در این پروژه سعی داریم که پیاده‌سازی سه الگوریتم مختلف، دو الگوریتم ناآگاهانه و یک الگوریتم آگاهانه، بهنیه‌ترین مسیر برای رساندن کره به مشتری را پیدا کنیم. ربات باید از جایی که قرار دارد ابتدا به کره برسد و سپس کره را به سمت مشتری هل دهد. در سه بخش زیر به توضیح پیاده‌سازی این سه الگوریتم میپردازیم. مقایسه‌ی سه الگوریتم در انتهای گزارش قرار دارد.

- IDS

در الگوریتم IDS با در نظر گرفتن یک cutoff برای عمق قصد پیاده‌سازی الگوریتم DFS را داریم. به صورت کلی حالت iterative از الگوریتم DLS است. به این صورت که ابتدا تا عمق صفر به دنبال حالت هدف می‌گردیم و در صورت پیدا نکردن آن یکی هدف را زیاد می‌کنیم؛ این کار را آنقدر تکرار می‌کنیم تا یا به جواب برسیم و یا پاسخ شکست را برگرداند. این الگوریتم در کل می‌تواند ۴ جواب داشته باشد:

۱. Success: در صورت رسیدن به جواب
۲. Failure: در صورت نرسیدن به جواب
۳. Cutoff: در صورت رسیدن به محدودیت در نظر گرفته شده
۴. Duplicate: در صورت تکراری بودن شاخه و جواب

سه تابع اصلی برای این کد وجود دارد:

۱. SearchAlgorithm: الگوریتم IDS را به تعداد کره‌ها صدا می‌زند.
۲. IDS: در یک لوپ ، DLS را صدا می‌زند و با توجه به جوابی‌که برمیگرداند (یکی از جواب‌های گفته شده در بالا) ، تصمیم می‌گیرد آیا نیاز به زیاد کردن عمق است یا به هدف رسیده است یا به هدف نرسیده است.
۳. DLS: که به صورت بازگشتی به دنبال هدف می‌گردد و یکی از پاسخ‌های بالا را برمی‌گرداند.

چهار کلاس برای این کد وجود دارد:

۱. Main: که تابع‌ها در آن صدا زده‌شود و کنترل کد را بر عهده دارد.
۲. Node: کلاس node یا گره است که موقعیت ربات را نگه‌می‌دارد.
۳. Board: که اطلاعات مربوط به board و زمین بازی را در خود نگه‌می‌دارد.
۴. AIFunction: که الگوریتم‌های اصلی برای پیاده‌سازی کد در این کلاس قرار دارد – توضیحات بیشتر در کد ذکر شده است.

** کلاس دیگری برای interface در کد قرار دارد که فرصت برای کامل کردن آن نشد.

- Bidirectional BFS

در الگوریتم Bidirectional BFS، به این صورت عمل می‌کنیم که هم از طرف هدف و هم از طرف ربات به سمت هم حرکت می‌کنند تا در نقطه‌ای و مکانی به هم برسند. حرکت هر طرف به سمت هم با کمک الگوریتم BFS پیاده‌سازی می‌شود.

شش تابع اصلی این کد:

۱. searchAlgorithm: این تابع الگوریتم سرچ را پیاده سازی می‌کند. تا زمانی که BFS از سمت جلو و BFS از سمت عقب به یک node مشترک در fringe list شان برسند، سرچ را ادامه میدهد و در یک حلقه همواره ابتدا BFS به سمت جلو را اجرا میکند و بعد BFS به سمت عقب را اجرا میکند. اگر fringe list برای الگوریتم به سمت جلو و یا به سمت عقب خالی شود، یعنی جوابی نداریم و False برمیگرداند.
برای اجرای BFS رو به جلو ابتدا یک initial node ساخته میشود و به forward fringe list اضافه میشود(fringe list را یک صف در نظر گرفتیم). سپس هر بار در این جست و جو یک node از صف fringe، dequeue میشود و ابتدا تابع checkGoal روی آن صدا زده می‌شود. اگر هدف نبود، بسط داده میشود یعنی تابع successor صدا زده میشود و حالت هایی که میتوانند فرزندان این node باشند را تولید میکند، در ادامه هر یک از این حالت ها در تابع checkExplored بررسی میشوند و اگر قبلا آن حالت بررسی نشده بود، یک node برای آن ساخته میشود و به forward fringe list اضافه میشود.
در اجرای BFS رو به عقب هم همین مراحل تکرار میشوند با این تفاوت که به جای initial node یک final node داریم و به جای forward fringe list یک backward fringe list داریم و به جای تابع successor از تابع predecessor استفاده میشود.
۲. checkExplored: چک میکند آیا node جدیدی که میخواهد ساخته شود قبلا بررسی شده است یا نه
۳. successor: این تابع با دریافت یک node، حالت هایی که از آن node در جست و جو رو به جلو قابل دسترسی هستند و میتوانند فرزندان آن node باشند را برمیگرداند.
۴. predecessor: این تابع با دریافت یک node، حالت هایی که از آن node در جست و جو رو به عقب قابل دسترسی هستند و میتوانند فرزندان آن node باشند را برمیگرداند.
۵. checkGoal: اگر در اجرای جست و جوی forward صدا زده شود، چک میکند آیا node ورودی تابع، درbackward fringe list وجود دارد یا نه، اگر وجود داشته باشد یعنی دو الگوریتم جلو و عقب در این node به هم رسیده اند. اگر در اجرای جست و جوی backward صدا زده شود، چک میکند آیا node ورودی تابع، درforward fringe list وجود دارد یا نه.

۶. calcPathAndCost: تابع SearchAlgorithm در این تابع صدا زده میشود، و اگر جوابی پیدا شود، مسیر و هزینه آن را محاسبه میکند.

پنج کلاس برای این کد وجود دارد:

۱. Main: که تابع‌ها در آن صدا زده‌شود و کنترل کد را بر عهده دارد.
۲. Node: کلاس node یـا گره اســت که وضــعیت ربات و کره ها، cost، depth، parent، action را نگه می‌دارد.
۳. Table: که اطلاعات مربوط به زمین بازی را در خود نگه‌می‌دارد.
۴. BidirectionalBFS: که کدهای اصـلی برای پیاده‌سازی جست و جو دوطرفه در این کلاس قرار دارند.
۵. DrawBoard: این کلاس برای نمایش حرکت ربات به صورت گرافیکی است.

• A* گرافی

الگوریتم A* برخلاف دو الگوریتم قبـل، یـک الگوریتم آگاهانه اســت؛ یعنی در اینجا علاوه بر هزینه‌ای پرداختی تا مکانی که قرار داریم - g(n) – هزینه‌ی رسـیدن تا هدف را نیز در نظر می‌گیریم که به آن – h(n) –تابع شــهودی گفته می‌شــود. برای انتخاب یک node از تابع f(n) = h(n) + g(n) اســتفاده میشود.

شش تابع اصلی این کد:

۱. searchAlgorithm: این تابع الگوریتم ســرچ را پیاده ســازی می‌کند. تا زمانی که به nodeای که دارای حالت هدف باشد(همه کره ها به همه آدمها رسیده باشند) در یک حلقه همواره الگوریتم A* را اجرا میکند. اگر fringe list خالی شود، یعنی جوابی نداریم و False برمیگرداند.
ابتدا یک initial node ساخته میشود و به fringe list اضافه میشود. سپس هر بار در این جست و جو، لیسـت fringe براسـاس f(n) سـورت میشـود و node با کمترین هزینه تخمین زده شـده انتخاب میشـود و از لیسـت خارج میشـود. ابتدا تابع checkGoal روی آن صدا زده می‌شود. اگر هدف نبود، بسـط داده میشود یعنی تابع successor صدا زده میشود و حالت هایی که میتوانند فرزندان این node باشند را تولید میکند، در ادامه هر یک از این حالت ها در تابع checkExplored بررسـی میشـوند و اگر قبلا آن حالت بررسـی نشـده بود، یک node برای آن سـاخته میشود و به fringe list اضافه میشود.

۲. checkExplored: چک میکند آیا node جدیدی که میخواهد ساخته شود قبلا بررسی شده است یا نه
۳. successor: این تابع با دریافت یک node، حالت هایی که از آن node در جست و جو رو به جلو قابل دسترسی هستند و میتوانند فرزندان آن node باشند را برمیگرداند.
۴. checkGoal: چک میکند که آیا node ورودی تابع حالت نهایی و هدف است یا نه.
۵. calcPathAndCost: تابع SearchAlgorithm در این تابع صدا زده میشود، و اگر جوابی پیدا شود، مسیر و هزینه آن را محاسبه میکند.
۶. calcNodeEvaluation: این تابع مقدار h(n) را محاسبه میکند و آن را با مقدار g(n) جمع میکند و f(n) یک node را مشخص میکند.

پنج کلاس برای این کد وجود دارد:

۱. Main: که تابع‌ها در آن صدا زده‌شود و کنترل کد را بر عهده دارد.
۲. Node: کلاس node یا گره است که وضعیت ربات و کره ها، cost، depth، parent، action را نگه می‌دارد.
۳. Table: که اطلاعات مربوط به زمین بازی را در خود نگه‌می‌دارد.
۴. AStar: که کدهای اصلی برای پیاده‌سازی جست و جو در این کلاس قرار دارند.
۵. DrawBoard: این کلاس برای نمایش حرکت ربات به صورت گرافیکی است.

توضیحات محاسبه h(n):

در محاسبه‌ی h، همه‌ی فواصل را با استفاده از روش Manhattan distance محاسبه کردیم. ابتدا h=۰ است. بعد فاصله‌ی ربات تا نزدیکترین کره را پیدا می‌کنیم و h را با آن فاصله جمع زدیم. سپس فاصله‌ی کره ای را که به عنوان نزدیک ترین کره انتخاب کردیم تا نزدیک ترین آدم به آن کره محاسبه میکنیم و این فاصله را نیز با h جمع می‌زنیم.

بررسی قابل قبول بودن:

همانطور که در قسمت قبل توضیح داده شد از روش Manhattan distance برای محاسبه‌ی h استفاده می‌کنیم. در این روش چون مانع‌هایی مثل blocks – x و موانع دیگر مانند کره بر سر راه (حتما برای هل دادن کره به سمت مشتری باید در پشت کره قرار گرفت) در نظر گرفته نمی‌شود بنابراین مقدار محاسبه شده از مقدار واقعی کمتر خواهد بود. پس heuristic قابل قبول و سازگار است.

مقایسه‌ی سه الگوریتم به صورت زیر است.

با توجه به مثال زیر جدول پر شده است:

		5	6		
1	1	1	1	1	1
2	1	1b	1r	1b	2
2p	x	1	1	1	2
2	1	1	1	1	2
1	1	1	1	1	1p

	A*	Bidirectional BFS	IDS
زمان صرف شده	۲۳.۹۷ s	۱.۷۷ s	۲۷۷۶ ms
پیچیدگی زمانی	$O(b^d)$	$O(b^{d/2})$	$O(b^d)$
تعداد گره‌های تولید شده	۱۱۸۰۰	۳۷۴۳	۴۰۱۲
تعداد گره‌های گسترش داده شده	۷۳۰۰	۳۷۴۱	۴۰۰۹
عمق راه‌حل	۱۶	۱۴	۱۴

** مراحل انجام کار درباره‌ی پیاده‌سازی الگوریتم در Trello موجود است – لینک هر کدام در زیر قرار دارد:

IDS

Bidirectional - BFS