Hand In for MAIfG 2012

Michele Ermacora, Tilman Geishauser and Francesco Guerra
Games (Technology)
IT University Copenhagen

Introduction

Aim

Our main goal with this project was to experiment with techniques, to learn from implementing them and to have fun. We were interested in prediction with neural networks and GOAP. Whilst thinking about how to implement GOAP we decided that it would be interesting to use Monte Carlo Tree Search (MCTS in the following) instead of A*. Our agent architecture also uses GOAP as a starting point.

We would not use these solutions if we were writing actual Unreal bots as AI Programmers for Epic, and would have focused other techniques, for example mirroring of the opponents actions, if we were serious about participating in the 2K BotPrize.

A secondary goal was to create a bot that would be able to navigate the map in a reasonable fashion. Furthermore, in a competitive sense, the bot should be interesting to play against for the best Unreal player in the team, Michele, and at the same time the bot should be defeatable by the worst player, Tilman.

Overall Design of the Bot

Adaption of GOAP to Unreal 2004

Unreal 2004 is different from F.E.A.R. in many ways. Game play is less tactical and there are no cover and no smart objects in the environment. Champandard (Champandard 2007) quotes Orkin on the matter which complexity GOAP is set out to solve:

> "In F.E.A.R., A.I. use cover more tactically, coordinating with squad members to lay suppression fire while others advance. A.I. only leave cover when threatened, and blind fire if they have no better position."

Thus we found it difficult to translate the abstraction level Orkin uses for his actions to the simpler single player game play in Unreal. For instance weapon changing can be done instantly in Unreal, but is a major tactical element as cover or squad behaviors are less relevant. We thus decided not to solve weapon selection in the Target Manager as it is done in GOAP, but to let our planning system solve the weapon selection problem. For example the bot can retreat to a health point whilst firing minigun, flak cannon or assault rifles - all weapons which fire many bullets with spread and low accuracy. If the bot has high health he can however decide to kill the player by engaging him using the potentially more powerful shock gun nuke action.

Setting and Performance

We only worked with 2-3 players at a time - usually testing with a 1-1 scenario and sometimes adding a second bot to test whether everything would work fine. This also reduced the significance of the working memory, for example target selection in such a simple scenario is not a highly interesting problem. Furthermore we do not need to care about detailed optimizations - we are using pogamut and gamebots, which is not an efficient solution compared to implementing bots inside a game, Unreal 2004 is an old game with low system requirements and we have only few agents at a time.

Architecture comparison

GOAP uses an architecture - shown in figure 0.1 - based on the C4 architecture by (Burke et al. 2001). Sensors pass perceptions, including internal and external ones, to the working memory. The information in the working memory is used for planning once a goal has been selected - a key difference to the C4 architecture being the real time planner . If

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

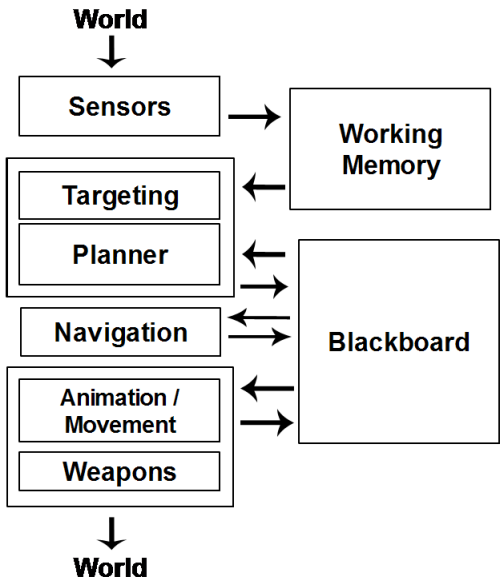

Fig. 0.1: GOAP agent architecture, taken from (Long 2007)

the facts in the working memory change the highest priority goal is re-evaluated. Subsystems such as navigation or the animation system are instructed via the blackboard, which stores information from these systems and the planning system. (Orkin 2005).

Our architecture is similar. We did however not get to implement a proper working memory and use the blackboard instead.

World state

Orkin uses fixed sized arrays for maintaining variables about a world state. Our world state is realized as a fixed size data structure by basing our world state boolean array on an enumeration of the symbols. As symbols in the goal state can be irrelevant in addition to being true or false, like uninstantiated variables in logic, we created an enum expressing true, false and uninstantiated. A third enumeration describes the possible goal, and a switch case in the world state class can safely create goal states based on this enumeration.

- Goals: KillEnemy, SearchRandomly, EmptyAmmunition, Survive, SearchAdrenaline, SuppressionFire, FindWeapons

Actions

We implemented various actions that can be used by the planner.

- Random Walk: Randomly run around on the map, trying to find a player.
  - preconditions:
  - postconditions
- Retreat with suppression fire: Try to find a health pack, and shoot at the player with minigun, flak cannon or assault rifle.
  - preconditions:
  - postconditions:
- Find shock gun ammuntion: If there is shock gun ammunition on the map go there.
  - preconditions:
  - postconditions:
- Shock gun nuke: Fire the secondary mode of the shock gun, and after some time shoot the primary mode. If the primary mode hits the secondary projectile a huge explosion will happen.
  - preconditions:
  - postconditions:
- Find health pack: Look for health packs. Which health pack to go to depends on various factors (TODO: FINISH THIS TEXT)
  - preconditions:
  - postconditions:
- Shoot grenade: Will fire a grenade at a player position.
  - preconditions:
  - postconditions:

Planner

The planning is done inside the GOAPPlanner class and replanning is triggered every time the state of the world changes in a way that makes a new goal more desireable than the current one, or when the actual plan either fails or succeeds. To create a plan our system uses Monte Carlo Tree Search (MCTS) with the UCT formula (we use as a starting point Michele's MCTS used for the PacMan project). We decided to use this algorithm, instead of the usual A* used by Orkin, for different reasons. On the one hand we were just interested in trying this out. We were furthermore expecting that we can search a more vast state space, while with the A* algorithm we take in each step the best action possible in that moment and then we iterate on that. With the MCTS we perform also exploration of other nodes that can

lead to other interesting plans - we were hoping to achieve more variety in agent behaviour. For example we could find two different plans which satisfy the same goal and have the same reward, and a different one randomly every time. Although, while the MCTS is probably more computational expensive in tems of CPU and RAM usage, it gives a desirable degree of control in the sense that the user can dynamically change the time used by the algorithm to create a plan and the number of simulations inside the simulation step. This allows us to scale the artificial intelligence of the bot to powerful machines.

The MCTS takes as parameters the actual goal and world state. From these it tries to look for a plan.

The most significant changes inside the algorithm are inside the Node class, as it is in this class that, for each node created, we store the new information regarding action selected, new world state, new goal state. The Node class also performs the simulation step. The algorithm works as follow:

- The MCTS applies the tree policy and, if the node has not yet generated all his children, it creates a new node passing, as parameters, the goals state and the world state simulated by the parent from which the node was created plus the action selected for the child. The action selection is done by the node, before creating the new child, and is picked from a pool of actions that represent the ones that can, by applying the postconditions, satisfy in a partial or complete way the goal state. In case the node is already fully expanded it picks the child selected by the UCT formula and tries to expand it as in the standard algorithm.
- After the node is created, it updates its internal goal state and world state by applying, in order (SEE EDMUND LONG p 33):
  - To a temporary copy of the world state the post condition of the action that was chosen for this node
  - To a temporary copy of the goal state the pre conditions of the action choose for this node
- Then the MCTS runs the simulation step where it has to calculate the delta value for that node. In the simulation step we give a big reward if the action chosen is a terminal condition - that is, it is a goal state. A small reward is given if one of the action has to take further steps to reach a final condition. This value is a product of 1 divided be the count of unsatisfied preconditions and an designed reward that allows us to control the value of that particular action according to our design decisions.

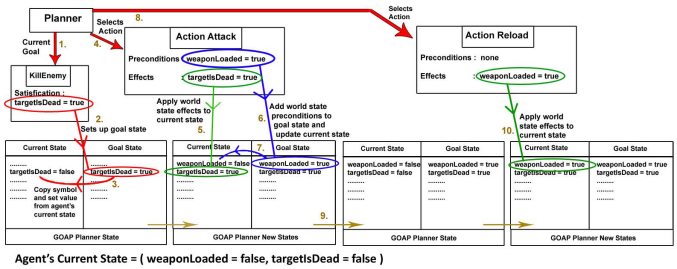

Fig. 0.2: Application of actions during planning, taken from (Long 2007)

- Finally, in the back propagation step, we back propagate the values. While also updating the parent's node best child variable in case the current node is better than the current one. This variable is used later to access in a fast way, from the root, the different actions composing the plan.

When the planner finally receives a plan from the MCTS, it stores it in the form of a stack in the blackboard to be consumed by the actor system.

Designed rewards

We gave every action a value that expresses how much we, as designers of the system, would like to see the bot do the action. For example shooting the rocket launcher has a high reward, whereas searching rocket launcher ammunition has a low reward. These values, whilst using a hardcoded base value, are calculated each time the bot is updated. For example if we are already have ammunition for the rocket launcher finding more ammunition of that kind is less attractive.

Movement

The low level logic of the movement is handle by the MovementLogic class. We have three different kind of behaviours which are controlled by our actions:

- If the bot can see an opponent and is shooting at him, the HuntingPlayer behaviour is activated. In this context, the bot tries to reach the player while performing random movements (as strafe left, right or double jump) if it is hit or it can see a projectile coming.
- A random walk behaviour, that make the bot move by taking random navigation points of the map. This behaviour is triggered when the bot is looking for other players and as fall back behaviour.

- A specific behaviour, in which the bot tries to reach a well defined point in the map selected by some of our actions. For example if the bot is looking for health he will go to health pack preferring, in the following order spawned, visible and strong health packs.

These methods are used only if there is no collision. If our raycast system, composed out 5 rays [COMPARISON WITH PAPER], detects collisions we move the bot manually and try to make it unstuck. This system doesn't work very well because we do not remember problematic areas, neither do we check whether a planned movement will get the bot stuck. Thus, if the bot is actually engaged in a fight, even if it gets unstuck, immediately thereafter the bot can perform a random movement that get him stuck again.

Prediction of enemy position with a neural network

One of the human features that for example hunters exhibit is to predict the location of some object after observing its previous movements. When an hunter, or, inour case, a player that is playing a first person shooter, has to predict the location of an object in the future in order to shoot at it, he considers some variables like the previous locations of the object, the speed of the object (and also his speed if he is moving), the speed of the projectile and the distance between him and the object. This human feature is not infallible, in fact if his assumptions are not met, for example when the object makes an unexpected movement, the hunter will miss the object.

We have tried to model this human feature and we modeled this behavior with a neural network. Between all the artificial intelligence techniques that we have studied during the lectures, we thought that the neural network is the best technique for this kind of problem, since the purpose is to understand a quite complex function that involves many variables. It is not the optimal solution for this problem, for example it is possible to use the prediction calculated by a Kalman filter (Welch and Bishop 1995) by setting the error in the matrix to zero. We were however not interested in an optimal solution, but in a solution that helps creating a fun, interesting and human like interaction with the bot. We also decided that it would be interesting to experiment with neural networks in a concrete and quite complex problem: a First Person Shooter environment.

Input and Postprocessing of Output

First, we had to decide which input to use. Basically we had to decide whether to collect a huge amount of player behaviours, like moving really fast on left and right, walking around, jumping continuously, and so on. This way we could have worked on long-term prediction. On the other hand we could collect simple, almost linear, movement, so that the neural network was able to infer a position in the near future, based on simple consideration. We thought at the end that the best solution would have been just to have short time prediction for two reasons: First it is more similar to human hunting behavior and exhibits the same shortcomings; Second it's easier to create a training set for the neural network to have better results.

Another problem very important that emerged was what kind of inputs we had to use for the neural network. The output had to be a position, which means three different outputs representing the position on X, Y and Z. Analyzing the Pogamut framework we realized that we didn't have to use the position of our bot as input, since there is a function allowing to shoot at a desired position. Therefore only focused on the parameters of the enemy. Additionally we assumed that we needed the previous position of the enemy and the corresponding time at which each position was recorded. Because of the fixed updates of the bot logic we receive the position of the enemy at regular intervals. Then we thought that if the time-frame which we take the position of the enemy is regular, we didn't need time as input. Since in this way the time is a constant, the neural network doesn't need it anymore to build the function. Another advantage of not using time as input is that the Pogamut Framework and the multi-threaded nature of the environment we work in results in our code being executed at slightly different times. We measured this and the error is usually in the range of some milliseconds. As the enemy movement logic is executed at the same intervals as our bot logic, using exact times of retrieval of bot position as input might have introduced small errors.

For our first test we decided to not use the direction and the speed of the enemy, because we thought that the neural network could be able to "infer" those parameters, and at the same time they could have been misleading for the neural network (even though we don't have proof of that because we didn't implement that solution since the neural network worked in the first test). One issue we had was to decide how many predicted position we needed to have and at which frame interval. This problems come from the fact

that each weapon has a different speed, and the time which the projectile reaches the target, also change according to the distance from it. To solve this problem we have chosen to have only one mid-term time prediction, and retrieving the all the predictions in between time 0 and the predicted time of the neural network, making a linear interpolation with vectors. In order to apply that algorithm we assumed that the movement of the enemy is linear. So we develop a lerp function that takes as parameter two locations and a weight. The first parameter is the current enemy position, the second one is the predicted position from the neural network and the third is a weight between 0 and 1, where 0 represent the first position, 1 the second one and all the number in between represent a linear interpolation between the two vectors. To decide the best setting for the neural network we had to face some problem. The first problem came from the Pogamut framework (was actually inherited from Unreal). That was not actually a problem, was more a limitation to avoid unfair bot behaviors. Basically is not possible to get an enemy position if it is not visible. In this way was not possible to make an instant prediction since we needed to collect previous locations to set the inputs for the neural network. The second problem was setting a proper value for the output of the neural network, which means a prediction in one particular time of the future. To set the best value we look at the speed of several weapons and we have noticed that the slower one was the rocket launcher. So we tested the speed of the projectile of the rocket launcher from several distance and we found a trade-off. So basically the prediction was stetted to 0.6 second. Having all those data we could also find a proper setting for the input. So we assumed a good setting for the neural network was 4 different inputs, each one a position whit 0.2 second of difference and the output a position in the future, 0.6 second of different from the last input. In implementing the neural network we also faced other problem like the preparation and the normalization of the data (inputs and the desired output for the back propagation in the training phase). The first option was to give as input the coordinate in world frame or local frame relative to the bot. Neither of those solutions was good, essentially for the problem of normalization. Each level is different and has a set of minimum and maximum coordinate, in this way is impossible to find a value that fits, in a proper way, our problem. So we came out with the idea of setting the input in local coordinate relative to the first input (position) in the neural network. The normalization factor was 400 which mean the max number of units a bot can cover in 0.6 second and the output was the position relative to the last input of the neural network. At that point the setting of the neural network was complete, with 12 input (4 positions) and 3 outputs (1 position). Basically the bot needs 0.8 second from when it sees a player to begin to make a prediction. If it visually lost its target for more than 0.1 second it has to start to collect data from scratch. To solve this problem the class Predictor, which manage data and the neural network to perform the prediction, returns the current target location in case a prediction is not available.

Training Phase

In order to train the neural network with backpropagation, we decided to create the data set recording the position of a character we were playing with. Since the Pogamut framework does not allow retrieval of the position of a player if he is not visible, we had to implement a UT Server that is omniscient, in order to get our character position for each frame. In the beginning we thought that it was not possible to create incoherent and misleading data for the neural network when just moving around the player, but this assumption was wrong and we are going to explain it in detail in the following.

After we collected the inputs from several games we had to prepare and convert the data for the neural network, which means converting it from world to local frame, normalize and put it in a proper data structure so that the neural network is able to read it (square arrays). The first test didn't work. We tried various topologies of the neural network and several values for the learning rate, but both the average error (calculated as desired output – actual output) for all the samples and the error of some of the sample was extremely slow (almost zero) in converging. The only thing the neural network was able to understand was that when the target doesn't move in the previous position, it will also not move in the future. Basically most of the output was near to 0, which means that the player is not moving from the last input position. After a while we understand the reason why: the data we collect were incoherent. Basically moving randomly we generated incoherent data. For example in one case the inputs were positive positions along a direction and the output (the position after 0.6 second after the last input) was also a positive value, the second time with the same kind of input we had a negative output, which means that we changed direction in the end of the path.

After we understood that error, we tried to generate training data again, trying to not make movement that could have

been misleading for the neural network. So basically we moved our character almost in a linear way, trying to cover as many directions as possible. Then we tested this new training set and the average error was decreasing, so the neural network was converging towards the desired output. Then we tried several topologies and parameters to optimize the result. Unfortunately it was difficult to explore the consequences of different settings since for each setting the convergence speed was pretty low. Based on these experiments and the remaining time we choose a setting that gave us a better speed of convergence more than an accurate result, assuming that we would not need a very high accuracy for our bot. Our final neural network was created with a learning rate of 0.9 (very high) and a topology following the "rule of thumb" (Heaton 2007), setting the number of the hidden layer as the average between the input and the output layer. When putting the neural network into our project we stopped learning and only exploited what had been created. This required some post-processing the output - application of the inverse operation of normalization and conversion from local to world frame. Testing the result both in a testing environment and the standard environment we discovered that the neural network was not able to make a right prediction for all the directions. For one direction we get a very good prediction, but the further away the actual movement is from this direction the less well we are able to predict where the enemy will move.

Evaluation

Interaction with the bot is enjoyable for a short time. He can do some fun things, for example he gets the shock gun nuke right at times, or dodges enemy fire interestingly when retreating. At the same time the bot is not challenging enough to create a really fun engagement.

Conclusion

Movment. A way for resolve this bug could be check, before perform a random movement, if the bot will collide with some other objects.

There are various factors we would have to consider in order to enhance the performance of our neural network. We now understand that it has been problematic to produce the training set "by hand". Our data set might be too sparse. It would also be interesting to train the neural network with a lower learning rate.

References

Burke, R.; Isla, D.; Downie, M.; Ivanov, Y.; and Blumberg, B. 2001. Creature Smarts: The Art and Architecture of a Virtual Brain. In *IN PROCEEDINGS OF THE COMPUTER GAME DEVELOPERS CONFERENCE*, 147–166.

Champandard, A. J. 2007. Assaulting F.E.A.R.s AI: 29 Tricks to Arm Your Game. `http://aigamedev.com/open/review/fear-ai/`.

Heaton, J. 2007. A Feed Forward Neural Network . `http://www.heatonresearch.com/articles/5/page2.html`.

Long, E. 2007. Enhanced NPC Behaviour using Goal Oriented Action Planning.

Orkin, J. 2005. Agent Architecture Considerations for Real-Time Planning in Games. In *Artificial Intelligence and Interactive Digital Entertainment*. AAAI Press.

Welch, G., and Bishop, G. 1995. An Introduction to the Kalman Filter. Technical report, Chapel Hill, NC, USA.

Appendix

World state symbols:

- PlayerIsVisible
- HasSuppressionAmmunition
- IsTargetDead
- ShockGunAmmunition
- HasLowHealth
- HasGrenadeAmmunition
- HasRocketAmmunition
- HasLightiningGunAmmo
- HasMachineGunAmmo
- HasFlakAmmo
- HasAdrenaline
- PerformAdrenalineAction
- SuppressionFire
- HasGunAmmunition