخرین حرکت، والد نود، هزینه و عمق گراف می باشد.

هرگاه نودی ساخته می شود حداقل یکی از این اطلاعات تغییر میکند(جلوتر توضیح داده می شود) و در فرانتیر این نود اضافه می شود و در توابع بعدی از آن استفاده می شود.

به طور کلی فرض های صورت پروژه همگی رعایت شده اند و ربات شروع به حرکت میکند و کره را به یک هدف متناظر تحویل می دهد. کشیدن، هل دادن و حمل کردن(هر دو در یک خانه باشند) همانطور که گفته شد مجاز نیست.

برای جستجو در هر الگوریتم بر اساس ویژگی هایی که درون هر نود ذخیره شده است، کار های مختلف از آن الگوریتم مربوطه را انجام می دهد. **به بیان ساده هر نود نشانگر صفحه ی میز در یک صحنه(زمان) است**.

در این پروژه ۳ الگوریتم IDS, bidirectional, A* پیاده سازی شده است.

# تابع شهودی انتخاب شده و بررسی قابل قبول بودن آن:

تابع شهودی فاصله هر نقطه تا هدف ها می باشد که چون cost همیشه یک یا بزرگتر از یک است تابع شهودی ما همیشه دارای مقداری کمتر از مقدار واقعی خواهد بود.

# توضیح کلی توابع و کلاس های تعریف شده از کد:

هر الگوریتم در یک فایل جداگانه ی پایتون نوشته شده است.

class Node:

این کلاس برای پیاده سازی هر حالت استفاده شده که شامل لوکیشن ربات، لوکیشن کره ها، آخرین حرکت، والد نود، هزینه و عمق گراف می باشد.

### class NodeForGoal:

این کلاس برای پیاده سازی حالت هایی که حداقل یکی از هدف ها در آن مورد تغییر می گیرد می باشد، در الگوریتم دوم مورد استفاده قرار گرفته است. این کلاس شامل لیست اهداف، والد، آخرین حرکت، هزینه و عمق می باشد.

### read_on_file, write_on_file:

برای ورودی و خروجی گرفتن است.

### get_path:

برای برگرداندن مسیر و نوشتن آن در خروجی، به این شکل عمل می کند که آخرین حرکت والد های نود مربوطه را بررسی میکند.

### generate_children:

فرزندان نود ورودی را تولید کرده و آنرا به عنوان خروجی می دهد. ابتدا همسایه های ربات را میسازد و چک میکند اگر آن همسایه ها در نقاط مرزی یا بلوک ها نبودند به لیستی اضافه میکند. به این شکل است که با تغییر کردن مکان ربات چک میکند که کره همسایه آن هست یا نه اگر بود اقدامات مبنی بر حرکت آن را انجام می دهد و حرکت آخر را در نود مربوطه ذخیره میکند.

### generate_goal_children:

یک نود و اندیس ورودی میگیرد و فرزندان آن نود را خروجی میدهد. اندیس ورودی همان شماره هدف مورد نظر است که فرزندان آن نود زمانی که آن هدف مربوطه تغییر میکند. تفاوت آن با تابع قبل این است که از سمت هدف شروع به تغییر میکند و در الگوریتم دوم فقط مورد استفاده قرار میگیرد.

### DFS:

پیاده سازی الگوریتم dfs است که در ورودی maxdepth را میگیرد تا در تابع ids از آن استفاده کند و در عمق های مربوطه تغییرات لازم را بدهد. همانطور که میدانیم ساختار DFS بر اساس LIFO می باشد و به این شکل پیاده سازی شده است که در کد مشخص است. فقط در الگوریتم اول استفاده میشود.

### IDS:

تابعی به بازگشتی که یک نود، ریشه، را میگیرد و جستجو را انجام میدهد و dfs را در عمق های مختلف انجام میدهد. فقط در الگوریتم اول استفاده می شود.

BID:

این تابع یک اندیس ورودی میگیرد که برای هدف مربوطه جستجوی دو طرفه را انجام دهد. به این شکل است که ابتدا یک مرحله از BFS را برای نود ریشه که تغییر ربات است و سپس یک مرحله از BFS برای هدف مربوطه را انجام میدهد و در فرانتیر های جداگانه میریزد. در صورتی که به همدیگر رسیدند این الگوریتم متوقف شده و مسیر و موارد دیگر را در فایل ذخیره میکند. لازم به ذکر است که ساختار BFS براساس FIFO میباشد و به این شکل پیاده سازی شده است. فقط در الگوریتم اول استفاده می شود.

check_goal:

این تابع برای بررسی خاتمه الگوریتم ها میباشد، البته در هر فایل کمی با هم تفاوت دارد، نسبت به الگوریتم استفاده شده تغییر کرده است. ورودی یک نود میگیرد و چک میکند که آیا حداقل یکی از کره ها در هدف قرار دارد یا خیر، در صورتی که وجود داشته مسیر را فراخوانی میکند و نود را به شکل خروجی میدهد.

find_min:

یک لیست و یک شماره هدف میگیرد و مقدار خانه اول لیست به علاوه هیوریستیک هدف مورد نظر و هزینه آن میکند. اگر از خانه بعدی مقدار آن کمتر بود(مطابق الگوریتم A* در درس که گفته شده) آن نود را ذخیره کرد و return میکند.

fill_heuristic:

همانطور که در قسمت تابع شهودی گفته شد، این متد، تابع هیوریستیک را میسازد. قدر مطلق فاصله هر خانه تا هدف مورد نظر را حساب میکند که در حلقه ای تو در تو این مقدار را در لیست اصلی میریزد.

A_star:

از دوباره گویی توضیحاتی که در قسمت سوم نوشته شده اجتناب کردیم. ولی به طور کلی این تابع به صورت بازگشتی الگوریتم A* را اجرا میکند. تا وقتی که روت جدید موجود باشه فرزند جدید از عنصر کمینه میسازد و در حلقه ای تو در تو در لیست کاوش شده ها میگردد و اگر فرزند مورد نظر در لیست کاوش شده باشد آن را حذف میکند تا در ادامه کار خللی پیش نیاید و در آخر آن فرزند را در فرانتیر اضافه میکند(کد واضح است).

# مقایسه روشهای پیاده سازی شده در موارد زیر

**زمان صرف شده:**

زمان صرف شده به طور معمول و کلی از الگوریتم اول به آخر نزولی است. در IDS در عمق های مختلف چون چند بار گراف ساخته می شود و نود های تکراری بار ها کاوش میشوند زمان زیادی صرف میشود. در bidirectional زمان کمتری صرف میشود چون از دو طرف بررسی میشود ولی همچنان در حالت هایی بسیار بدتر از A* برخورد میکند چون براساس تابع هیوریستیک کمترین هزینه در این الگوریتم بدست می آید.

**پیچیدگی زمانی:**

IDS: $O(b\hat{}d)$

bidirection: $O(b\hat{}d/2)$

A*: برای اغلب مسائل، تعداد نود ها در داخل کانتور هدف (یعنی گرههایی با **f(n)≤C***) برحسب طول راه حل نمایی است. شرط نمایی نبودن تعداد نود ها: خطای $\left|h(n)-h^*(n)\right| \le O(\log h^*(n))$ تابع هیوریستیک، سریعتر از لگاریتم هزینه واقعی مسیر رشد نکند.

**تعداد گره های تولید شده:**

در IDS به طور معمول از همگی بیشتر از است چون از اول در هر عمق همه ی آنها را تولید میکند، در دوطرفه وضعیت بهتری دارد گره های خیلی کمتر از آن تولید میشود چون در حافظه نگهداری میشود. در A* از دوطرفه بیشتر تولید میکند چون تمام نود های اطراف آن تولید میشود تا کمترین آنها پیدا شود.

**تعداد گره های گسترش داده شده:**

به ترتیب IDS, Bidirectional, A* زیاد می شود. یعنی در A* با اینکه نسبت به دو طرفه نود های بیشتری تولید میشود ولی سعی بر این است که با تابع هیوریستیک تعداد گره های گسترش یافتع مینیمم شود. دو طرفه همچنان بهتر از IDS میباشد چون حافظه دارد و تکراری ها دوباره گسترش پیدا نمیکنند.

**عمق راه حل:**

بر اساس خروجی هایی که داریم به ترتیب عمق A* و bidirectional و IDS می باشد.