این پروژه، پیاده سازی الگوریتم جستجوی A* است که برای حل پازل ۸ کاشی استفاده می شود. مسئله 8 پازلی یک پازل کشویی است که شامل یک قاب از کاشی های مربعی شماره گذاری شده به ترتیب تصادفی است که یک کاشی از دست رفته است. هدف این است که با انجام حرکات کشویی که از فضای خالی استفاده می کند کاشی ها را به ترتیب قرار دهید

در این پروژه ۳ روش متفاوت به کاربرده شده است که در قالب ۳ فایل به نام های hamming.py, manhattan.py و manhattan_set.py قرار داده شده اند. تفاوت دو فایل اول استفاده از توابع متفاوت برای محاسبه‌ی فاصله تا هدف و استفاده از دو ساختمان داده متفاوت (لیست و heap) هستند. فایل سوم اما از فاصله‌ی منهتن و ساختار داده‌ی جدول هش استفاده میکند که سریع ترین روش محسوب میشود.

توابع

manhattan.py:

get_zero (puzzle)

این تابع موقعیت کاشی صفر را در پازل برمی گرداند. یک پازل را به عنوان آرگومان ورودی می گیرد و فهرستی از دو عدد صحیح، شاخص سطر و ستون کاشی صفر را برمی گرداند.

get_neighbors (puzzle)

این تابع لیستی از همه همسایه های ممکن برای کاشی صفر را برمی گرداند. یک پازل را به عنوان آرگومان ورودی می گیرد و فهرستی از لیست ها را برمی گرداند که در آن هر لیست مختصات یک کاشی همسایه را نشان می دهد.

move(puzzle, neighbor)

این تابع پس از انتقال کاشی صفر به کاشی همسایه، یک پازل جدید برمی گرداند. یک پازل و یک کاشی همسایه را به عنوان آرگومان های ورودی می گیرد و پس از تعویض کاشی صفر و کاشی همسایه، یک پازل جدید برمی گرداند.

manhattan_distance (puzzle)

این تابع فاصله منهتن یک پازل را محاسبه می کند. فاصله منهتن به عنوان مجموع فاصله کاشی ها از موقعیت های هدفشان تعریف می شود. یک پازل را به عنوان آرگومان ورودی می گیرد و یک عدد صحیح را به عنوان فاصله منهتن برمی گرداند.

a_star_search (puzzle، goal)

این تابع الگوریتم جستجوی A* را انجام می دهد. یک پازل و حالت هدف را به عنوان آرگومان های ورودی می گیرد و لیستی از پازل ها را برمی گرداند که مراحلی از حالت اولیه تا حالت هدف هستند.

این تابع ابتدا یک صف اولویت را با استفاده از کتابخانه heapq برای ذخیره حالت هایی که نیاز به کاوش دارند، مقداردهی می کند.

سپس، ایالتی را با کمترین فاصله منهتن از صف نمایش می دهد و بررسی می کند که آیا وضعیت هدف است یا خیر.

اگر حالت هدف باشد، تابع لیست مراحل را از حالت اولیه به حالت هدف برمی گرداند.

اگر حالت هدف نباشد، تابع همسایه های ایالت را بررسی می کند و حالت های جدید را به صف اضافه می کند.

تابع فرآیند را تا زمانی که صف خالی شود تکرار می کند.

print_puzzle (puzzle، goal)

این تابع مراحل را از حالت اولیه تا حالت هدف چاپ می کند. یک پازل و حالت هدف را به عنوان آرگومان های ورودی می گیرد و پازل ها را در مراحل چاپ می کند.

hamming.py: (اکثرتوابع شبیه فایل قبل میباشند، جز موارد پایین)

hamming Distance (puzzle، goal)

این تابع فاصله همینگ را محاسبه می کند که تعداد کاشی های پازل است که در موقعیت مناسبی نسبت به پازل هدف قرار ندارند.

a_star_search (puzzle، goal)

این تابع الگوریتم جستجوی A* را برای یافتن مراحل مورد نیاز برای رسیدن به پازل هدف از پازل اولیه پیاده سازی می کند. frontier را بر اساس فاصله همینگ هر پازل مرتب می کند و فهرستی از پازل ها را نشان می دهد که مراحل انجام شده را نشان می دهد.

manhattan.py: (شبیه دو تابع قبل، جز موارد زیر)

manhattan_distance_hash_table(puzzle, goal):

محاسبه‌ی فاصله‌ی هممینگ این بار با ساخت یک جدول هش برای پازل هدف

hash_puzzle(puzzle):

تبدیل پازل به یک رشته و سپس هش کردن آن

a_star_search_hash_table(puzzle, goal):

این تابع الگوریتم جستجوی A* را برای یافتن مراحل مورد نیاز برای رسیدن به پازل هدف از پازل اولیه پیاده سازی می کند با این تفاوت که داده ها را توسط ساختار داده‌ی جدول هش ذخیره شده اند که سرعت را افزایش میدهد.

مقایسه‌ی روش ها

دو تابع جستجوی A* برای حل پازل استفاده می شود. کد اول یک الگوریتم جستجوی پایه A* است که از فاصله همینگ به عنوان یک heuristic برای مرتب کردن frontier استفاده می کند. کد دوم از همان الگوریتم جستجوی A* استفاده می کند، اما شامل مکانیزم مرتب سازی است که frontier را بر اساس فاصله همینگ مرتب می کند.

دو کد منهتن و همینگ ارائه شده از روش‌های مختلفی برای ذخیره ساختارهای داده‌های پازل استفاده می‌کنند که در طول جستجوی A* کاوش می‌شوند. کد اول از یک ساختار داده پشته برای ذخیره پازل استفاده می کند، در حالی که کد دوم از یک لیست استفاده می کند.

Heap ها درخت های دودویی هستند که دارای خاصیت خاصی هستند، یعنی گره والد باید کمتر یا مساوی با گره های فرزند باشد. این ویژگی به هیپ اجازه می دهد تا به طور موثر مرتب شود و حداقل عنصر را می توان در زمان ثابت پیدا کرد. در الگوریتم جستجوی A*، صف اولویت به صورت پشته پیاده‌سازی می‌شود، زیرا به روشی کارآمد برای یافتن پازل با کمترین مقدار heuristic نیاز دارد.

در مقابل، کد hamming.py از یک لیست برای ذخیره پازل ها استفاده می کند. لیست یک آرایه پویا است، به این معنی که می تواند به صورت پویا رشد یا کوچک شود. عناصر لیست در حافظه پیوسته

ذخیره می شوند، بنابراین دسترسی به هر عنصر لیست به زمان O(1) نیاز دارد. با این حال، هنگام درج عناصر در یک لیست، پیچیدگی زمانی O(n) است، که در آن n تعداد عناصر موجود در لیست است.

از نظر کارایی، استفاده از یک پشته به طور کلی سریعتر از استفاده از یک لیست خواهد بود، زیرا پیچیدگی زمانی یافتن حداقل عنصر در یک پشته O(1) است، در حالی که در یک لیست O(n) است. با این حال، عملکرد واقعی به عوامل بسیاری مانند اندازه لیست و پیچیدگی عملیات مرتب سازی بستگی دارد.

استفاده از ساختار داده صف اولویت (هپ) برای ذخیره frontier در الگوریتم جستجوی A* می تواند سرعت را به طور چشمگیری در مقایسه با استفاده از یک لیست بهبود بخشد. دلیل این امر این است که ساختار داده پشته راهی کارآمد برای دسترسی به آیتم با بالاترین اولویت (کمترین هزینه تخمینی) فراهم می‌کند و تعیین گره بعدی در الگوریتم جستجوی A* را آسان‌تر می‌کند. این می تواند منجر به کاهش قابل توجه تعداد گره هایی شود که نیاز به گسترش دارند و در نتیجه می تواند سرعت جستجو را بهبود بخشد.

اما ساختارهای داده ای که می تواند سرعت را به طور چشمگیری بهبود بخشد، جدول هش یا Hash Table است. جداول هش به طور متوسط با نگاشت مقادیر به کلیدهای منحصر به فرد و ذخیره آنها در یک آرایه، امکان دسترسی، درج و حذف عملیات با زمان ثابت را فراهم می کند. این می تواند هنگام جستجوی یک مقدار خاص در یک مجموعه داده بزرگ مفید باشد، زیرا نیاز به جستجوی خطی را از بین می برد.