
بسم الله الرحمن الرحیم

دانشگاه تبریز
دانشکده مهندسی برق و کامپیوتر

گزارش پروژه کارشناسی مهندسی برق-کنترل

پیاده سازی جستجوی مبتنی بر تعارض برای مسیریابی بهینه چند عاملی

سعید بنی نصرت

استاد ارجمند:

دکتر برادران نیا

زمستان ۱۴۰۲

چکیده

این تحقیق به بررسی مسیریابی چند عاملی (MAPF[1]) با استفاده از جستجوی مبتنی بر تعارض (CBS[2]) به عنوان یک روش حل مسئله می پردازد. هدف اصلی طراحی یک الگوریتم کارآمد است که حرکت چندین عامل را در یک پیچ و خم تعریف شده تسهیل می‌کند و در عین حال دو قانون اساسی را رعایت می‌کند: اجتناب از برخورد بین عوامل و دور زدن مسیرهای مسدود شده. راه حل پیشنهادی از الگوریتم A* برای هدایت عوامل از نقاط شروع مربوطه به نقاط پایانی تعیین شده استفاده می کند. بر خلاف روش‌های موجود، که ممکن است از ناکارآمدی یا عدم سازگاری رنج ببرند، رویکرد CBS یک روش سیستماتیک برای حل تعارض‌های بین مسیرهای عوامل ارائه می‌دهد و از مسیریابی روان‌تر و بهینه‌تر اطمینان می‌دهد. این تحقیق از طریق آزمایش و تحلیل گسترده، کارایی روش پیشنهادی را در افزایش کارایی مسیریابی چند عاملی در مقایسه با تکنیک‌های مرسوم نشان می‌دهد. به طور کلی، نتایج نوآوری و اثربخشی الگوریتم MAPF مبتنی بر CBS را در پرداختن به پیچیدگی‌های ذاتی در سناریوهای مسیریابی چند عاملی برجسته می‌کند.

کلید واژه: CBS, MAPF, A*

[1] Multi-Agent Path Finding
[2] Conflict Based Search

فهرست مطالب

عنوان **صفحه**

فصل ۱– مقدمه

۱-۱- پیشگفتار:

مسیریابی چند عاملی (MAPF) یک مشکل اساسی در رباتیک و هوش مصنوعی است، که در آن چندین عامل باید از موقعیت های شروع مربوطه خود به موقعیت های هدف فردی بدون برخورد با یکدیگر یا موانع ساکن در محیط حرکت کنند.

هدف پروژه حل مسئله MAPF با استفاده از الگوریتم جستجوی مبتنی بر تعارض (CBS) است که یک رویکرد بسیار گسترده برای راه‌حل‌های بهینه MAPF میباشد.

در این پروژه، یک محیط هزارتو[۱] مانند تعریف شده است که به صورت یک صفحه شطرنجی[۲] نمایش داده می شود، که در آن گره‌ها[۳] با مکان های قابل عبور مطابقت دارند، و یال ها حرکات معتبر بین گره های مجاور را نشان میدهند. محیط همچنین شامل موانع ایستا[۴] است که به صورت گره ها یا لبه های مسدود شده نمایش داده می شوند که عوامل نمی توانند آنها را اشغال کنند یا از آنها عبور کنند.

عوامل[۵] به عنوان موجودیت‌های منفرد مدل‌سازی می‌شوند که هرکدام دارای یک شروع و موقعیت هدف منحصر به فرد در پیچ و خم هستند. هدف این است که مسیرهای بدون برخورد[۶] را برای همه عوامل پیدا کنیم، تا اطمینان حاصل شود که هیچ دو عاملی به طور همزمان مکان یکسانی را اشغال نمی کنند، و ماموران سعی نمی کنند از مناطق مسدود شده عبور کنند.

برای رسیدن به این هدف، الگوریتم CBS پیاده سازی شده است که از یک استراتژی جستجوی سلسله مراتبی استفاده می کند. در سطح بالا، CBS درختی از کاندیداهای راه‌حل[۷] بالقوه را بررسی می‌کند، که در آن هر گره مجموعه‌ای از محدودیت‌های[۸] تحمیل‌شده بر مسیرهای یک عامل را نشان می‌دهد. در سطح پایین، CBS از الگوریتم A* برای برنامه‌ریزی مسیرهای هر عامل به طور مجزا که محدودیت های جستجوی سطح بالا را برآورده می کند، استفاده می کند.

در این پروژه تلاش شده تا با ترکیب CBS با جستجوی A*، راه حل های بهینه یا تقریباً بهینه برای مسئله MAPF پیدا شده و این اطمینان حاصل شود که همه عوامل به اهداف مربوطه خود می رسند و در عین حال به محدودیت های اجتناب از برخورد و موانع ثابت پایبند هستند.

---

[1] Maze
[2] Grid
[3] Node
[4] Block
[5] Agent
[6] Conflict
[7] Solution
[8] Constraints

۱-۲- تاریخچه:

مسیریابی چند عاملی (MAPF) مشکلی است که در حوزه‌های مختلفی مانند روباتیک، بازی‌های ویدیویی و کنترل ترافیک ایجاد می‌شود، که در آن چندین عامل باید در یک محیط حرکت کنند و از برخورد اجتناب کنند. این مشکل به طور گسترده در زمینه بررسی شده است. تحقیق هوش مصنوعی و عملیات

جستجوی مبتنی بر تعارض (CBS) یک الگوریتم محبوب برای حل مسائل MAPF است که توسط Guni Sharon، Roni Stern، Ariel Felner و Nathan R. Sturtevant در سال ۲۰۱۵ معرفی شد. CBS یک الگوریتم جستجوی دو سطحی است که یک جستجوی سطح بالا برای حل تعارض بین عوامل و جستجوی سطح پایین برای برنامه ریزی مسیرهای عامل فردی را ترکیب می کند. [۱]

ریشه های CBS را می توان در کار بر روی برنامه ریزی و هماهنگی چند عامله در اوایل دهه ۲۰۰۰ جستجو کرد، مانند کار سستا، اودی و اسمیت در مورد هماهنگی مریخ نوردهای متعدد. الگوریتم *A، که به عنوان جستجوی سطح پایین در CBS استفاده می‌شود، ریشه در کار اصلی هارت، نیلسون و رافائل در سال ۱۹۶۸ دارد. در طول سال‌ها، پیشرفت‌ها و توسعه‌های مختلفی برای CBS برای بهبود آن پیشنهاد شده است. عملکرد و کاربرد، مانند ادغام اکتشافی، موازی سازی، و مدیریت محیط های پویا [2]

۱-۳- هدف از انجام پروژه:

مقالات زیادی در حوزه نحوه پیاده سازی الگوریتم CBS بصورت تئوری نوشته شده‌اند، اما طبق تحقیقات انجام شده، بر روی سایت های Github و Gitlab، هیچ Repository[1] پیاده‌سازی انحصاری الگوریتم CBS را برای MAPF با استفاده از زبان برنامه نویسی Python به طوری که تعداد عوامل و هزارتوی مورد بررسی پویا[2] باشد، پیاده سازی نشده است.

۱-٤- ضرورت انجام پژوهش:

یافتن مسیر چند عاملی (MAPF) یک مشکل اساسی است که در حوزه‌های مختلف مانند روباتیک، بازی‌های ویدیویی و کنترل ترافیک ایجاد می‌شود، جایی که چندین عامل باید در یک محیط حرکت کنند و در عین حال از برخورد با یک دیگر اجتناب کنند. انجام تحقیقات بر روی الگوریتم های MAPF مانند الگوریتم CBS برای توسعه راه حل های کارآمد و مقیاس پذیر برای رسیدگی به این چالش های دنیای واقعی بسیار مهم است.

---

[1] یک مخزن فناوری اطلاعات و ذخیره کد یا مکانی متمرکز است که در آن داده‌ها بطور سازمان یافته، در حافظه رایانه یا فضای ابری، ذخیره و نگهداری می شوند

[2] Dynamic

تحقیقات برای پرداختن به چالش‌های علمی و آکادمیک در زمینه هوش مصنوعی و تحقیقات عملیاتی، مانند تعریف اهداف MAPF، و توسعه مدل‌های استراتژیک برای هماهنگی چند عامل در یک سیستم ضروری است.

CBS، به عنوان یک الگوریتم برجسته برای حل مسائل MAPF، نیاز به تحقیقات مداوم برای بهبود عملکرد، کاربرد و ادغام با سایر تکنیک‌ها دارد.

علاوه بر این، تحقیق برای جستجو و تجزیه و تحلیل مجموعه‌های داده بزرگ، مانند آن‌هایی که در تحقیقات Proteomics[1] تولید می‌شوند ضروری است، که در آن ابزارهای محاسباتی مانند CBS و A* برای پردازش و تجسم حجم وسیعی از داده‌ها ضروری هستند. الگوریتم‌های MAPF می‌توانند به طور بالقوه برای تجزیه و تحلیل و تجسم داده‌های بیولوژیکی پیچیده استفاده شوند که ضرورت انجام تحقیقات در این زمینه را برجسته می‌کند.

۱-۵- کاربردهای انجام پروژه:

یافته‌های تحقیق در مورد MAPF و CBS می‌تواند به توسعه تکنیک‌های مدل‌سازی جدید و محیط‌های حل مسئله برای کاربردهای مختلف کمک کند، مانند:

۱-۱-۵-۱- رباتیک و اتوماسیون:

الگوریتم های MAPF مانند CBS را می توان برای هماهنگ کردن حرکات چند ربات در انبارها، کارخانه ها یا سایر تنظیمات صنعتی به کار برد و از رفت و آمد های کارآمد و بدون برخورد اطمینان حاصل کرد.

۲-۱-۵-۱- توسعه بازی های ویدیویی:

الگوریتم های MAPF را می توان برای کنترل حرکات شخصیت های غیربازیکن (NPC[2]) در بازی های ویدیویی استفاده کرد و رفتارهای رفت و آمد های واقع بینانه و هوشمند را ارائه کرد.

۳-۱-۵-۱- مدیریت ترافیک:

CBS و دیگر الگوریتم های MAPF را می توان برای بهینه سازی مسیریابی وسایل نقلیه در شبکه های حمل و نقل، کاهش ازدحام و بهبود جریان ترافیک به کار برد.

۱- لجستیک[3] و مدیریت زنجیره تامین[4]:

تکنیک‌های MAPF را می‌توان برای برنامه‌ریزی و هماهنگ کردن حرکات چندین عامل، مانند وسایل نقلیه تحویلی یا رباتهای خودران، در عملیات‌های لجستیکی پیچیده مورد استفاده قرار داد. این ربات‌ها در کارخانه ها و انبارهای بزرگ در تمام دنیا مورد استفاده قرار میگیرد.

---

[1] ارزیابی کامل عملکرد و ساختار پروتئین ها برای درک ماهیت یک موجود زنده است

[2] Non-Player Character

[3] Logistics

[4] Supply Chain

۴-۱-۵-۱- تجزیه و تحلیل داده‌های بیولوژیکی:

همانطور که قبلاً ذکر شد، الگوریتم‌های MAPF می‌توانند به طور بالقوه برای تجزیه و تحلیل و تجسم داده‌های پیچیده بیولوژیکی، مانند داده‌های پروتئومیکس، استفاده شوند که بینش و اکتشافات جدیدی را در تحقیقات زیست‌شناسی سیستم‌ها ممکن می‌سازد.

با انجام تحقیقات بر روی MAPF و CBS، به توسعه راه‌حل‌های کارآمد و مقیاس‌پذیر برای مشکلات هماهنگی چند عاملی کمک می‌کنید و کاربردهای عملی را در حوزه‌های مختلف، از رباتیک و بازی‌های ویدیویی گرفته تا مدیریت ترافیک، تدارکات، و حتی تحلیل داده‌های بیولوژیکی ممکن می‌سازد.

۱-۶- نوآوری پروژه:

تصویرسازی[1] اطلاعات با روش نقشه گرمایی[2]، چاپ قدم به قدم حرکت همزمان[3] عامل‌ها در هزارتو، تصادفی کردن جایگاه موانع و نقاط شروع و پایان عامل‌ها و همچنین پویا سازی برنامه برای تعیین دلخواه تعداد عامل‌ها، اندازه هزارتو و تعداد موانع از جمله موارد نوآورانه پروژه میباشد.

۱-۷- ساختار گزارش:

در فصل اول ساختار کلی، اهداف، نیازها و کاربردها انجام پروژه شرح داده شده است.

در فصل دوم نحوه پیاده سازی الگوریتم CBS با استفاده از الگوریتم A* برای MAPF به صورت کامل توضیح داده شده است.

[1] Visualization
[2] Heat Map
[3] Synchronous

فصل ۲– تعریف اصطلاحات:

۲-۱- مقدمه

در این فصل اصطلاحات تخصصی و همچنین تئوری الگوریتم های استفاده شده در گزارش بصورت تفصیلی توضیح داده خواهد شد.

۲-۲- الگوریتم *A:

الگوریتم *A یکی از الگوریتم‌های معروف جستجوی مسیر در حوزه علوم کامپیوتر و هوش مصنوعی است. این الگوریتم برای یافتن کوتاه‌ترین مسیر از یک نقطه شروع به یک هدف در یک گراف استفاده می‌شود، جایی که هر گره در گراف یک مکان را نمایان می‌کند و یال‌ها بین گره‌ها نمایانگر اتصالات یا حرکات ممکن بین مکان‌ها هستند. *A به ویژه به دلیل ترکیب اصول الگوریتم دایکسترا[1] و رویکرد هیوریستیک[2]، به عنوان یک الگوریتم کارآمد شناخته می‌شود.

۲-۲-۱- اجزای الگوریتم *A:

۲-۲-۱-۱- گره‌ها:

نمایانگر یک نقطه در گراف است. گره ها دارای والد، مکان و توابع f، g و h میباشند. نحوه تعریف کردن یک گره در کد نوشته شده را در شکل ۲-۲-۱ مشاهده میکنید

Node

```python
class Node():

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position

    def __repr__(self) -> str:
        return f"{self.position}"
```

شکل ۲-۲-۱ : تعریف تابع گره در پایتون

[1] Dijkstra's Algorithm
[2] Heuristic

۲-۱-۲-۲- یال‌ها:

نمایانگر اتصال بین دو گره است، نشان‌دهنده این است که حرکت یا گذر معتبری بین این نقاط وجود دارد.

۲-۱-۲-۳- گراف:

مجموعه کاملی از گره‌ها و یال‌ها که محیط یا فضای مسئله را تعریف می‌کنند.

۲-۱-۲-۴- تابع (f(n)):

این تابع کل هزینه تخمینی ارزانترین مسیر از گره شروع به گره هدف را که از گره (n) می گذرد را نشان می دهد. به عنوان مجموع هزینه واقعی رسیدن به گره (n) از گره شروع ((g(n))) و هزینه تخمینی رسیدن به گره هدف از گره (((h(n))) (n) تعریف می شود. . از نظر ریاضی، ((f(n) = g(n) + h(n))

۲-۱-۲-۵- تابع (g(n)):

این تابع هزینه واقعی رسیدن به گره (n) را از گره شروع محاسبه می کند. این نشان دهنده هزینه مسیری است که تاکنون در طول مسیر از گره شروع به گره (n) متحمل شده است. در زمینه مسیریابی، اساساً هزینه مسیری است که تاکنون کشف شده است.

۲-۱-۲-۶- تابع هیوریستیک (h(n)):

هزینه تخمین زده شده برای حرکت از یک خانه داده شده در شبکه به مقصد نهایی است. از h معمولا با عنوان هیوریستیک یاد می‌شود. هیوریستیک چیزی به جز نوعی حدس هوشمندانه نیست. کاربر واقعا فاصله واقعی را تا هنگام یافتن مسیر نمی‌داند، زیرا هر مانعی (دیوار، آب و سایر موانع) ممکن است در مسیر باشد. راه‌های زیادی برای محاسبه h وجود داردکه رایج ترین آنها فاصله اقلیدسی و منهتن است. هیوریستیک فاصله اقلیدسی، فاصله خط مستقیم و هیوریستیک منهتن، مجموع فاصله افقی و فاصله عمودی گره‌های فعلی و هدف میباشد.

۲-۱-۲-۷- لیست باز (Open List):

لیست باز شامل گره‌هایی است که الگوریتم هنوز برای آن‌ها ارزیابی نکرده است.

در هر مرحله، گره با کمترین مقدار تابع f(n) از بین گره‌های لیست باز انتخاب می‌شود و برای ارزیابی انتخاب می‌شود.

پس از انتقال یک گره از لیست باز به لیست بسته، آن گره دیگر مورد ارزیابی قرار نمی‌گیرد.

۲-۱-۲-۸- لیست بسته (Closed List):

لیست بسته شامل گره‌هایی است که الگوریتم قبلاً آن‌ها را ارزیابی کرده و از لیست باز حذف کرده است. این گره‌ها معمولاً به عنوان گره‌هایی که الگوریتم قبلاً برای آن‌ها بهینه شده است، نگهداری می‌شوند.

استفاده از لیست بسته کمک می‌کند تا الگوریتم از تکراری شدن مسیرها جلوگیری کند و بهینه‌سازی در جستجو انجام شود.

۲-۲-۲- مراحل الگوریتم *A:

۱-۲-۲-۲- مرحله اول: مقداردهی اولیه

در این مرحله، لیست باز و لیست بسته را مقداردهی اولیه می‌کنیم.

همچنین، مقدار g وh برای گره شروع محاسبه می‌شود.

مقدار g گره شروع برابر با صفر و مقدار h با استفاده از تابع هیوریستیک به عنوان تخمینی از هزینه رسیدن به گره هدف محاسبه می‌شود.

۲-۲-۲-۲- مرحله دوم: گسترش و ارزیابی

در این مرحله، گره‌ها به ترتیبی که با استفاده از تابع f به کمترین مقدار f مرتب شده‌اند، از لیست باز انتخاب می‌شوند و ارزیابی می‌شوند.

برای هر گره انتخاب شده:

گره به لیست بسته منتقل می‌شود تا دوباره مورد ارزیابی قرار نگیرد.

همسایه‌های گره بررسی می‌شوند و برای هر همسایه:

مقدار g موقت برای آن محاسبه می‌شود.

اگر گره مجاور در لیست باز نبود یا مقدار g موقت کمتر از مقدار g فعلی گره مجاور بود:

مقدار g و h گره مجاور به‌روزرسانی می‌شود.

والد گره مجاور به گره کنونی تعیین می‌شود.

گره مجاور به لیست باز اضافه می‌شود.

۳-۲-۲-۲- مرحله سوم: پایان الگوریتم

اگر گره هدف در لیست باز قرار داشت، به این معناست که مسیری به گره هدف پیدا شده است و الگوریتم پایان می‌یابد. در غیر این صورت، اگر لیست باز خالی شود و هدف هنوز در لیست بسته نبود، به این معناست که مسیری برای رسیدن به هدف وجود ندارد.

۳-۲-۲- مزایای *A:

۱-۳-۲-۲- کامل بودن:

*A تضمین می‌کند که اگر یک راه‌حل وجود داشته باشد، آن را پیدا کند.

۲-۳-۲-۲- بهینگی:

اگر تابع هیوریستیک معتبر باشد، *A مسیر کوتاه‌تر را پیدا می‌کند (هرگز هزینه واقعی را بیشتر از حد تخمین ندهد)

۳-۳-۲-۲- کارآیی:

استفاده از تابع هیوریستیک باعث اولویت‌بندی در جستجو و کاهش تعداد گره‌های گسترش یافته نسبت به الگوریتم‌های جستجوی دیگر می‌شود. این نکته مهم است که کارآیی *A به شدت به دقت تابع هیوریستیک وابسته است. اگر تابع هیوریستیک معتبر و پایدار باشد، *A مسیرهای بهینه را با کارآیی بالا در برنامه‌های مختلف از جمله رباتیک، بازی‌های ویدئویی و مسیریابی شبکه پیدا می‌کند.

۲-۳- الگوریتم CBS:

CBS از طریق دو سطح جستجوی مجزا عمل می کند: جستجوهای سطح بالا و سطح پایین. در ابتدا، مسیر های هر عامل به طور مجزا مشخص میشود که ممکن است با یکدیگر تعارض نیز داشته باشند.

در جستجوی سطح بالا، یک درخت محدودیت (CT[1]) برای مدیریت محدودیت‌ها برای هر عامل جداگانه استفاده می‌شود. هر گره در CT محدودیت‌های خاصی را در رابطه با زمان و مکان برای یک عامل واحد ایجاد می‌کند.

به طور همزمان، در جستجوی سطح پایین، یک بررسی کامل برای همه عوامل در هر گره از CT انجام می شود. هدف این جستجو، یافتن مسیرهای تک عاملی است که با محدودیت های مشخص شده در آن گره CT خاص همسو هستند.

اگر تعارض‌ها حتی پس از جستجوی سطح پایین ادامه پیدا کند، که نشان‌دهنده مواردی است که چندین عامل به طور همزمان یک مکان را اشغال می‌کنند، گره سطح بالا مربوطه به‌عنوان غیر هدف علامت‌گذاری می‌شود. جستجوی سطح بالا سپس با افزودن مکرر گره‌های جدید با محدودیت‌های اضافی با هدف حل تعارض‌های در حال بوجود آمدن، ادامه می‌یابد.

[1] Constraint Tree

MAPF یک مشکل محاسباتی چالش برانگیز را در کاربردهای مختلف دنیای واقعی مانند روباتیک، لجستیک و سیستم های حمل و نقل ارائه می دهد. در MAPF، چندین عامل با موقعیت‌های شروع و هدف فردی باید در یک محیط مشترک حرکت کنند و در عین حال از برخورد اجتناب کنند و محدودیت‌های مختلف را نیز درنظر بگیرند. CBS به عنوان یک رویکرد الگوریتمی پیشرو برای پرداختن به MAPF کارآمد با حل سیستماتیک تعارض‌ها بین مسیرهای عامل ها بوجود آمده است.

۲-۳-۱- جستجوی سطح بالا

جستجوی سطح بالا CBS به عنوان یک ساختار داده برای مدیریت محدودیت ها برای هر عامل جداگانه عمل می کند. هر گره در CT مربوط به یک عامل خاص است و محدودیت‌هایی را در بر می‌گیرد که نواحی محیط و فواصل زمانی را که عامل باید از آنها اجتناب کند، تعریف می‌کند. در ابتدا، گره ریشه CT حاوی هیچ محدودیتی نیست، که نشان دهنده وضعیت اولیه جستجو است. گره های بعدی با اضافه کردن مکرر محدودیت ها برای حل تعارض های شناسایی شده در طول فرآیند مسیریابی تولید می شوند. هدف کلی از جستجوی سطح بالا، عبور از CT به طور موثر برای شناسایی مسیرهای بدون محدودیت برای همه عوامل است، در نتیجه از راه حلی بدون تعارض در سطح کلی اطمینان حاصل می شود.

۲-۳-۲- جستجوی سطح پایین

در CBS، مرحله جستجوی سطح پایین در هر گره از CT اجرا می‌شود تا مسیرهای بهینه را برای عوامل بصورت جداگانه محاسبه کند و در عین حال به محدودیت‌های مربوط به خود پایبند باشد. به طور معمول، این شامل استفاده از گونه‌ای از الگوریتم جستجوی A*، مانند STA*[1] است که هر دو بعد مکانی و زمانی را در فرآیند مسیریابی ادغام[۲] می‌کند. با در نظر گرفتن محدودیت‌های مکانی و زمانی، جستجوی سطح پایین می‌تواند به طور موثر مسیرهایی را شناسایی کند که از برخورد جلوگیری و محدودیت‌های مشخص شده برای هر عامل را نیز برآورده می‌کند. خروجی جستجوی سطح پایین مجموعه ای از مسیرهای بدون تعارض است که متناسب با محدودیت های اعمال شده در گره CT مربوطه است.

۲-۳-۳- حل تعارض

حل تعارض نقش مهمی در الگوریتم CBS ایفا می کند، به ویژه زمانی که تعارض در طول جستجوی سطح پایین ایجاد می شود. تعارض زمانی رخ می‌دهد که دو یا چند عامل قصد دارند مکان یکسانی را در

[1] Space-Time A*
[2] Merge

هزارتو به طور همزمان اشغال کنند. برای حل تعارض‌ها، CBS گره های فرزند[1] جدیدی را در CT با محدودیت های اضافی با هدف کاهش تعارض ایجاد می کند. محدودیت‌های خاص تحمیل‌شده برای حل تعارض ممکن است بر اساس استراتژی انتخاب شده متفاوت باشد، که می‌تواند شامل اولویت‌بندی عوامل خاص، تنظیم مسیر آنها، یا محدود کردن دسترسی به مکان‌ها و فواصل زمانی خاص باشد.[۱] سپس جستجوی سطح بالا به جستجو خود در CT ادامه می‌دهد و این فرآیند تا زمانی که به گره‌ای دست یابد که در آن، همه عوامل دارای مسیرهای بدون تعارض هستند، یا تا زمانی که فضای جستجو تمام شود، ادامه پیدا میکند.

۲-۳-٤- کارایی و پیشرفت

کارایی CBS از تجزیه مسئله MAPF به جستجوهای تک عامل مستقل در سطح پایین، همراه با مکانیسم های حل تعارض در سطح بالا ناشی می شود. با گذشت زمان، محققان پیشرفت های مختلفی را برای بهینه سازی بیشتر CBS پیشنهاد کرده‌اند.[۱] اینها شامل توسعه طرح‌های اولویت‌بندی برای عوامل بر اساس عواملی مانند فوریت یا اهمیت، ادغام توابع اکتشافی برای هدایت جستجو به سمت قسمت های با محدودیت کم CT، و پیشرفت در تکنیک‌های جستجوی سطح پایین برای بهبود کارایی محاسباتی و کیفیت مسیر است.

۲-٤- نتیجه گیری:

الگوریتم جستجوی مبتنی بر تعارض (CBS) با استفاده از ترکیبی از مدیریت محدودیت سطح بالا و مسیریابی سطح پایین، راه حلی عملی و مؤثر برای مشکلات مسیریابی چند عاملی (MAPF) ارائه می دهد. کارایی و اثربخشی آن، آن را به ابزاری ارزشمند در حوزه‌هایی تبدیل می‌کند که مسیریابی و حرکت هماهنگ چند عاملی در آن اهمیت دارد. در جستجوی سطح پایین به طور خلاصه، الگوریتم جستجوی *A نشان دهنده یک پیشرفت محوری[2] در حوزه الگوریتم های مسیریابی است. توانایی آن در ترکیب استراتژی های جستجوی آگاهانه با تعیین مسیر بهینه، آن را در زمینه های مختلف ضروری می کند. علیرغم پیچیدگی های محاسباتی، *A به دلیل کارایی، تطبیق پذیری و کاربرد آن در طیف وسیعی از مسائل دنیای واقعی، یک الگوریتم بنیادی باقی مانده است. با این وجود، طراحی و انتخاب درست توابع اکتشافی، نقش مهمی در استفاده از پتانسیل کامل *A در کاربردهای مختلف دارد.

[1] Child Nodes
[2] Pivotal

فصل ۳– پیاده سازی:

۳-۱- مقدمه

این بخش اجرای الگوریتم CBS را برای MAPF با استفاده از الگوریتم مسیریابی A* در پایتون ترسیم می‌کند. Multi-Agent Pathfinding یک نگرانی محوری در حوزه‌های مختلف ایجاد می‌کند، که پیمایش هماهنگ چندین عامل را در یک محیط مشترک ضروری می‌کند. این پیاده سازی با هدف اثبات کارایی و عملی بودن الگوریتم CBS در پرداختن به چالش های پیچیده ذاتی در سناریوهای MAPF است.

الگوریتم CBS به عنوان یک استراتژی برجسته برای حل معضلات MAPF ظاهر می شود و به طور مکرر تعارض‌های بین مسیرهای عامل را حل می کند. این تلاش شامل ادغام الگوریتم مسیریابی A* به عنوان یک روال اساسی است که استخراج مسیرهای بهینه را برای هریک از عوامل تسهیل می‌کند و همزمان تعارض‌های بالقوه را با همتایان خود تطبیق می‌دهد.

۳-۲- پیش نیاز های اجرای کد:

برای اجرای این برنامه، سیستم مجری برنامه میبایستی زبان Python و همچنین کتابخانه Plotly را بر روی خود نصب داشته باشد. IDE[1] پیشنهادی برای اجرای برنامه، Visual Studio Code میباشد.

دو فایل در پوشه پروژه با نام های cbs.py و cbs.ipynb وجود دارند. برای اجرای cbs.py نیازی به نصب برنامه خاصی نمیباشد و کافیست پس از بازکددر IDE، کلیدهای Ctrl + F5 فشرده شوند. برای اجرای cbs.ipynb نیاز است قبل از فشردن کلیدهای یادشده، Jupyter Notebook از طریق قسمت Extensions در Visual Studio Code نصب شود.

تنها کاری که غیر از موارد یاد شده نیاز است تا انجام شود این است که در قسمت Main Function مقدار مجهول ها متناسب با نیاز کاربر تغییر کند.

۳-۳- کتابخانه های استفاده شده:

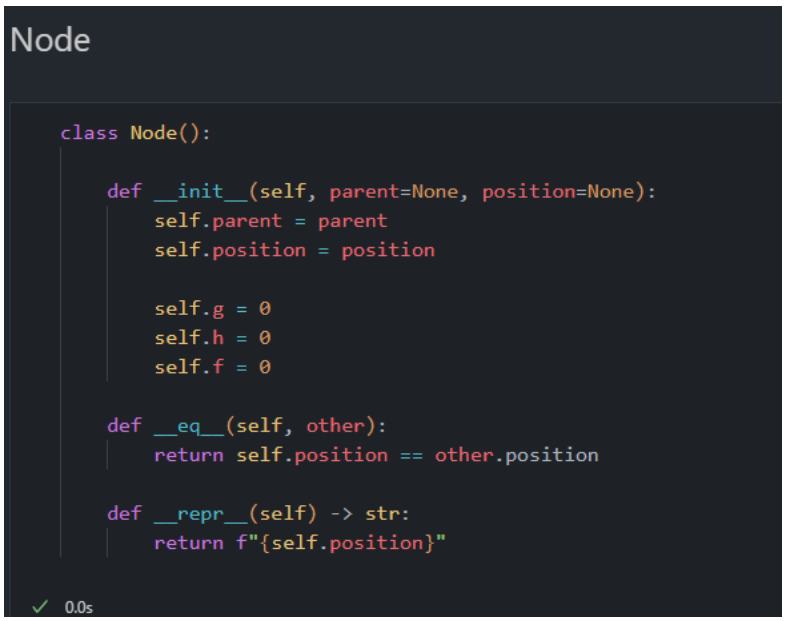
در این کد از دو کتابخانه Plotly برای مصورسازی[2] مسیرها و Random برای تصادفی سازی جایگاه موانع و نقاط شروع و پایان عامل‌ها استفاده میشود.

[1] Integrated Development Environment
[2] Visualization

شکل ۱-۳-۳: وارد کردن کتابخانه‌ها

۳-٤- تعریف گره:

برای تعریف گره بعنوان یک المان در کد، ویژگی های آن که شامل والد، مکان و توابع f و g و h است طبق شکل ۱-۴-۳ تعریف شده است.



شکل ۱-۴-۳: تعریف گره

۳-۵- تعریف عامل:

هریک از عوامل دارای ویژگی های گره آغاز، گره هدف، گره کنونی، لیست باز، لیست بسته و مسیر طی شده میباشد که طبق شکل ۱-۵-۳ تعریف شده است.

Agent

```python
class Agent:
    def __init__(self, start_position, end_position):
        self.start_node = Node(None, start_position)
        self.end_node = Node(None, end_position)
        self.current_node = Node(None, start_position)
        self.open_list = []
        self.close_list = []
        self.path = []
```

شکل ۱-۵-۳: تعریف عامل

۳-۶- ایجاد نقاط تصادفی برای گره های آغازین و پایانی عامل ها:

کد تصادفی سازی گره های آغازین و پایانی عامل ها با دو شرط میبایستی پیاده سازی شده است.

۱- گره های آغازین و پایانی عامل ها یکسان نباشد

۲- گره های آغازین و پایانی عامل ها روی موانع ایجاد نشود.

Random points Generator

```python
def generate_random_points(maze_size, maze):
    while True:
        start_x = random.randint(0, maze_size - 1)
        start_y = random.randint(0, maze_size - 1)
        end_x = random.randint(0, maze_size - 1)
        end_y = random.randint(0, maze_size - 1)
        if (start_x, start_y) != (end_x, end_y) and maze[start_x][start_y] != 0 and maze[end_x][end_y] != 0:
            return (start_x, start_y), (end_x, end_y)
```

✓ 0.0s

Agent Generator

```python
def generate_agents(number_of_agents, maze_size, maze):

    agents = []
    for _ in range(number_of_agents):
        start, end = generate_random_points(maze_size, maze)
        agent = Agent(start, end)
        agents.append(agent)
    return agents
```

✓ 0.0s

شکل ۱-۶-۳: ایجاد عامل بصورت تصادفی

۳-۷- تعریف هزارتو:

طبق شکل ۱-۷-۳، تابعی نوشته میشود که از کاربر اندازه و تعداد موانع را گرفته و بصورت تصادفی یک هزارتو بوجود می‌آورد.

Maze

```python
def create_maze(maze_size, zeros_number):
    # Create a new maze with all fives as free positions
    new_maze = [[5] * maze_size for _ in range(maze_size)]

    # Generate 20 random coordinates and set them to zero
    zeros_generated = 0

    while zeros_generated < zeros_number:
        x = random.randint(0, maze_size - 1)
        y = random.randint(0, maze_size - 1)
        if new_maze[x][y] == 5:
            new_maze[x][y] = 0
            zeros_generated += 1
    maze = []
    # Print the new maze
    for row in new_maze:
        maze.append(row)
    return maze
```

✓ 0.0s

شکل ۱-۷-۳: تعریف هزارتو

۳-۸- ایجاد و تغییر مقادیر لیست باز و لیست بسته:

در کد پیاده سازی شده، تمامی عوامل بصورت همزمان مسیر های ممکن در هشت جهت را به لیست باز خود اضافه کرده و پس از بررسی هریک از مسیرها، کم هزینه ترین مسیر را به لیست بسته اضافه میکنند.

```
for agent in agents:
    counter += 1
    if agent.current_node != agent.end_node:
    # Generate children
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0),(-1, -1), (1, 1), (-1, 1), (1, -1)]: # Adjacent squares

            # Get node position
            adjacent_position = (agent.current_node.position[0] + new_position[0], agent.current_node.position[1] + new_position[1])

            # Make sure within range
            if adjacent_position[0] > maze_x or adjacent_position[0] < 0 or adjacent_position[1] > maze_y or adjacent_position[1] < 0:
                continue

            # Make sure walkable terrain
            if maze[adjacent_position[0]][adjacent_position[1]] == 0:
                continue

            # Create new node
            new_node = Node(agent.current_node, adjacent_position)


            # Child is on the closed list
            if new_node in agent.close_list:
                continue

            # Create the f, g, and h values
            new_node.g = agent.current_node.g + 1
            new_node.h = ((new_node.position[0] - agent.end_node.position[0]) ** 2) + ((new_node.position[1] - agent.end_node.position[1]) ** 2)
            new_node.f = new_node.g + new_node.h


            # new_node is already in the open list
            if new_node not in agent.close_list:
                if new_node in agent.open_list:
                    conflicted_agent = agent.open_list.index(new_node)
                    duplicated = agent.open_list[conflicted_agent]
                    if new_node.g < duplicated.g:
                        agent.open_list.append(new_node)
                else:
                    agent.open_list.append(new_node)
```

شکل ۱-۸-۳: نحوه عملکرد لیست ها

۳-۹- پیاده سازی CBS:

الگوریتم CBS که منطق اصلی را برای حل تعارض و برنامه‌ریزی مسیر در بر می‌گیرد، به صورت توابع منسجمی به صورتی که در در تصویر مشخص شده ساختار یافته است.

```
        for agent in agents:
            final_path = []
            goal = agent.path[-1]
            while goal is not None:
                final_path.append(goal)
                goal = goal.parent
            final_path = final_path[::-1]
            final_paths.append(final_path)
        to_add = []
        for i, path in enumerate(final_paths):
            for j, other_path in enumerate(final_paths):
                if i < j:
                    for k, item in enumerate(path):
                        for p, other_item in enumerate(other_path):
                            if k == p and item == other_item:
                                if item.parent.h < other_item.parent.h:
                                    to_add.append((i, k, item.parent))
                                else:
                                    to_add.append((j, p, other_item.parent))

        for l, i, val in to_add:
            final_paths[l].insert(i, val)
        final_path_primes = final_paths
```

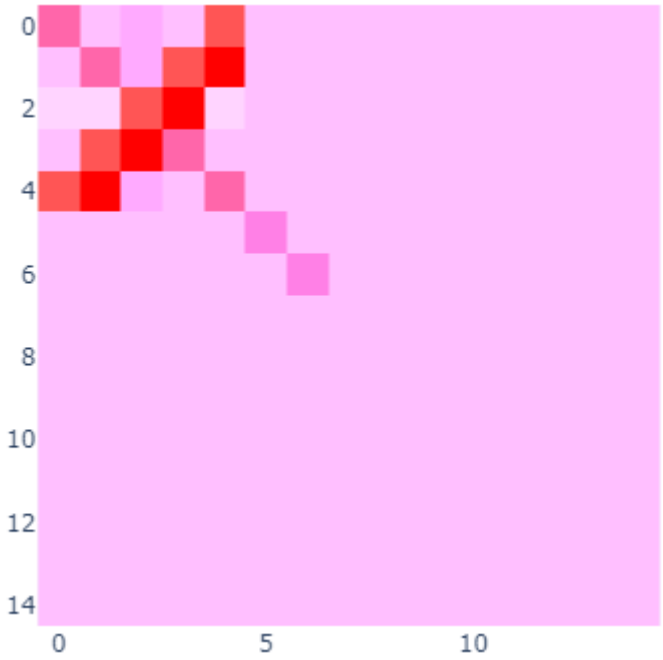
شکل ۱-۹-۳: پیاده سازی CBS

۳-۱۰- نمونه پیاده سازی ها:

در این قسمت نمونه هایی از استفاده از کد پیاده سازی شده در حالات مختلف را بررسی میکنیم.

```
start and end positions of agent  1 is:  (2, 0) (2, 4)
start and end positions of agent  2 is:  (0, 2) (4, 2)
start and end positions of agent  3 is:  (0, 0) (6, 6)
start and end positions of agent  4 is:  (4, 4) (0, 0)
start and end positions of agent  5 is:  (0, 4) (4, 0)
start and end positions of agent  6 is:  (1, 4) (4, 1)
agent  1  has the path [(2, 0), (2, 1), (2, 1), (2, 1), (2, 1), (2, 2), (2, 3), (2, 4)]
agent  2  has the path [(0, 2), (1, 2), (1, 2), (1, 2), (1, 2), (1, 2), (2, 2), (3, 2), (4, 2)]
agent  3  has the path [(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6)]
agent  4  has the path [(4, 4), (3, 3), (3, 3), (2, 2), (1, 1), (0, 0)]
agent  5  has the path [(0, 4), (1, 3), (1, 3), (1, 3), (2, 2), (3, 1), (4, 0)]
agent  6  has the path [(1, 4), (2, 3), (3, 2), (4, 1)]
```

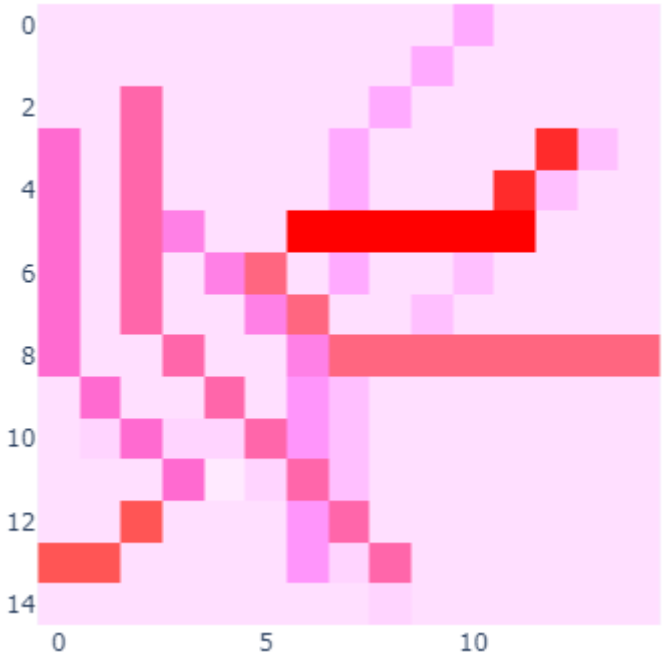
پیاده سازی۱-۱



پیاده سازی ۱-۲

```
start and end positions of agent  1 is:  (10, 7) (11, 4)
start and end positions of agent  2 is:  (14, 8) (10, 1)
start and end positions of agent  3 is:  (3, 13) (12, 7)
start and end positions of agent  4 is:  (0, 10) (6, 7)
start and end positions of agent  5 is:  (7, 5) (13, 6)
start and end positions of agent  6 is:  (8, 6) (4, 2)
start and end positions of agent  7 is:  (11, 3) (3, 0)
start and end positions of agent  8 is:  (13, 8) (2, 2)
start and end positions of agent  9 is:  (6, 5) (8, 14)
start and end positions of agent  10 is:  (12, 2) (13, 0)
start and end positions of agent  11 is:  (3, 12) (5, 10)
start and end positions of agent  12 is:  (5, 6) (5, 11)
agent  1  has the path [(10, 7), (11, 6), (11, 5), (11, 4)]
agent  2  has the path [(14, 8), (13, 7), (12, 6), (11, 5), (10, 4), (10, 3), (10, 2), (10, 1)]
agent  3  has the path [(3, 13), (4, 12), (5, 11), (6, 10), (7, 9), (8, 8), (9, 7), (10, 7), (11, 7), (12, 7)]
agent  4  has the path [(0, 10), (1, 9), (2, 8), (3, 7), (4, 7), (5, 7), (6, 7)]
agent  5  has the path [(7, 5), (8, 6), (9, 6), (10, 6), (11, 6), (12, 6), (13, 6)]
agent  6  has the path [(8, 6), (7, 5), (6, 4), (5, 3), (4, 2)]
agent  7  has the path [(11, 3), (10, 2), (9, 1), (8, 0), (7, 0), (6, 0), (5, 0), (4, 0), (3, 0)]
agent  8  has the path [(13, 8), (12, 7), (11, 6), (10, 5), (9, 4), (8, 3), (7, 2), (6, 2), (5, 2), (4, 2), (3, 2), (2, 2)]
agent  9  has the path [(6, 5), (7, 6), (8, 7), (8, 8), (8, 9), (8, 10), (8, 11), (8, 12), (8, 13), (8, 14)]
agent  10  has the path [(12, 2), (13, 1), (13, 0)]
agent  11  has the path [(3, 12), (4, 11), (5, 10)]
agent  12  has the path [(5, 6), (5, 7), (5, 8), (5, 9), (5, 10), (5, 11)]
```

پیاده سازی ۲-۱



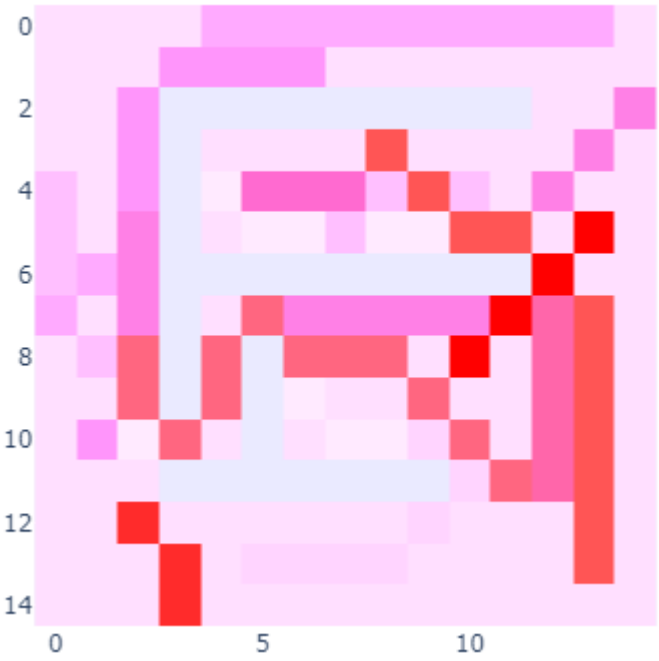
پیاده سازی ۲-۲

```
maze = [[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5],
        [5, 5, 5, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5],
        [5, 5, 5, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 0, 5, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 0, 5, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 5, 5, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5],
        [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]]
```

پیاده سازی ۳-۱

```
start and end positions of agent  1 is:  (4, 4) (10, 2)
start and end positions of agent  2 is:  (4, 12) (13, 5)
start and end positions of agent  3 is:  (5, 7) (4, 0)
start and end positions of agent  4 is:  (7, 0) (0, 13)
start and end positions of agent  5 is:  (10, 1) (1, 6)
start and end positions of agent  6 is:  (2, 14) (5, 2)
start and end positions of agent  7 is:  (4, 5) (4, 7)
start and end positions of agent  8 is:  (12, 13) (7, 12)
start and end positions of agent  9 is:  (11, 11) (8, 2)
start and end positions of agent  10 is:  (3, 8) (13, 13)
start and end positions of agent  11 is:  (12, 2) (14, 3)
start and end positions of agent  12 is:  (5, 13) (8, 10)
agent  1  has the path [(4, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9), (5, 10), (5, 11), (6, 12), (7, 11), (8, 10), (9, 9), (10, 8), (10, 7), (9, 6), (8, 6), (7, 5), (8, 4), (9, 4), (10, 3), (10, 2)]
agent  2  has the path [(4, 12), (5, 11), (6, 12), (7, 11), (8, 10), (9, 9), (10, 9), (11, 10), (12, 9), (13, 8), (13, 7), (13, 6), (13, 5)]
agent  3  has the path [(5, 7), (4, 8), (4, 9), (4, 10), (5, 11), (6, 12), (7, 11), (7, 10), (7, 9), (7, 8), (7, 7), (7, 6), (7, 5), (8, 4), (9, 4), (10, 3), (9, 2), (8, 1), (7, 0), (6, 0), (5, 0), (4, 0)]
agent  4  has the path [(7, 0), (6, 1), (5, 2), (4, 2), (3, 2), (2, 2), (1, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (0, 10), (0, 11), (0, 12), (0, 13)]
agent  5  has the path [(10, 1), (9, 2), (8, 2), (7, 2), (6, 2), (5, 2), (4, 2), (3, 2), (2, 2), (1, 3), (1, 4), (1, 5), (1, 6)]
agent  6  has the path [(2, 14), (3, 13), (4, 12), (5, 11), (6, 12), (7, 11), (7, 10), (7, 9), (7, 8), (7, 7), (7, 6), (7, 5), (8, 4), (9, 4), (10, 3), (9, 2), (8, 2), (7, 2), (6, 2), (5, 2)]
agent  7  has the path [(4, 5), (4, 6), (4, 7)]
agent  8  has the path [(12, 13), (11, 12), (10, 12), (9, 12), (8, 12), (7, 12)]
agent  9  has the path [(11, 11), (10, 10), (9, 9), (8, 8), (8, 7), (8, 6), (7, 5), (8, 4), (9, 4), (10, 3), (9, 2), (8, 2)]
agent  10  has the path [(3, 8), (4, 9), (5, 10), (5, 10), (5, 11), (5, 11), (6, 12), (7, 13), (8, 13), (9, 13), (10, 13), (11, 13), (12, 13), (13, 13)]
agent  11  has the path [(12, 2), (13, 3), (14, 3)]
agent  12  has the path [(5, 13), (6, 12), (7, 11), (8, 10)]
```

پیاده سازی ۳-۲



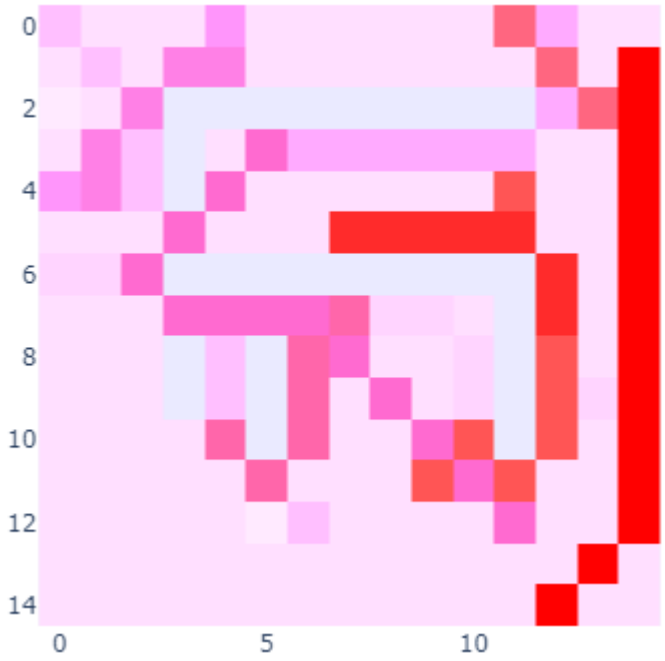
پیاده سازی ۳-۳

```
maze = [[5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5],
        [5, 5, 5, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 5, 5],
        [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 0, 5, 5, 5],
        [5, 5, 5, 0, 5, 0, 5, 5, 5, 5, 5, 0, 5, 5, 5],
        [5, 5, 5, 0, 5, 0, 5, 5, 5, 5, 5, 0, 5, 5, 5],
        [5, 5, 5, 5, 5, 0, 5, 5, 5, 5, 5, 0, 5, 5, 5],
        [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5],
        [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]]
```

پیاده سازی ۴-۱

```
start and end positions of agent  1 is:  (2, 0) (12, 5)
start and end positions of agent  2 is:  (6, 0) (6, 14)
start and end positions of agent  3 is:  (0, 0) (12, 6)
start and end positions of agent  4 is:  (4, 4) (0, 12)
start and end positions of agent  5 is:  (0, 4) (4, 0)
start and end positions of agent  6 is:  (1, 4) (4, 1)
start and end positions of agent  7 is:  (12, 11) (3, 5)
start and end positions of agent  8 is:  (7, 7) (10, 4)
start and end positions of agent  9 is:  (0, 11) (5, 14)
start and end positions of agent  10 is:  (11, 9) (4, 11)
start and end positions of agent  11 is:  (5, 7) (7, 12)
start and end positions of agent  12 is:  (14, 12) (1, 14)
agent  1  has the path [(2, 0), (3, 1), (4, 2), (5, 3), (6, 2), (7, 3), (8, 4), (9, 4), (10, 4), (11, 5), (12, 5)]
agent  2  has the path [(6, 0), (6, 1), (6, 2), (7, 3), (7, 4), (7, 5), (7, 6), (7, 7), (7, 8), (7, 9), (8, 10), (9, 10), (10, 10), (11, 11), (10, 12), (9, 13), (8, 14), (7, 14), (6, 14)]
agent  3  has the path [(0, 0), (1, 1), (2, 2), (3, 2), (4, 2), (5, 3), (6, 2), (7, 3), (8, 4), (9, 4), (10, 4), (11, 5), (12, 6)]
agent  4  has the path [(4, 4), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9), (3, 10), (3, 11), (2, 12), (1, 12), (0, 12)]
agent  5  has the path [(0, 4), (1, 3), (1, 3), (2, 2), (3, 1), (4, 0)]
agent  6  has the path [(1, 4), (1, 4), (1, 3), (2, 2), (1, 3), (1, 3), (2, 2), (3, 1), (4, 1)]
agent  7  has the path [(12, 11), (11, 10), (10, 9), (9, 8), (8, 7), (7, 6), (7, 5), (7, 4), (7, 3), (6, 2), (5, 3), (4, 4), (3, 5)]
agent  8  has the path [(7, 7), (8, 6), (9, 6), (10, 6), (11, 5), (10, 4)]
agent  9  has the path [(0, 11), (1, 12), (2, 13), (3, 14), (4, 14), (5, 14)]
agent  10  has the path [(11, 9), (10, 10), (11, 11), (10, 12), (9, 12), (8, 12), (7, 12), (6, 12), (5, 11), (4, 11)]
agent  11  has the path [(5, 7), (5, 8), (5, 9), (5, 10), (5, 11), (6, 12), (6, 12), (7, 12)]
agent  12  has the path [(14, 12), (13, 13), (12, 14), (11, 14), (10, 14), (9, 14), (8, 14), (7, 14), (6, 14), (5, 14), (4, 14), (3, 14), (2, 14), (1, 14)]
```

پیاده سازی ۴-۲



پیاده سازی ۴-۳

واژه نامه‌ی فارسی به انگلیسی

عامل	Agent
مسدود کردن	Block
تعارض	Conflict
جستجوی مبتنی بر تعارض	Conflict Based Search
محدودیت ها	Constraints
الگوریتم دایکسترا	Dijkstra's Algorithm
پویا	Dynamic
توری	Grid
نقشه حرارت	Heat Map
ابتکاری	Heuristic
لجستیک	Logistics
مارپیچ	Maze
مسیریابی چند عاملی	Multi-Agent Path Finding
گره	Node
شخصیت غیر بازیکن	Non-Player Character
محوری	Pivotal
راه حل	Solution
زنجیره تامین	Supply Chain
همزمان	Synchronous
تجسم	Visualization
عامل	Agent
مسدود کردن	Block
تعارض	Conflict
جستجوی مبتنی بر تعارض	Conflict Based Search
درخت محدودیت	Constraint Tree
محدودیت ها	Constraints
الگوریتم دایکسترا	Dijkstra's Algorithm
پویا	Dynamic
توری	Grid
نقشه حرارت	Heat Map

Heuristic	ابتکاری
Logistics	لجستیک
Maze	مارپیچ

Multi-Agent Path Finding	مسیریابی چند عاملی
Node	گره

Abstract

This research investigates Multi-Agent Path Finding (MAPF) using Conflict-Based Search (CBS) as a problem-solving method. The primary objective is to design an efficient algorithm that facilitates multiple agents navigating through a defined maze while adhering to two crucial rules: avoiding collisions between agents and circumventing blocked paths. The proposed solution employs the A* algorithm to guide agents from their respective start points to designated end points. Unlike existing methods, which may suffer from inefficiencies or lack adaptability, the CBS approach offers a systematic method for resolving conflicts between agents' paths, ensuring smoother and more optimized navigation. Through extensive experimentation and analysis, this research demonstrates the efficacy of the proposed method in enhancing the efficiency of multi-agent pathfinding in comparison to conventional techniques. Overall, the results highlight the innovation and effectiveness of the CBS-based MAPF algorithm in addressing the complexities inherent in multi-agent navigation scenarios.

Implementation of Conflict-Based Search For Multi-Agent Optimal Routing

Electrical-Control Engineering bachelor's degree project report

Electrical Engineering, Control Systems

Department of Electrical Engineering

University of Tabriz

By:

Saeed Baninosrat

Supervisor:

Dr. Baradarannia

Winter 2024

[1] Sharon, G., Stern, R., Felner, A., Sturtevant, N.R., 2015. Conflict-based search for optimal multi-agent pathfinding. Artificial Intelligence 219

[2] Sharon, G., Stern R., Goldenberg M., Felner A., 2013. The increasing cost tree search for optimal multi-agent pathfinding. Artificial Intelligence 195