دانشکده‌ی مهندسی کامپیوتر
دانشکده‌ی مهندسی کامپیوتر
گروه آموزشی نرم‌افزار

عنوان گزارش:

گزارش فاز اول پروژه هوش مصنوعی

نام گروه:

الماس آبی

نام دانشجویان:

نیلوفر علیخانی-۹۷۳۶۱۳۰۵۰

رومینا رئوفیان-۹۷۳۶۱۳۰۲۷

گرایش:

نرم افزار

نام استاد:

دکتر حسین کارشناس

۱۵ آبان ۱۴۰۰

فهرست مطالب

فصل ۱

توضیح پیاده‌سازی پروژه

۱-۱ فضایی حالت مسأله تعیین راهبرد جمع‌آوری الماس‌ها

۱-۱-۱ استیت

استیت‌های فضای حالت مسأله‌ی تعیین راهبرد جمع‌آوری الماس‌ها به فرم زیر است.

- مکانی که عامل در آن قرار دارد
- مکان و سطح الماس‌های برداشته شده (کشنری visited الماس‌ها)
- نوع وسطح (level) استفاده‌ی عامل از سیاه‌چاله برای تله‌پورت (دیکشنری visited سیاه‌چاله‌ها)
- امتیاز عامل
- سطحی از درخت که عامل در آن قرار دارد
- تعداد حرکات باقی‌مانده
- نوع و تعداد الماس‌های برداشته شده توسط عامل

درخت جستجوی سطح اول شامل استیت‌هایی با شرایط بالا هستند.

۱-۱-۲ استیت اولیه

استیت اولیه‌ی درخت جستجوی سطح اول به‌صورت زیر تعریف می‌شود.

- مکان اولیه‌ی عامل؛ مکانی در نقشه‌ی بازی که در آن کاراکتر A وجود دارد.
- دیکشنری خالی که برای اضافه کردن مکان و سطح الماس‌های برداشته شده است.
- دیکشنری خالی که برای اضافه کردن مکان و سطح سیاه‌چاله‌هایی که عامل در آن‌ها تله‌پورت انجام داده است.
- امتیاز اولیه داده شده به عامل
- سطح صفر
- ماکسیمم حرکت داده شده به عامل
- دیکشنری هر نوع الماس با تعداد صفر

۱-۱-۳ کنش‌ها

برای جابه‌جایی بین استیت‌های موجود در درخت جستجوی سطح اول کنش‌های زیر را تعریف کرده‌ایم.

- عامل می‌تواند با داشتن شرایط لازم [1] برای برداشتن یک الماس استیت خود را تغییر دهد.
- عامل می‌تواند با داشتن شرایط لازم بر روی سیاه‌چاله برود و عمل تله‌پورت را انجام دهد.

۱-۱-۴ حالت پایانی

با توجه به اینکه ما یک هدف مشخص نداریم و مساله‌ی ما بهینه‌سازی است برای چنین مسائلی حالت هدف نمی‌توان تعریف کرد اما می‌توان حالت‌های پایانی برای الگوریتم در نظر گرفت.
حالت‌های پایانی درخت جستجو در سطح یک به‌صورت زیر است.

[1] شرایط لازم در قسمت الگوریتم این سطح (سطح اول) توضیح داده شده است.

- تمام الماس‌های داخل نقشه توسط عامل برداشته شده باشند.
- تعداد حرکت‌های باقی مانده‌ی عامل به صفر رسیده باشد.
- سطحی که عامل در آن قرار دارد برابر با عمق تعیین شده باشد.[۲]

۱-۱-۵ تابع هزینه‌ی مسیر

برای تغییر استیت که در این سطح به معنای برداشتن یک الماس یا رسیدن به سیاه‌چاله، عامل نیاز دارد تا مسیری را طی کند که برای بدست آوردن فاصله‌ی این مسیر چند گزینه داشتیم مانند محاسبه‌ی فاصله‌ی منهتنی. اما به دلیل وجود دیوار در نقشه‌ی بازی، برای به دست آوردن فاصله‌ی دقیق و ارزش گذاری مناسب از الگوریتم Dijkstra استفاده کردیم.

۱-۲ الگوریتم مسأله تعیین راهبرد جمع‌آوری الماس‌ها

برای الگوریتم این سطح ما یک dfs با عمق محدود پیاده‌سازی کردیم. برای پیاده‌سازی dfs از الگوریتم andor کمک گرفتیم به این صورت که هر یک از الماس‌ها را مانند گره‌های or و هر یک از سیاه‌چاله‌ها را تا حدی مانند گره‌های and گرفتیم.

در ابتدا ما با گرفتن نقشه و کاراکتر عامل و دانستن کاراکتر الماس‌ها و سیاه‌چاله یک لیست از تاپل‌ها که مکان الماس و امتیاز هر الماس پر کردیم و لیستی دیگر به همین صورت برای سیاه‌چاله پر کردیم، همچنین مکانی که عامل قرار دارد را نیز پیدا کردیم.

یک دیکشنری به نام diccolor-numberبرای نگه داشتن تعداد الماس‌های از هر نوع که تا مرحله مورد نظر برداشته‌ایم، تعریف می‌کنیم.

برای تکمیل اطلاعات برای فراخوانی تابع dfs نیاز به تعیین عمق مورد نظر داریم.

$$\text{depth} = \text{floor}(\log((10 ** 4) * \text{timelimit}, \max(\text{sizedh}, 2)))$$

از آن جایی که cpu در هر کلاک به تعداد10^6عملیات انجام می‌دهد، با توجه به اینکه در الگوریتم dfs

[۲] نحوه‌ی محاسبه‌ی عمق در قسمت الگوریتم بیان شده است.

ما برای بدست آوردن فاصله به یک تابع dijkstra در سطح اول نیاز داریم و همچنین یک dijkstra برای مسیریابی در سطح دوم نیز داریم با آزمون و خطا به این نتیجه رسیدیم که برای تعیین عمق dfs تا 10^4عملیات را در محدودیت زمانی داده شده قرار دهیم. برای اینکه در هر سطح به اندازه‌ی تعداد الماس‌ها و سیاه‌چاله‌ها در درخت جستجو گره جدید ایجاد می‌کند در نتیجه عمق مورد نظر از فرمول بالا بدست آوردیم.

با توجه به اینکه اگر نسبت دیوارها به اندازه‌ی نقشه بازی از یک حدی کمتر بود در dijkstra سطح اول تعداد گره‌هایی که باید بررسی کند، زیاد می‌شود عمقی که با فرمول بالا بدست می‌اید زیاد است به همین خاطر با آزمون و خطا شرط زیر را برای بدست اوردن عمق اضافه می‌کنیم.

$$\text{if } (\text{walls} \,//\, (\text{height} + \text{width})) * 100 < 5 \text{ and } \text{len}(\text{hole}) == 0\text{: depth -= 1}$$

در تابع dfs ابتدا دو دیکشنری visited یکی برای الماس‌ها و دیگری برای سیاه‌چاله‌ها تعریف کردیم. کلید دیکشنری visited الماس‌ها به صورت تاپل دوتایی مکان آن الماس است که در آن مقدار تاپل true وlevel که آن الماس در این dfs بررسی شده است ذخیره می‌شود؛ کلید دیکشنری visited سیاه‌چاله‌ها به صورت یک تاپل سه‌تایی که مکان آن سیاه‌چاله و level است و در آن مقدار تاپل true و level قرار می‌گیرد. در کلید دیکشنری سیاه‌چاله‌ها به این دلیل level را اضافه کردیم که اگر عامل در یک حرکت به داخل سیاه‌چاله رفت، در حرکت‌های بعدی نیز بتواند به داخل آن سیاه‌چاله برود.

در هر تکرار dfs اگر سطح به عمق مورد نظر نرسیده است روی تمامی الماس‌های موجود و بعد از آن سیاه‌چاله‌هابرای بررسی آن‌ها for می‌زنیم. در for الماس‌ها اگر الماسی تا آن سطح بررسی نشده بود(not visited) برای بدست آوردن فاصله‌ی الماس نسبت به عامل تابع dijkstra سطح اول را فراخوانی می‌کنیم. برای اینکه حرکات باقی‌مانده در ارزشی که برای هر مسیر بدست می‌آوریم اهمیت زیادی دارد، برای بدست آوردن فاصله از الگوریتم dijkstra استفاده کردیم تا فاصله واقعی را به ما بدهد. در این الگوریتم عامل می‌تواند به چهار سمت بالا،پایین،چت و راست در صورتی که قابلیت حرکت در آن سمت را داشته یاشد؛ برود.

حالا اگر فاصله‌ی بدست آمده کمتر از تعداد حرکات که می‌توانیم انجام دهیم، بود با بررسی یک سری شرط تابع dfs را به صورت بازگشتی صدا می‌زنیم. قابل ذکر است که قبل از فراخوانی مجدد تابع،در دیکشنری diccolor-number تعداد الماس آن نوع را یکی اضافه کرده و بعد از فراخوانی باز یکی کم می‌کنیم.

شرط‌های مورد نیاز برای انتخاب آن الماس و فراخوانی مجدد تابع dfs با توجه به قوانین بازی شامل موارد زیر است:

- تعداد الماس‌های برداشته شده از آن نوع به حدنصاب نرسیده باشد.
- حداقل امتیاز مورد نیاز برای برداشتن آن نوع الماس را داشته باشد، باتوجه به اینکه به ازای هر حرکت یک امتیاز از عامل کم می‌شود.

در هر فراخوانی تابع dfs ما مقدار ،level مکان عامل، تعداد حرکات باقی‌مانده، امتیاز عامل تا آن مکان را به صورت بروزرسانی شده به تابع ارسال می‌کنیم. در هر دور بهترین مقدار بازگشتی از تابع را نگه می‌داریم و در اخر آن الماس را از دیکشنری visited خارج می‌کنیم.
در for سیاه‌چاله‌ها یک سیاه‌چاله را انتخاب می‌کنیم و فاصله‌ی آن را تا عامل از طریق dijkstra سطح اول بدست می‌آوریم. اگر عامل تعداد حرکت کافی برای رسیدن به آن سیاه‌چاله را داشت به ازای تمامی سیاه‌چاله‌های دیگر که احتمال دارد عامل از آن بیرون بیاید، تابع dfs را مجدد فراخوانی می‌کنیم. مقدارهای بازگشتی برای هر یک از سیاه‌چاله‌های که بررسی کردیم را باهم جمع می‌کنیم و در نهایت میانگین این مقدار را برای سیاه‌چاله‌ای که در ابتدا انتخاب کردیم در نظر می‌گیریم.
در هر فراخوانی تابع dfs مقادیر ،level مکان عامل، تعداد حرکات باقی‌مانده، امتیاز عامل بروزرسانی شده را به تابع ارسال می‌کنیم. در هر دور بهترین مقدار بازگشتی از تابع را نگه می‌داریم و در اخر آن سیاه‌چاله را از دیکشنری visited مربوط به سیاه‌چاله‌ها خارج می‌کنیم.
در این تابع به هر یک از مسیرهای که پیداشده یک ارزشی نسبت می‌دهیم، به دلیل اینکه هم تعداد حرکات باقی‌مانده و امتیاز الماس‌های بدست‌آورده برای ما اهمیت دارد، در نتیجه برای محاسبه این ارزش لازم است هر دوی آن‌ها را در نظر بگیریم. برای این کار ما بر هر یک از این دو یک وزنی نسبت می‌دهیم که این وزن با آزمون و خطا بدست آمد. همچنین به این نتیجه رسیدیم که اگر در مسیر الماسی بدست نیاورده بود (امتیاز بدست‌آمده برابر صفر می‌شود) بهتر است به جای استفاده از ارزش ذکر شده در بالا وزن کمتری به تعداد حرکات باقی‌مانده نسبت دهیم.
در تابع dfs که پیاده‌سازی کردیم سه حالت base case داریم

۱. اگر که مقدار سطح (level) که درعامل در آن قرار دارد برابر با مقدار depth حساب شده باشد، در این صورت اگر الماسی را در این مسیر مشاهده نکرده باشیم ازرش این راه به صورت زیر حساب می‌شود:

value = remain-turn * 50 // 100

در غیر این‌صورت ارزش این مسیر به صورت زیر حساب می‌شود:

value = (((20*(score-agent - current-score)) + (80*remain-turn)) //100)

اگر ارزش حساب شده در این مسیر بیشتر از ماکسیمم ارزشی که تا الان بدست‌آمده باشد، ارزش بدست آمده را به عنوان ماکسیمم ارزش بدست‌آمده تا الان قرار داده و همچنین روی دیکشنری visited الماس‌ها و سیاه‌چاله‌ها for زده و هر کدام که شانlevel برابر صفر بود که این بدین معنی است که مسیر پیدا شده با ماکسیمم ارزش تا الان از این الماس یا سیاه‌چاله شروع شده است پس آن را به عنوان حرکت بعدی انتخاب می‌کنیم.

۲. اگر که تعداد حرکات باقی مانده برای عامل صفر شود. ارزش این مسیر مانند حالت بالا حساب می‌شود.اگر ارزش حساب شده ماکسیمم بود مانند بالا آن را به عنوان ماکسیمم ارزش حساب شده تا الان قرار داده و حرکت بعدی عامل را همانند بالا بدست می‌آوریم.

۳. اگر تمامی الماس‌های موجود در نقشه بازی در دیکشنری visited مربوط به الماس‌ها موجود باشند. اگر ارزش حساب شده ماکسیمم بود مانند بالا آن را به عنوان ماکسیمم ارزش حساب شده تا الان قرار داده و حرکت بعدی عامل را همانند بالا بدست می‌آوریم.

در اخر هم حرکت بعدی که ماکسیمم ارزش را در کل مسیرهای پیداشده داشته را برمی‌گردانیم. اگر حرکتی که برگردانده شده است یک تاپل خالی بود به این معناست که عامل با تعداد حرکت باقی‌مانده نمی‌تواند به الماس یا سیاه‌چاله‌ای برسد که در این شرایط ما کنش no operation را قرار می‌دهیم اما اگر یک الماس را برگرداند امتیاز عامل و همچنین دیکشنری diccolor-number را بروزرسانی می‌کنیم و حرکت مورد نظر را به تابع جستجو سطع دوم برای انجام ادامه عملیات ارسال می‌کنیم.

۱-۳ فضای حالت مسأله مسیریابی

۱-۳-۱ استیت

استیت‌های فضای حالت مسأله‌ی مسیریابی به فرم زیر است.

- مکان عامل
- مکانی که عامل می‌خواهد به آن برود
- امتیاز عامل در آن خانه

۱-۳-۲ استیت اولیه

- مکانی که عامل در آن قرار دارد
- مکانی که عامل می‌خواهد به آن برود
- امتیازی که عامل در این مکان دارد

۱-۳-۳ کنش‌ها

عامل می‌تواند در صورت داشتن شرایط لازم در این سطح کنش‌های زیر را انجام دهد.

- حرکت به چپ
- حرکت به راست
- حرکت به بالا
- حرکت به پایین
- تله‌پورت از طریق سیاه‌چاله

۱-۳-۴ حالت پایانی

در این سطح دو حالت پایانی تعریف می‌شود.

- اگر مکان عامل و مکانی که می‌خواهد به آن برود یکی باشد، به معنای آن است که عامل می‌خواهد کنش تله‌پورت را انجام دهد.
- هنگامی که مکانی که عامل در آن قرار گرفت برابر با مکانی بشود که عامل می‌خواست به آن برود.

۱-۳-۵ تابع هزینه‌ی مسیر

در این سطح به دلیل اینکه عامل می‌تواند در هر جابه‌جایی به سمت بالا، پایین، چپ و راست برود یا عمل تله‌پورت را انجام بدهد، هزینه‌ی هر یک از جابه‌جایی‌ها برابر با یک واحد است.

۱-۴ الگوریتم مسأله مسیریابی

با توجه به مقدار مکان بعدی که در الگوریتم سطح اول بدست اوردیم و نگه داشتن مکانی که عامل وجود دارد در این سطح ما یک الگوریتم dijkstra پیاده‌سازی کردیم.
در ابتدا یک دیکشنری visited برای نگه‌داری مکان‌های مشاهده شده نقشه بازی در حین مسیریابی در نظر می‌گیریم، کلیدهای این دیکشنری شامل تاپل‌های دوتایی از هایindex مکان موجود در نقشه است که مقدار آن را برابر با true قرار می‌دهیم. همچنین برای نگه داشتن جهت‌های پیمایش شده یک دیکشنری parent در نظر می‌گیریم که کلید آن شامل تاپلی از هایindex مکان‌های موجود در نقشه و مقدار هر کدام تاپلی سه تایی از index خانه پدر و جهتی که با آن از مکان پدر به مکان فرزند رفته‌ایم، قرار می‌دهیم.
اگر مکانی که عامل در آن قرار دارد و مکانی که می‌خواهد به آن برود یکسان بود به این معنی است که عامل روی سیاه‌چاله قرار داشته و می‌خواهد کنش تله‌پورت انجام دهد پس کنش تله‌پورت را برمی‌گردانیم.
در غیر این صورت مسیریابی را برای عامل انجام می‌دهیم.
ما از ساختمان داده صف اولویت برای dijkstra استفاده می‌کنیم که در آن مقدار فاصله بدست آمده برای آن خانه تا آن لحظه و هایindex آن خانه را نگه داری می‌کنیم. همچنین یک دیکشنری distance داریم

که در این دیکشنری مینیمم فاصله‌ای که تا آن خانه وجود دارد را نگه‌داری می‌کنیم. ما مسیریابی را تا زمانی ادامه می‌دهیم که کلیدی با هایindex مربوط به خانه مقصد در آن وارد شود.
در هر خانه می‌تواند به چهار جهت بالا، پایین، چپ و راست در صورتی که شرایط زیر برقرار باشد، برود.

- خانه‌ای که می‌خواهیم برویم قبلا مشاهده نشده باشد.
- در صورت حرکت در جهت خاص از نقشه بازی بیرون نزنیم.
- خانه‌ای که در آن می‌خواهیم برویم خالی یا سیاه‌چاله باشد.
- اگر در خانه‌ای که می‌خواهیم برویم الماس وجود داشت و این خانه مقصد ما نبود باید چک کنیم که با توجه به امتیاز عامل یا تعداد الماس‌هایی که از آن نوع برداشتیم نتوانیم آن الماس را برداریم و فقط از آن عبور کنیم. (به دلیل اینکه ما در الگوریتم سطح اول بهترین حرکت بعدی برای عامل را پیدا کردیم در صورتی که در حال حرکت به سمت مقصد (بهترین خانه بعدی برای عامل) هستیم، نمی‌خواهیم الماس دیگری را برداریم.)

به دلیل اینکه به ازای هر کنش یک امتیاز از عامل کم می‌شود، برای آنکه شرط مقایسه با مقدار امتیازی که عامل در حال حاضر دارد و حداقل امتیازی که برای برداشتن هر نوع الماس نیاز دارد، درست اعمال شود. ما نیز به ازای حرکت به خانه انتخاب شده یک واحد از امتیاز عامل کم می‌کنیم. قابل ذکر است که شرایطی که در بالا ذکر شد در تابع dijkstra سطح اول هم اعمال شده است.
اگر خانه‌ای که با توجه به شرایط بالا می‌توانیم به آن برویم در دیکشنری distance نبود آن را به دیکشنری distance با مقدارفاصله‌ی پدر از مکان اولیه عامل به علاوه یک اضافه می‌کنیم و به صف اولودیت و دیکشنری parent مقدارهای مورد نظر را اضافه می‌کنیم. اما اگر خانه در دیکشنری distance وجود داشت اگر که مقدار فاصله‌ی پدر از مکان اولیه عامل به علاوه یک کمتر از مقدار distance موجود بود دیکشنری distance ، parent و صف اولویت را با مقدار جدید بروزرسانی می‌کنیم.

اگر هایindex خانه‌ای که از صف اولویت خارج می‌شود برابر با هایindex خانه مقصد باشد به این معنی است که مسیریابی به پایان رسیده و حالا باید کنش‌هایی که با انجام آن‌ها به خانه مقصد رسیده‌ایم را

بدست آوریم.

برای نگه‌داری مسیر طی شده از مکان اولیه تا رسید به مقصد از یک ساختمان داده LifoQueue که مانند استک عمل می‌کند با نام way استفاده می‌کنیم. اگر خانه‌ی مقصد سیاه‌چاله باشد به این معنی است که عامل می‌خواهد کنش تله‌پورت را انجام دهد؛ بنابراین ما در ابتدا در way کنش تله‌پورت را اضافه می‌کنیم. در یک وایل از خانه مقصد شروع کرده و از طریق دیکشنری parent کنش مربوط به آن خانه را به way افزوده و به خانه پدرش می‌رویم هنگامی که به مکان اولیه عامل رسیدیم وایل به پایان می‌رسد و way را برمی‌گردانیم.

۵-۱ نحوه‌ی لحاظ کردن محدودیت زمان تصمیم گیری برای عامل

با توجه به اینکه محیط (نقشه‌ی بازی) پویا نیست، در ابتدا ما یک لیست از سیاه‌چاله‌ها و الماس‌ها با بررسی کامل نقشه‌ی بازی پر می‌کنیم که در آن مکان هر الماس یا سیاه‌چاله و امتیاز آن را قرار می‌دهیم و تا اخر بازی از این لیست‌ها استفاده می‌کنیم که این کار باعث می‌شود تا در نوبت نیاز به بررسی کل نقشه‌ی بازی نداشته باشیم و هر بار که الماسی توسط عامل برداشته می‌شود، آن الماس را از لیست الماس‌ها خارج می‌کنیم.

ما اگر در تابع dfs به صورت کامل تمامی درخت جستجو را پیمایش می‌کردیم عامل بهترین مسیر ممکن را برای جمع‌آوری الماس‌ها انتخاب می‌کرد اما با توجه به محدودیت زمانی تعریف شده این کار ممکن نبود به همین دلیل برای آنکه در بازه‌ی زمانی مناسب عامل بهترین حرکت ممکن را پیدا کند از dfs با عمق محدود استفاده کردیم و برای هر مسیر پیدا شده ارزشی قرار دادیم تا باتوجه به این ارزش‌ها عامل بهترین حرکت بعدی خود را در تایم قابل قبول انتخاب کند.

مقدار عمق با در نظر گرفتن تعداد عملیاتی که cpu در هر ثانیه می‌تواند انجام دهد و تعداد گره‌های باز شده در هر سطح بدست می‌آید.

تعداد گره‌هایی که در هر سطح باز می‌شود برابر با تعداد الماس‌ها و سیاه‌چاله‌ها است.

تعداد عملیاتی که cpu در هر ثانیه انجام می‌دهد 10^6 است. البته ما چون در هر نوبت علاوه بر dfs به دو تابع dijkstra در سطح اول ودوم نیاز داریم برای بدست آوردن عمق مناسب با توجه به محدودیت زمانی

با آزمون و خطا به این نتیجه رسیدیم که مقدار 10^4 در محدودیت زمانی داده شده برای تعداد عملیات ممکن در dfs قرار دهیم. در نتیجه عمق را از طریق فرمول زیر بدست می‌آوریم.

```
depth = floor(log((10 ** 4) * timelimit, max(sizedh, 2)))
```

اگر که در نقشه‌ی بازی نسبت تعداد دیوارها به ابعاد زمین بازی از یک حدی کمتر باشد باعث می‌شود در تابع ایdijkstra که در سطح اول برای بدست اوردن فاصله حقیقی مکان عامل با الماس یا سیاه‌چاله استفاده می‌شود تعداد خانه‌های زیادی برای پیمایش وجود داشته باشد در نتیجه برای بررسی درخت جستجو تا آن عمق که در بالا محاسبه شده به زمان بیشتری نیاز دارد پس ما برای عمق شرط زیر را نیز اضافه می‌کنیم.

```
if (walls // (height + width)) * 100 < 5 and len(hole) == 0: depth -= 1
```

فصل ۲

نقش افراد گروه

در ابتدا مدتی برای فهم مساله و تعیین راه حل و انتخاب نوع الگوریتم‌های مورد نیاز با یکدیگر صحبت کردیم. برای تعیین راهبرد مسأله جلساتی در اسکایپ گذاشتیم و برای راحت‌تر شدن انتقال ایده‌ها از جم‌بورد استفاده کردیم.

در این جلسات ما علاوه‌بر بررسی الگوریتم‌هایی که برای حل این مسأله پیشنهاد می‌کردیم، به محاسبه‌ی مدت زمان مورد نیاز این الگوریتم‌ها برای حل مسأله با توجه به نقشه و شرایط موجود در بازی می‌پرداختیم. با توجه به محاسباتی که در این جلسات انجام دادیم به این نتیجه رسیدیم که به جای تشکیل کامل درخت جستجو برای به‌دست آوردن جواب مسأله (ترتیب جمع‌آوری الماس‌ها)از dfs با عمق محدود به همراه ارزش‌گذاری برای هر مسیر استفاده کنیم و همچنین برای مسیریابی عامل در نقشه‌ی بازی از الگوریتم dijkstra استفاده کنیم.

برای پیاده‌سازی پروژه ابتدا یک ریپازیتوری در گیت‌هاب درست کردیم. برای افزایش دقت و سرعت و تسلط کافی به قسمت‌های مختلف پروژه ، پیاده‌سازی آن را به صورت pair programming انجام دادیم، به گونه‌ی که درحین پیاده‌سازی هر تابع در اسکایپ صفحه خود را share کرده؛ ابتدا برای نوشتن آن تابع درباره‌ی اسکلت کلی کد و شرط‌های آن با یکدیگر صحبت می‌کردیم و به صورت نوبتی توابع را پیاده‌سازی کردیم. این کار در قسمت دیباگ کردن کد بسیار کمک کننده بود.

فصل ۳

تعهدنامه اخلاقی

ما (نیلوفرعلیخانی و رومینا رئوفیان) تعهد می‌نماییم که پروژه تحویل داده شده نتیجه کار ما بوده و در هیچ یک از بخش‌های انجام شده از کار گران کپی برداری نشده است. در صورتی که مشخص شود که این پروژه کار ما نبوده است، طبق ضوابط آموزشی با ما برخورد شده و حق اعتراض نخواهیم داشت.