Proceedings of the First
Artificial Intelligence and Interactive
Digital Entertainment Conference

A Goal-Based Architecture for Opposing Player AI

Kevin Dill

420 Massachusetts Ave
Lexington, MA 02420
kad@bluefang.com

Denis Papp

TimeGate Studios, Inc.
14140 Southwest Freeway, Suite 400
Sugar Land, TX 77478
denis@timegate.com

Abstract

This paper describes a goal-based architecture which provides a single source for all high level decisions made by AI players in real-time strategy games. The architecture is easily extensible, flexible enough to be rapidly adapted to multiple different games, and powerful enough to provide a good challenge on a random, unexplored map without unfair advantages or visible cheating. This framework was applied successfully in the development of two games at TimeGate Studios – Kohan2: Kings of War and Axis & Allies.

Overview

Our mission was to create AI players for two simultaneously developed real-time strategy games (Kohan2: Kings of War and Axis & Allies). After having worked on previous projects that required large amounts of custom-written AI code for each decision made, we decided that our primary objective was to provide a single, easily extensible source for all high-level decisions. In addition, we needed to provide a solid challenge against a human player on a random map, with no economic or military advantage and no visible cheating. Finally, we had some tertiary objectives such as support for different AI personalities, user-created AIs, limited teamwork between allied AIs, and the ability for human players to use team commands to influence the actions of allied AIs.

In order to accomplish these goals, we developed a two-layered architecture. The strategic AI uses our *goal engine* to make broad decisions which outline the overall strategy to be followed. It is computationally expensive, but uses a time-sliced architecture and only needs to think occasionally. The reactive AI then fills in the second-to-second decisions involved in implementing that strategy.

Our general philosophy throughout this project was that it is much better to give the AI the ability to make its own decisions about what to do based on the current situation rather than hard coding or scripting the AIs decisions. We felt that this was vital for replayability and for the AI to be flexible enough to play successfully on random maps. We also felt (correctly, as it turned out) that this philosophy would promote emergent behavior that would make the AI far more effective than anything we might script. With all of that said, we also recognized that there are situations in which the designer knows best. Particularly in the early game, there are certain steps which should be reproduced every time in order to give the AI a strong opening.

Architecture

Goals

The basic building blocks of our architecture are *goals*. Every action that a player might make is described as a goal, which can be assigned a *priority* which indicates the importance of executing that action given the current situation. A simplified set of goal types might include:

ATTACK – attack an enemy player
DEFEND – defend your own or an allied position against enemy attack
RECOVER – retreat a damaged actor to a safe location where it can be repaired
EXPLORE – explore the map, either to discover new locations or to scout for new enemy construction in areas you've already seen
RECRUIT – build new military or civilian actors
CONSTRUCT – build new buildings
GIVE – give money or actors to an allied player

Copyright © 2005, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

DESTROY – sell a building or disband an actor

Each goal type can have many instances, and goal instances can be dynamically generated and destroyed. For example, you might have one ATTACK goal for each enemy actor, dynamically creating and destroying these goals as targets enter and exit the game world. The vast majority of the logic is placed in the goal itself, rather than in the engine, so that custom goals can be written which generate their priority based on whatever information seems appropriate.

Goals are implemented by subclassing from a base class. The subclass must include code to determine whether the goal is currently *active*, whether it is *finished*, and what its *base* and *current priorities* are, as well as an indication of what sorts of *resources* (such as money or military actors) are required for execution.

A goal is *active* if it is currently reasonable to execute it. Consider, for example, the RECOVER goal. This goal allows an actor to recover when it is damaged, with a priority that depends on the amount of damage the actor has taken. If the actor is currently at full health then this goal is inactive, which allows us to short circuit the priority calculations and reduce the number of goals we are comparing in the optimization step (more on all this later).

A goal is *finished* if, for one reason or another, we can get rid of it entirely. For example an ATTACK goal is finished if we successfully destroy the target actor. A CONSTRUCT goal is finished once the building in question has been built. Marking a goal as finished gives us a way to schedule the instance for deletion at a safe time.

The *base priority* is the relative priority of that goal compared to all others given the current game environment but assuming some reasonable allocation of resources. For example, the base priority of the ATTACK and DEFEND goals assumes that the total friendly strength assigned to the goal is 1.2 times the total enemy strength in the region (a tweakable setting defined for each AI personality). Similarly, the CONSTRUCT and RECRUIT goals assume that sufficient resources can be reserved to pay for the actor being created.

The *current priority* is the priority of the goal with the resources actually assigned. For example the ATTACK and DEFEND goals will have higher priority if you assign more strength to them, and lower priority if you assign less strength. Likewise, the priority of the EXPLORE goal will vary depending on the location of the scout actor assigned.

A goal is *selected* if we are actually going to execute it.

The Think Cycle

High-level strategic decisions are made by periodically running through the goal engine's "think" process. Goals selected during one think will remain active at least until the next time the goal engine thinks.

The vast majority of thinks are scheduled approximately 30 seconds apart, but we also have support for certain event-driven thinks. For example we might want to think immediately when we capture an enemy city, or when we build a new structure. We generally only use event-driven thinks early in the game, when it is essential that no time be wasted if we are to remain competitive with human players.

The first step of the think cycle is to go through all of the goals and find those which are either finished or currently inactive. These goals are culled from the list of candidates so that we don't waste further time on them.

Next, we determine the base priority of each candidate goal. To this we add a random *fuzzy factor*, and if the goal was selected in the previous think cycle we also add a bonus for *goal inertia*. The goal inertia bonus is generally larger than the fuzzy factor, and is intended to prevent the AI from flip-flopping between two similarly attractive actions.

Now that we know the priority of all the active goals, we face a fairly standard resource allocation problem. The specific resources to be assigned could include actors, money, or whatever other resources are necessary to execute a goal. For example we assign military actors to attack and defend goals, money and builder actors to CONSTRUCT goals, and scout actors to EXPLORE goals. We need to allocate these resources to each goal in such a way as to optimize the total priority of all selected goals. This problem is made more complicated by the fact that the priority of a goal may vary depending on the resources assigned to that goal. Our third and fourth steps address this problem.

In our third step, we sort the goals by priority and generate an initial resource assignment to each one. For each goal we assign just enough resources to have a reasonable chance of success – in general, this is the same as the resource assignment assumed when the base priority is calculated.

For goals which require money, such as CONSTRUCT or GIVE, we apply the notion of *goal commitment*. The general idea is that we want to be able to save up sufficient money to execute the more expensive goals as long as that money can be obtained within a reasonable amount of time. In practice, this means that we specify the maximum amount of time we're willing to wait and save money in order to accomplish a goal. When we come to a goal for which we have insufficient cash on hand but all other resources (such as builder actors) are available, we check how long it will take to save the required cash at our current income rate. If this is less than the specified time limit then we mark all lower priority goals which require monetary resources as inactive for this think cycle in order to prevent them from using resources needed for this goal.

The fourth step is actor optimization. In this step, we go through all the goals which require military actors (such as ATTACK and DEFEND goals) and calculate the change in overall priority if we move actors between them. In certain

cases it might be advantageous to launch one extremely strong attack, for example, while in other cases you might have enough forces to launch two or more weaker attacks.

Similar to goal commitment, we found some cases where it was advantageous to *lock actors* onto a goal. This is an issue which needs to be approached cautiously, since it violates our basic philosophy that it is better to let the AI make its own decisions rather than hard-coding behavior. In almost all cases you want the AI to retain the flexibility to be able to change its mind when the situation changes, but there are a few specific circumstances where it was necessary to override that.

There are two situations in which we lock actors. First, if an actor is assigned to an ATTACK goal and it has already engaged the enemy forces it is targeting, it appears like a mistake for that actor to leave in the middle of the attack to fulfill another goal (unless it is forced to retreat, of course). Even if this reassignment is the correct course of action (which is extremely rare), the player perceives that the AI has made an error. We always want to maintain the illusion of intelligence in the eyes of the player, and doing so requires first and foremost that we avoid actions which appear to be poorly selected. The second situation is when an actor is on a RECOVER goal, we lock it until it has reached full strength (assuming there is a safe location available for recovery). Prior to adding this lock we would frequently observe the AI using actors when they were not quite fully healed, and again this appeared like the AI was making a mistake.

Once we've completed actor optimization we can mark the goals which we are executing as selected and issue the appropriate commands. The goal engine then sleeps until the next think.

The Reactive AI

The goal engine provides only the high level strategy, which will carry us over relatively long time periods. Obviously, a finer granularity AI is needed to make the second-to-second decisions which keep the AI competitive. For this we used the *reactive AI*, or RAI. The RAI runs on a one second think cycle, and makes decisions such as selecting formations, coordinating arrival of actors, ordering damaged actors to retreat, and selecting targets for military actors in combat. Unlike the SAI it is lightweight enough that it doesn't require interruption during its decision cycle.

Egos

One important requirement of our system is that the designers want to be able to create multiple distinct personalities for the AI. For example we might have AIs that prefer certain types of actors, AIs that prefer a rushing strategy, AIs that prefer to build a strong economic base before attacking, and so forth. In addition, we want to allow the players to create their own AIs. This proved to be valuable during beta testing, when several testers assisted with final polish.

In order to support this requirement, we use a data-driven design. Each opposing player is described by an *AI profile*, which consists of a name, a brief description, a list of the nations or races that the profile can best support, and a list of the *egos* which can be used by that profile (including a starting ego to be used when the game begins).

The vast majority of the data required to run the AI is contained in the egos. A fully defined ego typically includes thousands of lines of values – although we do have support for inheritance, so many of those values are defined once and then inherited by all other egos. A typical AI profile includes at least three egos (one for the early game, one for when things are going well, and one for emergencies), although many of the player-created profiles contain far more than that. The information provided here represents only a tiny subset of the total amount of data provided in the ego, but it includes what we believe to be the most important (and commonly used) sections.

Each ego first contains a list of filters which have to be satisfied in order for that ego to be activated. It is the responsibility of the designer to ensure that the filters are set up so that there will always be at least one ego available (although this failure is handled gracefully). Once an ego is selected, it will continue to be used until one or more of its filters are no longer satisfied. At that point, we activate the first acceptable ego on the list.

We have a fairly powerful filter system, in which any of a variety of tracked statistics can also be used to filter egos. The following are the most commonly used statistics.

- current monetary income
- number of cities controlled
- number of military actors available

One thing we do not filter on is elapsed time, again due to our philosophy of allowing the AI to make its own decisions based on the game situation. It is infeasible to predict how quickly the AI will develop given a random map and an unpredictable set of opponents. For example, in one game the AI might control three cities in the first ten minutes of the game, while in another the AI struggles for longer than that just to obtain a second city. From a design point of view, it is better to find ways to filter based on the game situation rather than using an arbitrary measure which has no bearing on the events which have actually occurred.

The next section of the ego includes control parameters for the goal engine and RAI. For example, values to enable or disable event-based thinks of various types, and flow control requirements for the network traffic. It also includes the fuzzy factor and goal inertia to be used when calculating base priority.

The ego can include construction templates, which give it sets of buildings which should be built together in cities.

All of the stock egos include at least one construction template which ensures that we have a city which can build strong military actors. Another commonly used construction template encourages strong economic development in the remaining cities. Construction templates simply provide a bonus to the construct goal – it is up to the designer to determine whether that bonus should be large (making it a mandate) or small (making it a suggestion).

We also support military templates, which tell the ego which military actors to use together. For example we might have an ego that prefers to mix infantry and ranged actors, or one which likes to use a mix of light and heavy cavalry. Again, these can be mandates or suggestions, and we can also control what portion of our total actors will be influenced by these templates.

Next we have controls which allow us to tweak the overall balance between goal types. These include additive and multiplicative bonuses for each goal type. Any goal with a base priority greater than zero will have these bonuses applied. One of the most useful pieces of debugging output we have is a spreadsheet which contains every candidate goal considered during the course of a game, when it was considered, and its base and current priorities. This allows us to easily balance the overall priorities of the different goal types. It also gives us a first indication of what has gone wrong when we see the AI making poor decisions.

Finally, we have a plethora of values used to tune the priorities for each goal type as well as the RAI. Here are some common tricks we use to tune goal priorities:

- Additive and multiplicative bonuses, which can be used to scale any numerical value depending on its importance to the ego. An additive bonus is only applied if the value is already greater than zero.
- Exponents, which can be used to curve the range of values. For example when you have insufficient strength to launch an attack, the goal is still given some priority but that priority drops off exponentially based on the ratio between your strength and the enemy's strength.
- Minimum and maximum values, which keep some components from getting too large (or, in the case of a multiplier, too small).
- Inversion. In some cases, certain egos want a value to be large while others want it to be small. For example, some egos prefer to explore close to known territory while others prefer to explore distant areas first. The first strategy helps us to find useful areas such as hostile lairs and resource locations, while the second strategy is more effective in searching for enemy players. In order to control this, we give the option to take the inverse of that value (1 / value) for that component of the priority. We then use a multiplicative bonus and a maximum value to scale the result appropriately
- Repeat penalties. For example, we might have a repeat penalty for recruiting certain actor types. We apply the penalty once for each existing actor of that type. If we have so many actors of that type that the total priority is less than zero, then we won't recruit any more.
- One time bonuses. These are similar to repeat penalties but are only applied if we don't have any actors of the specified type. For example we might have a one time bonus to build a structure which allows us to recruit certain types of actors. By using one time bonuses with different values we can exert a high level of control over the order in which the ego constructs its initial structures.
- Fuzzy factors. In many cases we need a bit of extra randomness. For example we use a fuzzy factor in the RECRUIT goal to influence the selection of the specific actors to be recruited.

The result is a complicated yet extremely powerful set of parameters for describing how strongly to weight every consideration used by the AI. It is obviously still the responsibility of the programmer to identify what factors the AI should consider, but designers or even players can then make AIs with highly distinct preferences by tweaking the weights of those considerations.

Cheating

The original goal was to create an AI that did not cheat at all. For the most part we accomplished this. However we did find that in very limited circumstances it was useful to provide the AI with information that it technically should not possess, but which would often be available to a human player through intuition and meta-knowledge of how random map generation works.

Our first cheat was to provide the AI with perfect knowledge of enemy strengths (but not actual troop locations). We tracked enemy strength using a modified influence map, and when doing so we included the strengths of actors that were concealed from the AI by the fog of war. We did not provide the AI with specific knowledge of those actors or allow it to attack actors (or buildings) that it could not see. While the information provided is technically more than a human would have, we find that a moderately experienced player actually has a fairly good intuition of enemy strengths, so we didn't feel that we were being unreasonable in providing a similar intuition to the computer player.

This cheat had two significant benefits. First, it saved us the headache of trying to track the probable location of actors which the AI has seen once and then lost into the fog. This is a difficult (and bug-prone) problem. Second, it creates the illusion of an AI that is intelligent. Reviewers raved about an AI which could launch diversionary

attacks, probe, and use all sorts of advanced strategies. In reality, much of this was an illusion created by the fact that the AI was tracking the players changing troop strengths and reacting accordingly. Judging by player reactions, it seems that this illusion made the AI quite a bit more fun to play against.

Our second cheat was in the domain of exploration. Quite simply, we found that occasionally the AI would simply fail to find the enemy (or anything else of interest) until fairly late in the game, and therefore would never get off the ground. Obviously, an AI which doesn't present any challenge in this way isn't a lot of fun to play against, and even if it's something that happens only occasionally we would like to minimize that frequency. In order to do this, we provided the AI with a small bonus to explore areas with "interesting" items (such as monster lairs, resource points, enemy buildings, etc). Exactly how large the bonus was and which sorts of items the AI would look for depended on the ego. We found that experienced players are able to predict with a fair level of accuracy where these interesting areas are likely to be based on past observation of randomly created maps, so this seemed like reasonable information to give to the AI as well.

Related Work

The philosophy behind our AI borrows much from the Dark Reign model for Strategic AI (Davis 1999). Although the implementation details of our systems vary, the general architecture and desired results are much the same.

The GRUE system presented in (Gordon & Logan 2004) matches resources to goals fairly strictly. In contrast, the goal engine can vary the priority of a goal based not only on which resources are assigned but how many resources of a particular type are assigned. This can be arbitrarily complicated (for instance giving a bonus to the priority of an attack goal which has both ranged and melee units assigned). On the other hand, GRUE can handle chaining multiple goals (for instance acquire the machine gun in order to attack the enemy). The goal engine simply executes whichever goals are highest priority at the current point in time.

The SOAR Quakebot (Laird 2000) implements a death match player for Quake II. Although SOAR is designed to reason about goals, this architecture lacks the ability to execute multiple goals at the same time. While that makes sense for a Quakebot controlling a single agent, it would obviously not work well for an RTS AI.

The being-in-the-world agent (DePristo & Zubek 2001) uses a hybrid architecture approach with a truth maintenance and reasoning component formulating the high level goals for a reactive layer. Although this system was successful in executing goals and surviving in the complex world of a Multi-user Dungeon, it ran into problems representing continuous resources (such as gold) and re-enabling invalidated goals in the truth maintenance system. Rather than trying to make every goal conform to the requirements of our reasoning engine, the goal engine places the majority of the logic in the goal itself so that it can be customized to the needs of that particular task.

Finally, this section would be remiss if it failed to mention the AI Game Programming Wisdom books (Rabin 2002; Rabin 2003). These books have a wide variety of articles on game AI, many of which could be related to our work in one way or another.

Conclusions

Overall, our architecture was a tremendous success (it delivered a fun, challenging game experience) and received significant critical acclaim. We believe that the most important factor in this success was our decision to avoid hard-coded decisions to the greatest extent possible, and instead create an AI which contains the intelligence needed to make its decisions on the fly based on the current perceived situation. Our data driven design and easily expansible architecture also made it far easier to keep up with the ever-changing design of the game than it had been on previous projects. This was critical to our success when we had design changes being made late in the project.

Creating egos turned out to be more difficult than we had hoped; it requires a fairly technical mindset and a solid understanding of the design philosophy behind the AI. Although we hoped to have designer-created egos, in practice the AI programmer ended up doing the majority of the data work as well. Some beta-testers did create their own profiles, and some were extremely good (able to beat the shipping profiles on a random map), but doing so required significant support on our part. Some time could be invested to improve this situation, such as tutorials and tools, but the bottom line is that any architecture this powerful and flexible will be complex. In fact, there are many advantages to having the AI programmer create the profiles, because this helps understand the code changes needed to solve problems and limitations encountered.

Perhaps the biggest failure was one not inherent in the design, but rather a matter of focus. Our primary focus was to develop an AI which could handle random maps, but we lost track of the (often divergent) needs of the campaign AI. The campaign missions were scripted and the designers needed the ability to control behavior or communicate objectives to the AI in ways which we did not support. This resulted in some cases where the AI was awkward or simply disabled in the campaign. Had we recognized this problem earlier and focused on it, we believe both modes of play could have been well supported.

Acknowledgements

The authors would like to thank the rest of the team at TimeGate Studios for making this work possible. A great deal of time and effort went into these two projects, and

the AI greatly benefited from the feedback and assistance we received from all involved.

References

Davis, I. L. 1999. Strategies for Strategy Game AI. In *Proceedings of the 1999 AAAI Spring Symposium on Artificial Intelligence and Computer Games*.

DePristo, M., and Zubek, R. 2001. being-in-the-world. In *Proceedings of the 2001 AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment.*

Gordon, E., and Logan, B. 2004. Game Over: You have been beaten by a GRUE. In *Proceedings of the 2004 AAAI Workshop on Challenges in Game AI.*

Laird, J. 2000. It knows what you're going to do: Adding anticipation to a Quakebot. In *Proceedings of the AAAI 2000 Spring Symposium on Artificial Intelligence and Interactive Entertainment.*

Rabin, S. ed. 2002 *AI Game Programming Wisdom*. Hingham, Mass.: Charles River Media

Rabin, S. ed. 2003 *AI Game Programming Wisdom 2*. Hingham, Mass.: Charles River Media