ROBOCUP Soccer Simulator2D Research Project

Summer 2022, Final Report

Melissa Katz, under the supervision of Joseph Vibyhal

COMP 396

School of Computer Science
Faculty of Science
Mcgill University

June 25^{th}, 2022

Abstract

In order to participate in the advancement of AI capabilities, proving that it can solve more and more complex problems, this project aimed to create the basis of an AI that could play soccer using the RoboCup Soccer Simulator 2D[1], opening a way for McGill to participate in future competitions. By implementing a Goal Oriented Action Planning system, it was able to create a goalkeeper that can successfully protect the goal and a player that can score goals and respect the rules of the simulation. With the system currently in place, future researchers will be able to implement new functionality in a very fast manner.

1. Introduction

The RoboCup competition is a way for computer scientists to break barriers in the field of Artificial Intelligence, as it was done before with Go and Chess. The goal is for teams of robots to compete in a game of soccer, until eventually the technology is good enough for an artificial team to win against a team of humans. For now however, we will use RoboCup Soccer Simulator 2D, which provides a fully simulated environment for the players, allowing us to focus entirely on the AI without the need to additionally worry about the mechanical aspects of the task. Previous work on this project allowed us to correctly send and receive information to and from the server. Now, using Goal Oriented Action Planning, we have successfully implemented a rudimentary AI that can devise a plan that will result in a goal being scored.

2. The Simulator

The code that our agents run obviously relies on the RoboCup Soccer Simulator. It is necessary to download the Server and the Monitor from the project's GitHub page[1]. Installation differs slightly depending on the Operating System. Linux and MacOS have more straightforward instructions: download the two tar files related to the server and monitor, install any dependencies using the command-line, then *configure* and *make* the project (for more detailed instructions, you can refer to our README[4] and the official documentation[2][3][1], section 3). For Windows, follow the tutorial made by Carlton University[8]. Do note that at the time of writing, the latest version of the server available for Windows is 15.0.0 while the Linux and MacOS versions are already up to 17.0.1.
After the server and monitor are successfully installed, you can run any Java code with a main method that creates a Player, and you should see the player appear on the the monitor, along with a "player connected" message on the command-line. Finally, to start the game, simply click on the "Kick Off" option in the "Referee" menu of the monitor, or hit Ctrl+K. The game will then run for 3000 cycles until half-time, when it will wait for a kick-off again, and then for an extra 3000 cycles (or until a goal is scored if after these 3000 cycles the game ends ups tied).

1. The two manuals listed are quite similar, both having their strengths and weaknesses. The one on readthedocs has a very convenient look-up system, but has missing sections. The one on github has a bit more information on some functionality, as well as higher quality pictures. However, I was sometimes unable to find information in either manual. In any case, I provide both for convenience.

3. Previous Work

The first batch of work done on this project[9] consisted of creating all the structure needed for players to interact with the server. In that original configuration, the Player class was used to start communications with the server, and would hold a reference to the FromServer class that would allow to capture any visual information from the server and parse it. This parsed information would be transmitted to an instance of the GlobalMap class, which would then store this information and use it to update an illustation of what a Player sees (if that Player had the appropriate option enabled). It had some basic movement options for the player, but they were all controlled from the command-line.

In this first stage, multiple things were missing: first, the server also sends hear and body sensor commands, both of which carry valuable information about the state of the game. Next, some decisions were left up to future developers: some placeholder classes with no functionality were created, like the positionEst class, while other chunks of code had multiple versions available, such as FromServer and FromServer2, which both get information from the server, with the first running in a separate thread while the second is non-threaded. Some functionality was needed only to show off communications with the server, like the option to see what players saw in their field of view, or the ability to interact with players from the command-line. And, of course, there was no AI in the code.

Outside of the small tweaks that the code needed, my main objective was to implement two AI's for the players, one for the goalkeeper and one for all other players. My code had two objectives: I wanted a way for the goalkeeper to guard its goal and catch any ball that gets too close, and I needed the general player to run to the ball and kick it towards the goal. All players also had to respect any hear commands from the server (for example, when a goal is scored, the players must return to their starting positions). These two situations were tested separately. First, the goalkeeper is tested by setting a dummy player (a player with minimal functionality) to kick the ball towards the goal, and checking whether the goalkeeper will successfully catch it. Second, two teams with three general players, without goalkeepers, were set to play against each other. These matches do not feature any tactical behaviour from the players, but they do show off the player's ball-seeking behaviour and their adherence to the simulator's rules.

4. Implementation of GOAP

The AI described above was implemented using Goal-Oriented Action Planning, or GOAP for short. GOAP is a system that removes the need for complex finite state machines in planning, letting the planner create a way to reach the goal on the fly. Each player is given a set of goals it should fulfill as well as a set of actions it can execute. Whenever the agent selects a goal (by figuring out what is the priority in its current state), it passes this goal to a planner which returns the sequence of actions to execute. The main advantage of this approach is that the code can be very easily expanded upon: adding a new goal means giving it any conditions it needs, and adding actions means giving then some effects and preconditions along with a way to execute it. Any new additions to the available goals or

actions of a player will immediately be taken into account next time the code will be run.

I will briefly present the classes related to GOAP that exist in this project. My implementation is based on descriptions from papers[5][6], but also on an already existing implementation [7].

4.1 StateKeys

StateKeys is an enum that functions mostly as a way to prevent typos: throughout the code, HashMaps are used to describe the state of the world by associating a condition (something known about the state of the world, such as the ball being in view of a certain player), and a boolean value. However, leaving the keys as Strings is dangerous since typos could easily become the cause of tedious debugging sessions. Enums impose a spelling, eliminating this problem.

4.2 GOAPAction

This is an abstract class used for defining what as Action is. It uses two HashMaps[2] to hold the preconditions and the effects of the action. It also provides some basic getters and setters for those actions and values. This class has 3 abstract methods: resetActionSpecifics() provides a way to reset any parameters exclusive to some action. checkProceduralPrecondition() allows for us to check for a precondition that cannot be simply described by a key-value pair, and it generally implies searching for a valid target of an action. For example, this could be useful for a pass action, where the procedural precondition would involve looking for the closest player our agent could pass a ball to. It can also be used if the action can be disqualified immediately depending on the circumstances (for example, there is no need to consider any actions related to teleporting when the move command is not allowed), which means that there are less actions to consider during planning. Finally, executeAction() must contain the code to be run whenever this action gets scheduled. It returns either true or false, depending on whether the action was completed successfully.

The following classes currently extend GOAPAction: GetBallInFOVAction, which makes the player turn until the ball is found in its field of view; CenterBallInFOVAction, which makes the player look directly at the ball; CatchBallAction, which prepares the agent for a catch; BringBallToGoalAction, which kicks the ball towards the goal with a light kick (this allows the player to basically run alongside the ball), KickBallToGoalAction, which kicks the ball towards the goal with a strong kick; MoveToBallAction, which makes the player dash towards the ball; and TeleportSelfAction, which allows the player to teleport back to its start position.

4.3 GOAPGoal

GOAPGoal is an abstract class for defining Goals that an agent would want to achieve. These should be as broad as possible to hopefully leave our agent some planning freedom, which should lead to more complex behaviour. Anything extending this class has a

2. Unless otherwise specified, all HashMaps use StateKeys as keys and Booleans as values.

HashMap containing the state we wish to reach. The abstract functions are calculatePriority(), which either returns the same number every single time or performs some calculation if the priority changes with time, as well as validitySpecifics(), partially determines whether the goal should be considered given the current context. To determine the true validity of the goal, the function isValid() should be used, which allows us to rule out any goals that are already satisfied. GOAPGoal also implements the Comparable interface, using priority to determine comparisons.

There are currently four classes extending this one: FocusOnBallGoal, which results in the player looking at the ball; InterceptBallGoal, which should result in the player acquiring the ball; KickBallGoal, which makes the player kick the ball, although the direction and strength of the kick is determined by the actions; and finally, ResetPositionsGoal, which results in the player moving to some new position unrelated to the ball.

4.4 GOAPNode

GOAPNode provides a way to wrap actions using a node in order to run the A* search algorithm through them and find a way to satisfy the goal. For A* to work, we need a move function g(n) and a heuristic function h(n), the sum of which create the cost function, f(n). Currently, the move function g(n) is just the number of nodes from the root. h(n) on the other hand, uses the value of an action, which is determined based on how many of the action's effects can be used to work towards the goal (this percentage is inverted in order to have the actions that have a higher percentage be at the start of the priority queue). For now this seems to be reasonable, but experimenting with other heuristics will be an important step in the future as the project develops. This class also contains a method to retrieve the path found from the goal back to the start state using the parent attribute of each node.

4.5 GOAPPlanner

This class contains a single method, plan(), which returns a linked list containing the actions we should take in order to reach a given goal. This function first filters out any actions that should not be considered right now, depending on whether or not their procedural preconditions are satisfied, then runs A* on this set, and finally retrieves the path to the goal.

4.6 GOAPAgent

GOAPAgent is an interface defining some methods needed for the players to interact with the classes above, the most important of which is planAndExecute(), which should be used to allow the agent to create a plan and then execute it. The other two functions, getNewGoal() and goalStillValid(), should be used to deal with a goal preempting the execution of the current goal.

5. Current State of the code

In order to integrate GOAP, as well as fix the small issues outlined earlier, the previous code had to be reworked. First, output from the server will be dealt with by FromServer2 instead of FromServer. This is because FromServer, the thread version, created synchronisation problems which caused the players to behave incorrectly. Even if these issues were resolved, it would still mean that one player would need two threads to function properly (one that does the planning and sends commands to the server, and one that receives updates from the server), which translates to one team of 11 players needing 22 threads, which is quite costly time-wise. On top of that, since the server sends commands at the start of every cycle, and since the function that reads from the server blocks if its buffer is blank, that means the non-threaded version provides a simple way to synchronize our agent with the cycles of the simulation. Next, I've expanded the LocalView class to make it capable of parsing through hear and body sensor messages, and made it possible for FromServer2 to recognize these new messages. Depending on the message, LocalView will use a different way of converting the String into a list of Tuples.

These changes lead to the need to slightly re-purpose the GlobalMap class; it is no longer able to paint the point of view of the player (that functionality was removed entirely from the class). However, it now takes care of building the state HashMap that is used by the GOAP system, since the agent can only know that the world changed if it is able to observe these changes. For example, while iterating through Tuples that representing visual information, if it sees the label "b", it will know that the ball is in its field of view, and will thus set the value of the key "ball_in_FOV" to true. It will then analyze its distance to the ball and will set the value of "has_ball" to true if the ball is close enough. Similar manipulations can be made with any information sent by the server.

Another important change happened in the Player class: It no longer contains all of the functionality needed to correctly use an agent in the server, and instead shares this responsibility primarily with three other classes: ActivePlayer, GoalKeeper and Team (and, of course, the GOAP system). Player has been made an abstract class that connects the players to the server and defines the behaviour functions declared by the GOAPAgent interface. ActivePlayer and GoalKeeper are classes that implement the Player class's abstract function specificRun(), which defines how exactly a player should run in the server. They also allow for us to reserve certain actions for certain types of players: for example, only GoalKeeper has access to the CatchBallAction class, while not having access to the MoveToBall class. Finally, the Team class offers a convenient way to initialize, start, and join all desired players at the same time. It also gives any member of a team convenient access to all of its teammates, which can be used in the future as a way to coordinate passes between players for example, or share any other useful information. The Player class also defined some functions that would calculate things like angles or distance to certain points. These, along with other math functions defined in the GlobalMap class, were moved to the PlayerMath class for convenience.

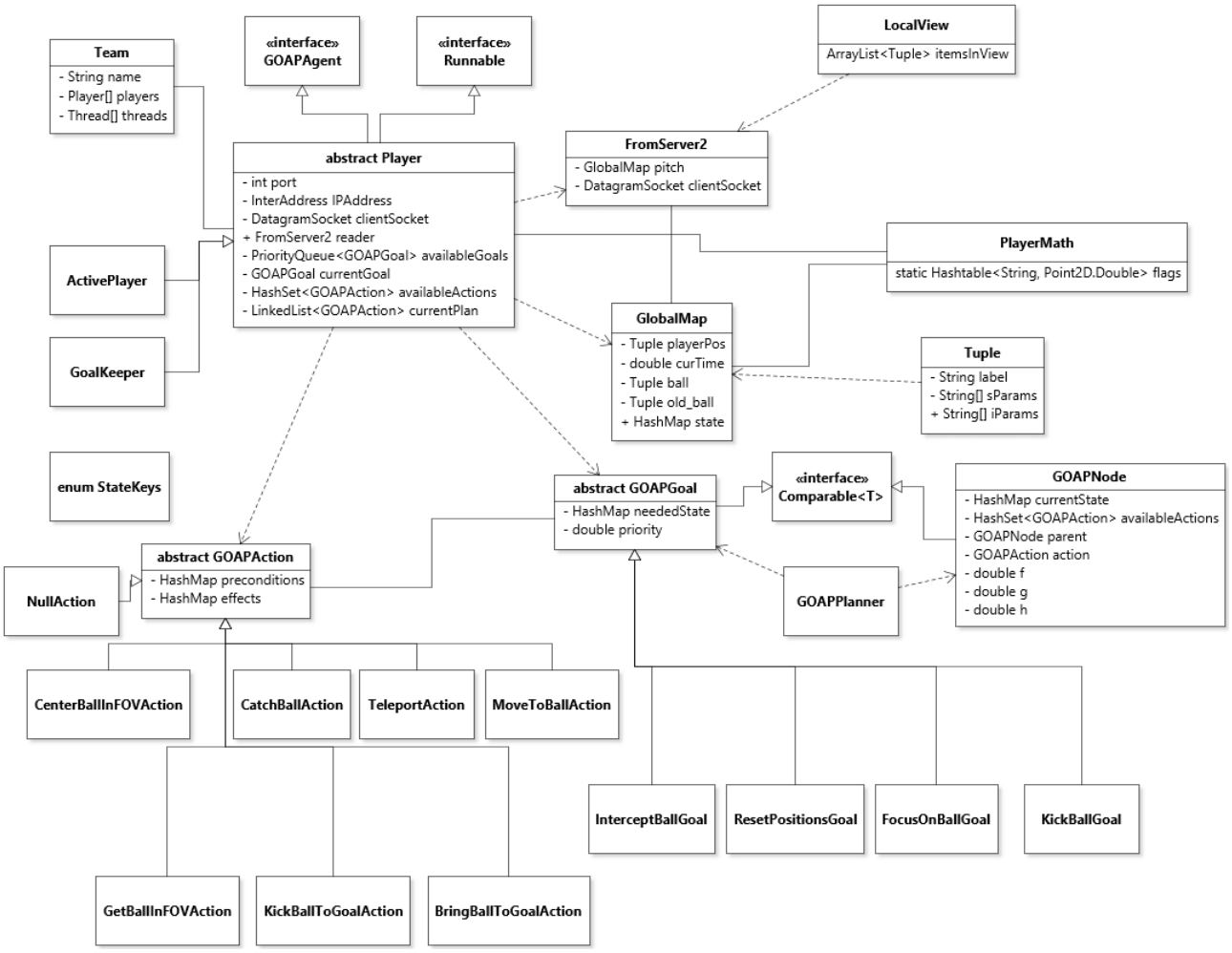

Figure 1: UML Domain Model of the current code.

5.1 Tester code

The functionality of the code can be observed by running the classes tester and gktester. Let us first consider gktester: in it, two goalkeepers will be initialized, one for each team, as well as a DummyPlayer if the proper code is commented out. If you decide to leave out the DummyPlayer, you can observe the behaviour of the goalkeepers by dropping the ball in various locations of the field. If the ball is close enough to the goal, the goalkeeper will try to match its y position. However, if that y position happens to be outside of the boundaries of the goal, the goalie will not follow the ball and will instead stay just inside the goal posts, observing the ball from there. If, finally, the ball gets kicked to the goal (the DummyPlayer will need to be active in order for that to happen), the goalie will catch it. This demo also clearly highlights some missing behaviour for the goalkeeper: indeed, even if the ball is just barely out of reach for a catch, the goalkeeper will not run towards it to either catch it that way or kick it away. Also, after the catch, the goalkeeper will do nothing with the ball until the server forcibly puts it back into play. These are all actions that have

not been implemented yet (even though kicking the ball away could be implemented by simply giving the goalkeeper access to the KickBallToGoalAction).

Figure 2: The goalkeeper running to defend a ball that is too far on its left. It doesn't run away from the goal to reach it.

The other class, tester, displays the behaviour of active players; currently, it initializes 3 players on each team, although the code should run correctly with any number of players. After the kick off is manually given, one player will start running to the ball. This is because whenever a team is created using the Team class, it automatically chooses a "designated kicker", which makes that player responsible for any kick offs or kick ins. This simulates the desired behaviour of having only one player run towards the ball to restart the match, but it is only a temporary fix: Indeed, as more actions are introduced into our system, it could be possible for the designated kicker to be on the other side of the field when a foul is declared, which would mean that it would have to run all the way to the ball to kick it off, even though another player could be closer to the ball. We would thus need a way to decide who becomes the designated kicker during run-time.
As soon as the ball is kicked off, all the other players will start converging on its location. There will be moments where the ball will leave the field of view of a player. In that case, it will stop running, find the ball again, then go back on course. Whenever a player is able to reach the ball, it will kick it in the direction of its opponent's goal, then once again start moving towards the ball. Of course, since they are only programmed to go towards the ball, after a few moments all six players will just agglomerate around the ball, with the ball progressing towards one goal or the other by random chance. There could also be a situation where one of the teams randomly commits a foul; in that case, the players will be moved around by the server, and the designated kicker will kick off the ball. Eventually, the ball will reach one of the goals; at that moment, the players will go back to their original positions, and the whole process will restart. If we keep the simulation running, once the time allotted to the match runs out, the players will disconnect, ending the program. The

current code does not handle one scenario however: it doesn't function correctly if the game reaches a penalty shootout stage; this is because, per the documentation, the shootout mode expects there to be a goalkeeper and a team of 11 players. This means that if this state happens to be reached, the players will do absolutely nothing, and the game will have to be terminated manually.

5.2 Deliverables and Schedule

All work was done by me, starting in early May and finishing in end of June. The GOAP system was made first, with the infrastructure (abstract classes for actions and goals, along with the planner and the node class) being finished in the end of May. The goalkeeper's first AI was done by early June, and the other players were done by mid-June. The revamp of the previous code was made continuously as it became more clear what functionality had to be updated. Final bug fixes were performed during the last week of June.

6. Future Work

There is of course still much work to be done before the code can be considered complete. First, there should be many more actions and goals available to the players. Next, cooperation and general awareness of team members should be expanded upon, which would most probably mean adding functionality to the Team class. For example, the Tuples that represent the ball could be migrated to that class, making all players aware of the location of the ball, even if they do not see it. For more awareness of opponent movements, it would be important to reinstate the code that would analyze positions of players on the field[3], and perhaps also place it in the Team class. Cooperation could also be helped by writing a coach for the team, which would connect to the match and give players more knowledge about the status of the field.

One feature I implemented but am not fully satisfied with is the MoveToBall action. Normally the GOAP system still includes a small finite state machine, which has just three states: an Idle state, for planning; an Animate state, for playing animations; and a Move state, for moving the agent. Since the Animate state is not needed for obvious reasons in our case, I decided against implementing the FSM and simply made the Move state into an Action. However, it might not have been wise, since if the ball ever leaves the player's field of view, the MoveToBall action breaks execution and forces the agent to re-plan. This is not necessary. It might be possible to have movement be dealt with by a separate class or function, which would be invoked if the next action in our plan has the "needRange" attribute set to true (which already exists in my code but is simply not being used), which would make the player more efficient.

Finally, there is room for hyperparameter tuning and experimenting with algorithms. Indeed, currently the priorities of the goals are set with arbitrary values that, for now, yield the behaviour I was expecting. However, as more goals get added it will be increasingly

3. The previous project created such code which would just place teammates of opponents into separate lists. It was momentarily removed from the project, but the code itself exists in the "old functions" text file, along with any other functions that may or may not be of use as this project develops.

difficult to decide a value like this for every goal. This could instead be tuned by playing a multitude of games and deciding which values result in the behaviour we desire. Similar experiments could be done for the setValue function in the GOAPNode class, which allows us to determine the heuristic cost of a node.

7. Conclusion

In conclusion, this project successfully created an effective system that will make future progress much faster. Integrating new goals would simply mean creating a new StateKey for the HashMaps, making the GlobalMap class identify when the key is satisfied or not, and creating some actions that would result in that goal completing. The current AI, although far from being capable with competing with any serious teams, shows that GOAP gives some promising results.

References

[1] The RoboCup Soccer Simulator. URL https://rcsoccersim.github.io/.

[2] Itsuki Noda et al. The RoboCup Soccer Simulator manual, . URL https://rcsoccersim.readthedocs.io/en/latest/introduction.html#f1.

[3] Mao Chen et al. The RoboCup Soccer Simulator manual, . URL https://rcsoccersim.github.io/rcssserver-manual-20030211.pdf.

[4] Melissa Katz Joseph Vybihal, Pedro Khalil Jacob. McGill soccer tournament repository, main branch, README. URL https://gitlab.cs.mcgill.ca/jvybihal/soccer-tournament/-/blob/main/README.md.

[5] Jeff Orkin. Applying goal-oriented action planning to games. . URL https://alumni.media.mit.edu/~jorkin/GOAP_draft_AIWisdom2_2003.pdf.

[6] Jeff Orkin. Three states and a plan: The A.I. of F.E.A.R. . URL https://alumni.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.pdf.

[7] Brent Owens. C# GOAP repository. URL https://github.com/sploreg/goap/tree/master/Assets/Standard%20Assets/Scripts/AI.

[8] Carleton University. Setting up a game of RoboCup soccer on Windows. URL https://carleton.ca/nmai/wp-content/uploads/Lesson-1-Setting-Up-a-Game-of-RoboCup-Soccer-.pdf.

[9] Joseph Vybihal and Pedro Khalil Jacob. McGill soccer tournament repository, master branch. URL https://gitlab.cs.mcgill.ca/jvybihal/soccer-tournament/.