دانشگاه علم و صنعت

دانشکده مهندسی کامپیوتر

هوش مصنوعی و سیستم های خبره

تمرین عملی ۱

گردآورنده:

پرنیان شاکریان - 99400064

استاد:

دکتر آرش عبدی

سال تحصیلی: خرداد ۱۴۰۲

بخش اول: پیاده سازی الگوریتم UCS و A*

در این بخش از ما خواسته شده که الگوریتم‌های UCS و A* را پیاده سازی کرده و با گراف داده شده آن را تست کنیم. در هر مرحله برای هر دو الگوریتم داده شده nodeهای درون Frontier و در پایان مسیر نهایی را به عنوان خروجی چاپ میکنیم. همچنین در هر مرحله از خروجی خود نشان میدهیم که کدام node در Frontier میشود. همچنین برای هر node در Frontier مقدار g+h (در A*) و مقدار g (در UCS) را مشخص و در خروجی چاپ خواهیم کرد.

تحلیل تمرین و مفاهیم پایه

Uniform Cost Search (UCS):

UCS یک الگوریتم جستجو است که گراف را با در نظر گرفتن هزینه مسیر (g) از node شروع به هر node بررسی میکند. UCS دامنه جست و جو خود در node ها را گسترش داده و همیشه node، با کمترین هزینه از node شروع را انتخاب میکند. (از یک صف اولویت برای حفظ frontier استفاده می کند، به گونه‌ای که nodeها با هزینه کمتر اولویت بیشتری دارند.) الگوریتم زمانی خاتمه می یابد که به node هدف برسد یا زمانی که frontier خالی باشد. UCS تضمین میکند که بهینه‌ترین مسیر را از نظر هزینه کل پیدا کند.

A* (A Star):

A* یک الگوریتم جستجو است که هزینه واقعی (g) را با هزینه باقی‌مانده تخمینی (h) از هر node به node هدف بررسی میکند. این الگوریتم اولویت nodeها را با در نظر گرفتن و محاسبه مجموع g و h (g+h) پیدا میکند. (از یک صف اولویت برای حفظ frontier استفاده می کند، به گونه‌ای که node ها با مقادیر g+h کمتر اولویت بیشتری دارند.) این الگوریتم زمانی خاتمه می یابد که به node هدف برسد یا زمانی که frontier خالی باشد. اگر تابع اکتشافی قابل قبول و سازگار باشد، A* تضمین کرده که مسیر بهینه را پیدا میکند.

شرح کد

heuristic (node, goal) : تابع اول ما مقادیر h را برای هر node در گراف تعریف میکند تا در ادامه برای محاسبه الگوریتم A* از آن استفاده کند. درون تابع، دیکشنری به نام heuristic_values به منظور ذخیره مقادیر h برای هر node استفاده میشود. تابع heuristic یک node و goal را به عنوان ورودی گرفته و مقدار h را برای آن node برمیگرداند.

ucs (graph, start, goal): تابع دوم را برای محاسبه الگوریتم Uniform Cost Search پیاده‌سازی میکنیم. تابع، گراف، node شروع و node هدف را به عنوان پارامترهای ورودی گرفته سپس یک صف اولویتی را با node شروع و هزینه آن را به عنوان عنصر اولیه ۰ مقداردهی میکند. الگوریتم، هزینه رسیدن به هر node را با استفاده از cost_sofar پیگیری می کند. (cost_sofar با هزینه ۰ برای node شروع مقداردهی می شود) همچنین از dictionary برای ذخیره والد هر node بازدید شده استفاده میکند. (dictionary والد با None برای node شروع راه اندازی می شود) سپس تابع وارد یک حلقه می شود و تا خالی شدن صف ادامه می یابد. در هر تکرار، node های موجود در frontier و

هزینه آنها را چاپ میکند. (لیست frontier_nodes شامل node های frontier و لیست frontier_costs شامل هزینه های مربوطه از cost_so_far است) node که کمترین هزینه را دارد از صف خارج می شود. سپس همسایگان node فعلی را با تکرار بر روی آیتم‌های گراف، بررسی کرده و هزینه جدید برای رسیدن به هر همسایه را محاسبه میکند. اگر همسایه ای از قبل در cost_so_far نبوده یا هزینه جدید کمتر از هزینه موجود باشد، cost_so_far را به روز کرده و همسایه با هزینه جدیدش را به صف اضافه می کند. همچنین dictionary والد node فعلی را به عنوان والد همسایه به روز میکند. اگر به node هدف رسید، تابع با استفاده از dictionary والد، مسیر را از ابتدا تا هدف، با دنبال کردن مکرر nodeهای والد می‌سازد و در آخر مسیر چاپ می شود. اگر node هدف پیدا نشد، پیام Path not found را چاپ میکند.

A* (graph, start, goal): تابع سوم را برای محاسبه الگوریتم A* پیاده‌سازی میکنیم. تابع، گراف، node شروع و node هدف را به عنوان پارامترهای ورودی گرفته سپس یک صف اولویتی را با node شروع و f-score آن به عنوان عنصر اولیه مقداردهی میکند. f-score از جمع g_score (هزینه واقعی) و heuristic value محاسبه می‌شود. الگوریتم، هزینه رسیدن به هر node را با استفاده از g_score بررسی می کند. (node g_score شروع ۰ مقداردهی اولیه میشود) همچنین از یک dictionary والد برای ذخیره والد هر node بازدید شده، استفاده میشود. سپس تابع وارد یک حلقه شده و تا خالی شدن صف ادامه می یابد. در هر تکرار، node های موجود در frontier را به همراه f و g-score آنها چاپ میکند. لیست frontier_nodes شامل nodeهای frontier و لیست‌های frontier_g_scores و frontier_f_scores حاوی امتیازات g و f مربوطه است. node که کمترین مقدار f-score را دارد از صف بیرون می آید. همسایگان node فعلی را با تکرار بر روی آیتم‌های گراف[current_node] که همسایه‌ها و هزینه‌های لبه آنها را نشان می‌دهد، بررسی می‌کند. در ادامه همسایه‌های گره فعلی را با تکرار روی نمودار [current_node] که نشان‌دهنده همسایه‌ها و هزینه‌های frontier آن‌ها است، کاوش می‌کند. اگر یکی از همسایه‌ها قبلاً درdictionary g_score نبوده یا نمره g آن کمتر از g-score موجود باشد، g_score dictionary را به روز می کند. با جمع g و h امتیاز f را برای همسایه محاسبه کرده و f-score همسایه را به صف اضافه می کند. اگر به node هدف رسید، تابع با استفاده از dictionary والد، مسیر را از ابتدا تا هدف، با دنبال کردن مکرر nodeهای والد می‌سازد و در آخر مسیر چاپ می شود. اگر node هدف پیدا نشد، پیام Path not found را چاپ میکند.

بخش دوم: پیاده سازی الگوریتم CSP

در این بخش نیاز است که با استفاده از روش‌های جستجوی افزایشی و جستجوی کامل مسئله ۸ وزیر را پیاده سازی کنید و گزارش مختصری در خصوص پیاده‌سازی انجام شده بدهید.

الف) با استفاده از جستجوی افزایشی و با ترکیب روش MRV+FC مسئله ۸ وزیر را پیاده سازی کنید. حداقل نتایج میانی ۱۰ گام از حل مسئله را به طور دقیق و کامل نمایش دهید.

در ابتدای کد تابع is_valid بررسی می کند که آیا قرار دادن یک وزیر در یک موقعیت خاص (ردیف، ستون) روی بورد معتبر است یا خیر. روی ردیف های بالای ردیف فعلی تکرار می شود تا تداخل با وزیر های قبلی را بررسی کند. اگر تداخلی وجود داشت، False و در غیر این صورت، True را برمی گرداند.

تابع solve_n_queens یک تابع بازگشتی است که سعی در حل مسئله N-Queens دارد. سه پارامتر را در بوردارد:

۱. بورد (آرایه ای که وضعیت فعلی بورد را نشان می دهد)
۲. ردیف فعلی در نظر گرفته شده
۳. اندازه کل بورد n

اگر ردیف فعلی برابر با n باشد، به این معنی است که تمام وزیرها با موفقیت قرار گرفته اند، و برای نشان دادن یافتن راه حل، True را برمی گرداند. در غیر این صورت، روی ستون‌های ردیف فعلی تکرار می‌شود و بررسی می‌کند که آیا قرار دادن یک وزیر در هر موقعیت معتبر است یا خیر. اگر معتبر باشد، وزیر را در آن موقعیت قرار داده و به صورت بازگشتی solve_n_queens را برای ردیف بعدی (ردیف + ۱) فراخوانی می کند. اگر فراخوانی بازگشتی True را برگرداند، به این معنی است که راه حلی پیدا شده است. حال اگر فراخوانی بازگشتی False را برگرداند، به این معنی است که قرارگیری فعلی وزیر ها منجر به تداخل در پایین خط می شود، بنابراین موقعیت فعلی را بازنشانی می کند و به ستون بعدی در حلقه ادامه می دهد. اگر جای معتبری در هیچ ستونی یافت نشد، False را برمی‌گرداند تا به ردیف قبلی برگردد و موقعیت‌های مختلفی را برای وزیر ردیف قبلی امتحان کند.

تابع solve_8_queens پارامترهای اولیه را برای حل مسئله ۸ وزیر تنظیم میکند. یک بورد خالی را که با آرایه‌ای به اندازه ۸ نشان داده شده مقداردهی اولیه می‌کند تا جایی که هر عنصر به ۱- مقداردهی شود. (همچنین بورد خالی اولیه را چاپ می کند.)

تابع Sol_n_queens را با بورد اولیه فراخوانی می‌کند، از ردیف اول (ردیف = ۰) شروع می‌شود و اندازه بورد $n = 8$ را ارسال می‌کند. اگر True را برگرداند، به این معنی است که یک راه حل پیدا شده است، بنابراین آخرین وضعیت بورد را با وزیر هایی که با "Q" نشان داده شده اند و سلول های خالی با "-" چاپ می کند. اگر False را برگرداند، به این معنی است که هیچ راه حلی پیدا نشده، پس پیامی را چاپ می کند که نشان می دهد هیچ راه حلی وجود ندارد.

ب) بار دیگر با استفاده از جستجوی کامل (جستجوی محلی) مسئله ۸ وزیر را پیاده سازی کنید و ۱۰ گام را نمایش دهید. برای این کار ابتدا وزیر ها را به صورت تصادفی در صفحه قرار دهید (با رعایت این شرط که در هر ستون فقط یک وزیر باشد) سپس جستجو را آغاز کنید. در آزمایش خود حداقل یک بار جست و جوی موفق و یک بار شکست را ازمایش کنید و در گزارش بیاورید.

در ابتدای تابع evaluate تعداد تداخل ها بین وزیرها را در بورد محاسبه میکند. (تداخل ها را به صورت افقی، عمودی و مورب بررسی می کند.) اگر دو وزیر در یک ردیف، ستون یا مورب مشترک باشند، تداخل‌ها افزایش می‌یابد.

تابع gene_initial_board یک پیکربندی اولیه بورد تصادفی ایجاد می کند. (اعداد را از ۰ تا N-1 به طور تصادفی به هم می‌ریزد تا موقعیت‌های وزیرها را روی بورد نشان دهد.)

تابع local_search الگوریتم جستجوی محلی را انجام می دهد که یک پیکربندی اولیه بورد را به عنوان ورودی گرفته و به طور مکرر راه حل های همسایه را برای بهبود پیکربندی فعلی بررسی می کند.

تابع display_board برای چاپ پیکربندی درون بورد استفاده میشود. نمایی تصویری از بورد را در اختیارمان گذاشته که در آن وزیر ها با 'Q' و مربع های خالی را با '-' نشان میدهد.

بخش اصلی کد، الگوریتم جستجوی محلی را در دو سناریو آزمایش میکند:

۱. جستجوی موفقیت آمیز: یک پیکربندی اولیه بورد تصادفی ایجاد می کند و local_search را برای یافتن راه حل فراخوانی میکند. این الگوریتم به طور مکرر راه حل های همسایه را بررسی میکند، پیکربندی بورد هر مرحله را چاپ می کند و زمانی که راه حلی بدون تضاد پیدا کند متوقف میشود.

۲. جستجوی ناموفق: از یک پیکربندی بورد از پیش تعریف شده استفاده می کند که در آن وزیر ها در حالت غیر حل قرار می گیرند (همه وزیر ها در یک ردیف). تابع local_search دوباره فراخوانی میشود، اما از آنجایی که راه حل معتبری وجود ندارد، پیکربندی بدون تداخل پیدا نمیکند.

در آخر نتایج هر دو حالت را چاپ کرده و در jupyter notebook نمایش میدهیم.