

Article draft, please do not cite or distribute.

To appear as: Zubek, R. 2010. Needs-based AI. In Lake, A. (ed.), Game Programming Gems 8. Cengage Learning, Florence, KY.

Needs-Based AI

Robert Zubek

Intro

Needs-based AI is a general term for action selection based on attempting to fulfill a set of mutually competing *needs*. An artificial agent is modeled as having a set of conflicting motivations, such as the need to eat or sleep, and the world is modeled as full of objects that can satisfy those needs at some cost.

The core of the AI is an action selection algorithm which weighs those various possible actions against each other, trying to find the best one given the agent's needs at the given moment. The result is a very flexible and intuitive method for building moderately complex autonomous agents, which are nevertheless efficient and easy to understand.

This paper presents some technical details of needs-based AI. We begin with a general overview, and then proceed to dive directly into technical implementation, presenting both general information as well as some specific hints born out of experience implementing this AI. Finally, we finish with an overview of some design consequences of using this style of AI.

Background

In terms of its historical context, the needs-based AI approach is related to the family of *behavior-with-activation-level* action selection methods, common in autonomous robotics (for an overview, see [Arkin98], p. 141). In game development, it was also independently rediscovered by the Sims, where it has been enjoyed by millions of game players. The Sims also contributed a very useful innovation on knowledge representation, where behaviors and their advertisements are literally distributed “in the world” in the game, and therefore very easily configurable.

My own interest in this style of AI was mainly driven by working with the Sims (at Northwestern University, and later at EA/Maxis). I have since re-implemented variations on this approach in two other games: *Roller Coaster Kingdom*, a web business simulation game, and an

unpublished RPG from the *Lord of the Rings* franchise. I found the technique to be extremely useful, even across such a range of genres and platforms.

Unfortunately, very few resources about the original Sims AI remain available; of those, only a set of course notes by Ken Forbus and Will Wright¹, plus a Sims 2 presentation by Jake Simpson² are freely downloadable on the web at this point. My goal here is to present some of this knowledge to a wider audience, based on what I've gained from robotics and the Sims, as well as personal experience building such agents in other games.

Needs-Based AI Overview

There are many ways to drive an artificial agent; some games use finite-state machines, others use behavior trees, and so on. Needs-based AI is an alternative with an exciting benefit: the smarts for picking the next action configure themselves automatically, based on the agent's situation as well as internal state; yet the entire algorithm remains easy to understand and implement.

Each agent has some set of ever-changing needs that demand to be satisfied. When deciding what to do, the agent looks around the world, and figures out what can be done, based on what's in the area. Then it scores all those possibilities, based on how beneficial they are in satisfying its internal needs. Finally, it picks an appropriate one based on the score, finds what concrete sequence of actions it requires, and pushes those onto its action queue.

The highest-level AI loop looks like this:

- While there are *actions* in the queue, pop the next one off, perform it, and maybe get a reward
- If you run out of actions, perform *action selection* based on current needs, to find more actions
- If you still have nothing to do, do some *fallback* actions

That second step, the action selection point, is where the actual choice happens. It decomposes as follows:

1. Examine objects around you, and find out what they advertise
2. Score each advertisement based on your current needs
3. Pick the best advertisement, get its action sequence
4. Push the action sequence on your queue

The next sections will delve more deeply into each of these steps.

¹ http://www.cs.northwestern.edu/~forbus/c95-gd/lectures/The_Sims_Under_the_Hood_files/frame.htm

² <https://www.cmpevents.com/Sessions/GD/ScriptingAndSims2.ppt>

Needs

Needs correspond to individual motivations: for example, the need to eat, drink, or rest. The choice of needs depends very much on the game. The Sims, being a simulator of everyday people, borrowed heavily from Maslow's hierarchy (a theory of human behavior based on increasingly important psychological needs), and ended up with a mix of basic biological and emotional drivers. A different game should include a more specific set of motivations, based on what the agents should care about in their context.

Inside the engine, needs are routinely represented as an array of numeric values, which decay over time. In this discussion we use the range of [0, 100]. Depending on the context, we use the term 'need' to describe both the motivation itself (written in boldface, eg. **hunger**), or its numeric value (eg. 50).

Needs routinely have the semantics of "lower is worse and more urgent", so that **hunger=30** means "I'm pretty hungry," while **hunger=90** means "I'm satiated." Need values should decay over time, to simulate unattended needs getting increasingly worse and more urgent. Performing an appropriate action then refills the need, raising it back to a higher value.

For example, we simulate the agent getting hungry if they don't eat, by decaying the **hunger** value over time. Performing the "*eat*" action would then refill it, causing it to become less urgent (for a while).

Advertisements and Action Selection

When the time comes to pick a new set of actions, the agent looks at what can be done in the environment around them, and evaluates the effect of the available actions.

Each object in the world *advertises* a set of action/reward tuples – some actions to be taken, with a promise that they will refill some needs by some amount. For example, a fridge might advertise a "*prepare food*" action with a reward of +**30 hunger**, and "*clean*" with the reward of +**10 environment**.

To pick an action, the agent examines the various objects around them, and finds out what they advertise. Once we know what advertisements are available, each of them gets scored, as described in the next section. The agent then picks the best advertisement using the score, and adds its actions to their pending action queue.

Advertisement Decoupling

Please notice that the *discovery* of what actions are available is decoupled from *choosing* among them: the agent "asks" each object what it advertises, and only then scores what's available. The object completely controls what it advertises, so it's easy to enable or disable actions based on

object state. This provides great flexibility. For example, a working fridge might advertise “*prepare food*” by default; once it’s been used several times it also starts advertising “*clean me*”; finally, once it breaks, it stops advertising anything other than “*fix me*” until it’s repaired.

Without this decoupling, imagine coding all those choices and possibilities into the agent itself, not just for the fridge but also for all the possible objects in the world – it would be a disaster, and impossible to maintain.

On the other side of the responsibility divide, the agent can also be selective about what kinds of advertisements it accepts. We can use this to build different agent subtypes or personalities: for example, in a later section we will describe how to use advertisement filtering to implement child agents, with different abilities and opportunities than adults.

Advertisement Scoring

Once we have an object’s advertisements, we need to score them, and stack them against all the other advertisements from other objects. We score each advertisement separately, based on the reward it promises (eg. **+10 environment**), and the agent’s current needs. Of course it’s not strictly necessary that those rewards actually be granted as promised; this is known as *false advertising*, and can be used with some interesting effects, described later.

Here are some common scoring functions, from the simplest to the more sophisticated:

a. Trivial scoring

$$future\ value_{need} = current\ value_{need} + advertised\ delta_{need}$$

$$score = \sum_{all\ needs} (future\ value_{need})$$

Under this model, we go through each need, look up the promised future need value, and add them up. For example, if the agent’s **hunger** is at 70, an advertisement of **+20 hunger** means the future value of **hunger** will be 90; the final score is the sum of all future values.

This model is trivially easy, and has significant drawbacks: it’s only sensitive to the magnitude of changes, and doesn’t differentiate between urgent and non-urgent needs. So increasing hunger from 70 to 90 has the same score as increasing thirst from 10 to 30 – but the latter should be *much more important*, considering the agent is very thirsty!

b. Attenuated need scoring

Needs at low levels should be much more urgent than those at high levels. To model this, we introduce a non-linear attenuation function for each need. So the score becomes:

$$score = \sum_{all\ needs} A_{need} (future\ value_{need})$$

where A_{need} is the *attenuation function*, mapping from a need value to some numeric value. The attenuation function is commonly non-linear and non-increasing: starts out high when the need level is low, then drops quickly as the need level increases.

For example, consider the attenuation function $A(x) = 10/x$. An action that increases hunger to 90 will have a score of $1/9$, while an action that increases thirst to 30 will have a score of $1/3$, so three times higher, because low thirst is much more important to fulfill. These attenuation functions are a major tuning knob in needs-based AI.

You might also notice one drawback: under this scheme, improving hunger from 30 to 90 would have the same score as improving it from 50 to 90. Worse yet, worsening hunger from 100 to 90 would have the same score as well! This detail may not be noticeable in a running system, but it's easy to fix, by examining the need delta as well.

c. *Attenuated need-delta scoring*

It's better to eat a filling meal than a snack, especially when you're hungry, and worse to eat something that leaves you hungrier than before. To model this, we can score based on need level difference:

$$score = \sum_{\text{all needs}} (A_{need}(\text{current value}_{need}) - A_{need}(\text{future value}_{need}))$$

For example, let's consider our attenuation function $A(x) = 10/x$ again. Increasing hunger from 30 to 90 will now score $1/3 - 1/9 = 2/9$, while increasing it from 60 to 90 will score $1/6 - 1/9 = 1/18$, so only a quarter as high. Also, decreasing hunger from 100 to 90 will have a negative score, so it will not be selected unless there is nothing else to do.

Action Selection

Once we know the scores, it's easy to pick the best one. Several approaches for arbitration are standard:

- *Winner-takes-all*: the highest-scoring action always gets picked.
- *Weighted-random*: do a random selection from the top n (eg. top 3) high-scoring advertisements, with probability proportional to score.
- Other approaches are easy to imagine, such as a priority-based behavior stack.

In everyday implementation, *weighted-random* is a good compromise between having some predictability about what will happen, and not having the agent look unpleasantly deterministic.

Action Selection Additions

The model described above can be extended in many directions, to add more flexibility or nuance. Here are a few additions, along with their advantages and disadvantages:

a. *Attenuating score based on distance*

Given two objects with identical advertisements, an agent should tend to pick the one closer to them. We can do this by attenuating each object's score based on distance or containment:

$$score = D \left(\sum_{\text{all needs}} (\dots) \right)$$

where D is some distance-based attenuation function, commonly a non-increasing one, such as the physically-inspired $D(x) = x / distance^2$

However, distance attenuation can be difficult to tune, because a distant object's advertisement will be lowered not just compared to other object of this type, but also compared to all other advertisements. This may lead to a "bird in hand" kind of behavior, where the agent always prefers a much worse action nearby rather than a better one further away.

b. Filtering advertisements before scoring

It's useful to add pre-requisites to advertisements: for example, kids should not be able to operate stoves, so the stove should not advertise the "cook" action to them. This can be implemented in several ways, from simple attribute tests, to a full language for expressing predicates.

It's often best to start with a simple filter mechanism, because complex prerequisites are more difficult to debug when there are many agents running around. An easy prerequisites system could be as simple as setting Boolean attributes on characters (eg. *is-adult*, etc.), and adding an attribute *mask* on each advertisement; action selection would only consider advertisements whose mask matches up against the agent's attributes.

c. Tuning need decay

Agents' need levels should decay over time; this causes agents to change their priorities as they go through the game. We can tune this system by modifying need decay rates individually. For example, if an agent's hunger doesn't decay as quickly, they will not need to eat as often, and will have more time for other pursuits.

We can use this to model a bare-bones personality profile, eg. whether someone needs to eat/drink/entertain themselves more or less often. It can also be used for difficulty tuning: agents whose needs decay more quickly are harder to please.

d. Tuning advertisement scores

The scoring function can also simulate simple personality types directly, by tuning down

particular advertisement scores. To do this, we would have each agent contain a set of tuning parameters, one for each need, which modify that need's score:

$$\text{new score}_{\text{agent, need}} = \text{old score}_{\text{agent, need}} * \text{tuning}_{\text{agent, need}}$$

For example, by tuning down the **+hunger** advertisement's score, we'll get an agent that has a stronger preference for highly-fulfilling food; tuning up a **+thirst** advertisement will produce an agent that will happily opt for less satisfying drinks, and so on.

e. Attenuation function tuning

Attenuation functions map from low need levels to high scores. Each need can be attenuated differently, since some needs are more urgent than others. As such, they are a major tuning knob in games, but a delicate one because their effects are global, affecting all agents. This requires good design iterations; but analytic functions (eg. $A(x) = 10/x$) are not easy for designers to tweak or reason about.

A happy medium can be found by defining attenuation functions using piecewise-linear functions (ie. point pairs that define individual straight-line segments, rather than continuous, analytic formulas). These can be stored and graphed in a spreadsheet file, and loaded during the game.

Action Performance

Having chosen something to do, we push the advertisement's actions on the agent's action queue, to be performed in order. Each action would routinely be a complete mini-script. For example, the stove's "*clean*" action might be small script that:

- Animates the agent getting out a sponge, scrubbing the stove
- Runs the animation loop, and an animated stove condition meter
- Grants the promised reward

It's important that the actual reward be granted manually as part of the action, and not be awarded automatically. This gives us two benefits:

- Interrupted actions will not be rewarded.
- Objects can falsely advertise, and not actually grant the rewards they promised.

False advertisement is an especially powerful, but dangerous option. For example, suppose that we have a food item that advertises a hunger reward, but doesn't actually award it. A hungry agent would be likely to pick that action – but since they got no reward, at the next selection point they would again likely pick, and then again, and again. This quickly leads to very intriguing "addictive" behaviors.

This may seem like a useful way to *force* agents to perform an action. But it's just as hard to make them stop once they've started. False advertisements create action loops that are very difficult to tune. In practice, forcing an action is easier done by just pushing the desired action on the agent's action queue.

Action Chaining

Performing a complex action such as cooking a meal usually involves several steps (such as preparation and cooking), and several objects (a fridge, a cutting board, a stove). This sequence must not be atomic – steps can be interrupted, or they can fail due to some external factors.

Complex sequences are implemented by chaining multiple actions together. For example, eating dinner might decompose into several separate actions:

- Take a food item from the fridge.
- Prepare the food item on a counter.
- Cook the food item on the stove.
- Sit down and eat, thereby getting a hunger reward.

It would be suboptimal to implement this as a single action; there is too much variability in the world for it to always work out perfectly.

We can create action sequences in two ways. The simpler way is to just manufacture the entire sequence of actions right away, and push the whole thing on the agent's queue. Of course these steps can fail, in which case the remaining actions should also be aborted. For some interesting side-effects, aborting an action chain could create new actions in its place; for example, a failed "cook food" action sequence could create a new "burned food" object that needs to be cleaned up.

The second method, more powerful but more difficult, is to implement action chaining by "lazy evaluation." In this approach, only one action step is created and run at a time, and when it ends, it knows how to create the next action and front-loads it on the queue.

For an example of how that might look, consider eating dinner again. The refrigerator's advertisement would specify only one action: "*take food*". That action, towards the end, would then find the nearest kitchen counter object, ask it for the "*prepare food*" action, and load that on the queue. Once "*prepare food*" was done, it would find the nearest stove, ask it for a new "*cook food*" action, and so on.

Lazy action chaining makes it possible to modify the chain based on what objects are available to the agent. For example, a microwave oven might create a different "*cook food*" action than a stove would, providing more variety and surprise for the player. Second, it makes interesting failures easier. For example, the stove can look up some internal variable (eg. repair level) to determine failure, and randomly push a "*create a kitchen fire*" action instead.

In either case, using an action queue provides nice modularity. Sequences of smaller action components are more loosely coupled, and arguably more maintainable, than standard state machines.

Action Chain State Saving

When an action chain is interrupted, we might want to be able to save its state somehow, so that it gets picked up later.

Since all actions are done on objects, one way to do this is to mutate the state of the object in question. For example, the progress of “*cleaning*” can be stored as a separate numeric *cleanness* value on an object, which gets continuously increased while the action is running.

But sometimes actions involve multiple objects, or the state is more complicated. Another way to implement this is by creating new *state objects*. An intuitive example is food from the original Sims: the action of prepping food creates a “prepped food” object, cooking then turns it into a pot of “cooked food”, which can be plated and turned into a “dinner plate”. The state of preparation is then embedded right in the world: if the agent is interrupted while prepping, the cut up food will just sit there, until the agent picks it up later and puts it on the stove.

Design Consequences of Needs-Based AI

With the technical details of needs-based AI behind us, let’s also consider some of the design implications of this style of development, since it’s different from more traditional techniques.

First of all, the player’s experience with this AI really benefits from adding some feedback to the agents. Developers can just look at the internal variables, and immediately see: the agent is doing this because it’s hungry, or sleepy, or other such. But the player will have no such access, and is likely to build an entirely different mental model of what the agent is doing. Little bits of feedback, like thought bubbles about what needs is being fulfilled, are easy to implement, and go a long way towards making the system comprehensible to the player.

Second point is about tuning: some of the tunable parameters have global effect, and are therefore very difficult to tune after the game has grown past a certain size. The set of needs, their decay rates, score attenuation functions, and other such elements, will apply to all characters in the game equally, so tuning them globally requires a lot of testing and a delicate touch.

If a lot of variety is desired between different parts of the game, it might be a good idea to split the game into a number of smaller logical partitions (levels, etc.) and have a different set of those tunable parameters, one for each partition. Ideally, there would be a set of global tuning defaults, which work for the entire game, and each partition could specifically override some of them as needed. Partitioning and overriding tuning values buys us greater flexibility, although at the cost of having to tune each partition separately.

Third, this AI approach tends heavily towards simulation, and makes it hard to do scripted scenes or other triggered actions. Imagine implementing some actions on a trigger, such as having the agent approach the player when he comes into view. One might be tempted to try to implement that using just needs and advertisements, but the result will be brittle.

If particular one-off scripted behaviors are desired, it would be better to just manually manufacture appropriate action sequences, and forcibly push them on the agent's action queue. But in general, this overall approach is not very good for games that need a lot of triggered, scripted sequences (eg. shooter level designs). Needs-based AI works better for simulated worlds, rather than scripted ones.

Final Words

Needs-based AI is computationally very efficient; only a trivial amount of the CPU is required to pick what to do, and to handle the resulting action sequence. The system's internals are very easy to understand, by just inspecting the agent's internal needs values one can get a good idea of why it does what it does. And by externalizing the set of possible actions into the world, the system also achieves great modularity: the AI can be "reconfigured" literally by adding or removing objects around the agent.

In spite of unusual design consequences, the needs-based approach is very capable, easy to implement, and effective at creating good characters. It's a powerful tool for many situations.

Acknowledgments

Thanks to Ken Forbus and Richard Evans, from whom I've learned most of what I know about this style of AI.

References

[Arkin98] Arkin, R., *Behavior Based Robotics*, MIT Press, 1998.