48 Implementing N-Grams for Player Prediction, Procedural Generation, and Stylized AI

Joseph Vasquez II

48.1 Introduction
48.2 N-Grams Understood
48.3 N-Grams Implemented
48.4 N-Grams in Your Game
48.5 Future Research
48.6 Conclusion

48.1 Introduction

AI learning provides exciting possibilities for the next generation of games. As game systems increase in computational power, the fanciest online learning techniques become more feasible for the games of tomorrow. However, until we get there, the AI programmers of today need solid techniques that provide good results with minimal cost. N-grams are a powerful and computationally inexpensive learning technique that can provide adaptive AI in the game you are working on right now [Laramée 02].

N-grams store statistics on behavioral patterns, which simulates learning a style. By exposing an N-gram to a history of player actions, it can predict the player's next move with surprising accuracy—in fact, for certain types of games (such as *Mortal Kombat*-style fighting games) they can work so well that they have to be toned down in order to give the player a chance to win [Millington 06]! This learning can allow your cooperative AI to become more helpful, and your enemy AI to offer a more personalized challenge. By exposing an N-gram to a designer's creation patterns, it can procedurally generate content that mimics the designer's style. The code provided on this book's website (http://www.gameaipro.com) includes a well-optimized N-gram implementation, which you are free to use in your game. Hopefully, this will allow you to easily evaluate just how helpful N-grams can be for you.

This article focuses on getting an N-gram up and running in your game. First we'll go over what N-grams are and how they make predictions. Next we'll cover implementation specifics and optimizations, while aiming for the most accurate predictions possible. Finally, we'll discuss in-game usage of N-grams, and the issues that pop up during integration. At the end, you'll have an N-gram library that you can drop in your game, and you'll know how to use it effectively.

48.2 N-Grams Understood

Before we can use them, we need to know what N-grams are and how they make predictions.

48.2.1 Examining Probability: A Likely Story

Probability can be defined as the likelihood that a given event will occur. Mathematically, we define it as a ratio of the number of ways an event can occur divided by the number of possible outcomes. This ratio is strictly between 0 and 1, so we treat it as a percentage, as shown in Equation 48.1.

$$P(event\ e) = \frac{\#\ of\ \ ways\ e\ can\ happen}{\#\ of\ \ all\ possible\ outcomes} \tag{48.1}$$

Let's assume that a player has three move options: Jump, Attack, and Dodge. Without any other information, the probability of choosing Jump would be P(Jump) = 1/3 = 0.33 = 33%. Informally, we take this to mean that Jump occurs 33% of the time on average. In 100 moves, we expect Jump to occur on average 33 times. Working backwards from this understanding, if we're given a sequence of 100 events in which Jump occurred 33 times, the probability of Jump occurring again is assumed to be 33%, as shown in Equation 48.2.

$$P(event\ e) = \frac{\#\ of\ \ times\ e\ occurred}{\#\ of\ \ times\ any\ event\ occurred} \tag{48.2}$$

This is useful for deriving probability from a sequence of past events. From this history, we can predict that the next event is most likely the event that has occurred most often.

In this sequence: "Jump, Jump, Dodge, Attack, Dodge, Attack, Jump, Jump, Dodge," P(Jump) = 4/9 = 44%, P(Attack) = 2/9 = 22%, and P(Dodge) = 3/9 = 33%. Based on raw probability, the player will jump next. Do you agree with this prediction? Probably not, since you've likely picked up on a pattern. N-grams are used to find patterns in sequences of events. An N-gram could predict that a "Dodge, Attack" pattern was being executed. N-grams provide predictions more accurately than raw probability alone.

48.2.2 Enter N-grams

Each N-gram has an order, which is the *N value*. 1-grams, 2-grams, and 3-grams are called *unigrams*, *bigrams*, and *trigrams*. Above that, we use 4-gram, 5-gram, and so on. The *N* value is the length of patterns that the N-gram will recognize (the N-tuples). N-grams step through event sequences in order and count each pattern of *N* events they find. As new events are added to the sequence, N-grams update their internal pattern counts.

Consider a game where we need to predict the player's next move, Left or Right. So far, the player has moved in the following sequence: "R, R, L, R, R". From this, we can compute the following N-grams:

- A unigram works the same as raw probability; it stores: R: 4, L: 1.
- A bigram finds patterns in twos, "RR, RL, LR, RR", and stores: RR: 2, RL: 1, LR: 1.
- A trigram finds patterns by threes, "RRL, RLR, LRR", and stores each pattern once.
- A 4-gram will store RRLR: 1 and RLRR: 1.
- A 5-gram will store the entire sequence once.
- There isn't enough data yet for an N-gram with order greater than 5.

Notice that we store occurrence counts rather than probabilities, which requires less computation. We can perform the divisions later to calculate probabilities when we need them. With patterns learned, our N-grams can now predict the next event.

48.2.3 Predicting with N-grams

A quick note for your math teacher: Technically, N-grams only store statistics and cannot make predictions. In implementation, however, we usually wrap the storage and functionality together. With this in mind, we'll treat our N-grams as the entire prediction system for convenience.

An N-gram predicts by picking out an observed pattern that it thinks is being executed again. To do so, it considers the most recent events in the sequence, and matches them against previously observed patterns. This set of recent events we'll call the *window*. (See Figure 48.1.)

The length of the window is always *N*-1, so each pattern includes one more event than the window length. Patterns "match" the window if their first *N*-1 events are the same. All matching patterns will have a unique *N*th event. The pattern with the most occurrences has the highest probability, so its *N*th event becomes the prediction. No division is required.

Let's see what our N-grams will predict next in the same sequence, "R, R, L, R, R." (See Table 48.1.)

Using N-gram statistics, we can make the following predictions:

- The unigram predicts Right, since it uses raw probability. Its window size is $N - 1 = 0$.
- The bigram has observed two patterns that match its window, RR and RL. Since RR occurred once more than RL, the bigram chooses RR and predicts Right.

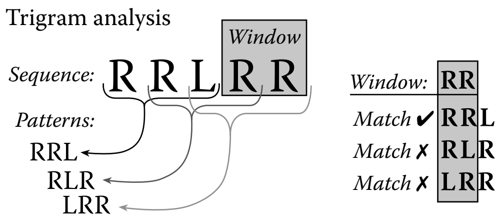

Figure 48.1

The trigram window and observed patterns of the sequence "R, R, L, R, R."

Table 48.1 Assorted N-gram statistics for the sequence, "R, R, L, R, R"

	Unigram	Bigram	Trigram	4-gram
Window	None	R	RR	LRR
Patterns observed	5	4	3	2
Window matches	R, L	RR, RL	RRL	None
Most occurrences	R: 4	RR: 2	RRL: 1	N/A
Prediction	Right	Right	Left	None

- The trigram finds that only RRL matches its window, so it uses RRL to predict Left.
- The 4-gram has not observed any patterns that start with LRR, so it cannot make a prediction.

In order to make a prediction, N-grams must consider the occurrence counts for all patterns that match the current window. An efficient data structure will facilitate fast pattern lookups. We'll explore good storage options in the implementation section.

48.2.4 Probability of Event Patterns

Equation 48.2 uses occurrence counts of events. N-grams above order 1 count patterns, not events. To calculate pattern probabilities, we need a new formula generalized for any *N* value.

You don't need to calculate probabilities in order to find the most probable event, as seen previously. However, the probability can be useful in many ways. For example, you might want your AI to ignore a prediction if the probability is too low. As another example, you may want to check the probability of a very meaningful event. Even if the most probable event is Friendly_Handshake with 55%, you might find it useful to know that the probability of a Backstab event is 43%.

Since we are only considering patterns that match the current window, we are now calculating conditional probabilities. This makes the probability of an unmatched pattern equal to 0%. Each matching pattern has a unique *N*th event. The probability that an event *e* will occur next is equal to the probability of the matching pattern containing *e* as its *N*th event, as shown in Equation 48.3.

$$P(event\ e\ is\ next) = P(matching\ \ pattern\ ending\ in\ e\ is\ next) \tag{48.3}$$

The probability of a matching pattern is its occurrence count divided by the total occurrences of all matching patterns (all patterns whose first *N*–1 events match the window), as shown in Equation 48.4.

$$P(matching\ \ pattern\ mp) = \frac{\#\ of\ \ mp\ occurrences}{Sum\ of\ \ all\ matching\ \ pattern\ occurrences} \tag{48.4}$$

During prediction, you'll likely check the occurrences for all matching patterns to find the max. You can sum them while you're at it, and find the denominator in the same pass.

If needed, you can calculate the probability that an event pattern will occur again anytime in the future, as shown in Equation 48.5. This is a nonconditional probability, as it doesn't use the current window.

$$P(\textit{event pattern ep}) = \frac{\#\ \textit{of ep occurrences}}{\textit{Sum of all pattern occurrences so far}} \tag{48.5}$$

With the length L of the event sequence, you can calculate the denominator directly:

$$\textit{Total pattern occurrences } T = L-(N-1) \tag{48.6}$$

It is simply the sequence length minus the first window. Note that T will be negative when L is less than the window size. If $T < 0$, this means that $|T|+1$ more events are required for a single N length pattern to occur.

48.3 N-Grams Implemented

We now know how N-grams work. Let's cover implementation details to get them ready for a game.

48.3.1 Making a Prediction: I Choose You

You can calculate probabilities for each possible event, but how do you choose one to predict? For example, if a particular N-gram calculated that Right occurred 70% of the time and Left occurred 30% of the time, do you predict Right or Left? You can naively choose the most probable event, in this example Right; however, this isn't your only option. You can tweak your prediction accuracy by varying the selection. There are many ways to do this, such as occasionally choosing a less probable event or choosing an event based on a distribution of probabilities, known as a *weighted random*.

In a weighted random, the percentages calculated by the N-gram for each event give you the percentage chance that you will predict that event. Continuing with the example, to make a prediction, we would roll the dice and predict Right 70% of the time and Left 30% of the time. In this way, the AI prediction doesn't always blindly choose the most probable prediction, and the AI itself becomes less predictable.

48.3.2 Issues with Large Numbers of Events: Show Me Your Moves

Rock-Paper-Scissors only has three possible events. Most games use a lot more than three. Increasing this number has a big effect on an N-gram's storage needs and prediction accuracy.

From combinatorics, each pattern is a permutation of events without repetition. So, with E possible events, E^N total patterns of length N are possible. An N-gram will need to keep counts for E^N different patterns. If $E = 10$ and $N = 6$, $E^N = 1$ million patterns. Some patterns may never occur, especially with large E values. You don't need to store counts for unseen patterns.

Higher order N-grams count longer patterns, so they require longer event sequences to learn from. Larger E values create more pattern possibilities, which raises the sequence length required before an N-gram can make accurate predictions. N-grams learn a style from an event sequence. They need enough exposure to learn the style well. You can't always decrease E, so use the smallest N value possible that still achieves good prediction accuracy.

48.3.3 Adding Robustness

Let's cover a couple situations that will stump an N-gram's predictions in a real game. First we'll describe these two problems, and then we'll offer a solution to address them both. This solution is used in the N-gram library provided with this book.

48.3.3.1 Prediction Failures: Heads or Tails Never Fails

In a simple coin toss, will an N-gram predict heads or tails as the first move? There are no events in the sequence yet, so it will fail to predict anything.

Prediction failures occur when there is no pattern matching the current window. This is always the case when the sequence length L is smaller than N. An N-gram will have to handle this somehow. One method is to report a failure, which nobody likes to check for. A better method is to design the implementation such that it never fails, which we will do below.

48.3.3.2 Unseen Events: Expecting the Unexpected

In a game of *Rock-Paper-Scissors*, a player has made the following moves: "Rock, Paper, Rock, Paper, Rock, Paper." Our N-grams would predict Rock next, and suggest you throw Paper. Assuming a tricky player, what do you think is coming next? Scissors, right?

So far, our N-grams only know of events they've seen. This lowers probability accuracy when the event sequence is too short or unvaried. For more accurate probabilities, you should factor in the possibility that unseen events can occur.

48.3.3.3 Unseen Events: Free Occurrences to the Rescue

With a simple trick, we can provide probabilities for unknown events, and solve the failure problem mentioned previously; we give a single freebie occurrence to each possible pattern.

Let events be represented by an integer ranged [0, E), where E is the number of different event types. Enumerations work well. Give the constraint that E must be provided to our N-gram at initialization. If we need to predict an unseen event, we can simply pick any integer in this range, without caring which game event it even represents.

Actually storing the freebies would be a waste of memory. Instead, we can make them implicit. After looking up an occurrence count, we just add 1 to the count in any calculation. If the pattern is not stored, it's unseen, and we treat the count as 0 + 1.

We need new probability formulas to account for the implicit freebies. There are always E patterns matching the window, so the probability of the next event becomes:

$$P(matching\ \ pattern\ mp) = \frac{\#\ of\ \ mp\ occurrences\ +1}{Sum\ of\ \ all\ matching\ \ pattern\ occurrences\ +E} \tag{48.7}$$

For the probability of an event pattern occurring in the future, remember that there are E^N total patterns possible, and each gets a freebie. Using Equation 48.6, the probability is now:

$$P(event\ \ pattern\ ep) = \frac{\#\ of\ \ ep\ occurrences\ +1}{T + E^N} \tag{48.8}$$

If you really want the probability of an event e occurring in the future, you need the sum of all pattern occurrences that ended in e. However, don't start traversing through all your patterns. Each time an event *e* occurred, it completed a pattern, right? Yes, aside from in the first $N - 1$ events. You can keep a separate list of single event occurrences (a unigram), and a copy of the first window. Don't forget to add in a freebie for all patterns that end in *e*; there are E^{N-1} of them. The future occurrence probability is now:

$$P(event\ e) = \frac{\#\ of\ e\ occurrences - \#\ of\ e\ in\ first\ window + E^{N-1}}{T + E^{N}} \tag{48.9}$$

For the "Rock, Paper, Rock, Paper, Rock, Paper" sequence, our old trigram using Equation 48.4 would have predicted Rock 100%, Paper 0%, and Scissors 0%. Our new trigram using Equation 48.7 will predict Rock 60%, Paper 20%, and Scissors 20%. We've toyed with the probability values by stealing from the rich to give to the poor. This weakens the integrity of the values perhaps, but we can rest assured that they will not change in order from most probable to least. Also, we won't fail if asked to predict the probability of Scissors, which is unseen.

48.3.4 Computation Time: A Need for Speed

At maximum, there will be *E* matching patterns for a given window. Each time we make a prediction, we need to access the stored occurrence counts for all of them. How expensive are all these searches? Naturally, it depends on your data structure.

The good news is that we can optimize until we only need a single search. If we organize our storage in such a way that *all window matching patterns' counts are stored consecutively in sorted order*, then we'll only need to find the first pattern in a matching set. Afterwards, we can simply increment through the rest for free. A sorted array would perform the search at $O(\log_2 T)$, while a hash table would provide O(1) as usual. A structure such as a prefix tree could provide lookups at $O(N \log_2 E)$ complexity, (where *N* is the *N* value), which is pretty good, and is used in the code library provided with this book. Both predictions and Equation 48.7 calculations require the same search, so it is recommended to cache the search result. In practice, probabilities are usually requested with the same window as the prediction (if they are requested at all). If we get a probability request, we can cache the denominator to speed up subsequent requests.

Lookups will also need to be performed whenever a new event is added to the sequence. This will involve incrementing a stored count or inserting a new pattern altogether. As a further optimization, we can keep a max occurrences index and a total count value for each set of matching patterns. Any time we search to add a pattern or increment a pattern's count, we can take the opportunity to update these values. This keeps us from having to walk through all the matching patterns in the set during our next prediction or probability calculation. This has a maximum additional storage requirement of $2E^{N-1}$ values.

In typical usage, you can expect to perform two searches per update: one for updating your N-gram with the latest event, and another for predicting the next event. Any probability requests are free when using the optimizations above.

48.3.5 Ensuring Accurate Predictions: Self-Accountability

On each update, an N-gram is usually provided with an event that occurred since the previous update. This usage allows the N-gram to check if its last prediction was correct

Listing 48.1. A pseudocode algorithm for finding which *N* value provides the highest prediction accuracy for pre-existing event sequences.

```
Instantiate an N-gram for each candidate N value ranged [1,20]
Create accuracy result storage for each N-gram
For each event sequence,
    For each event e ∈ [0,L-1] in the sequence,
        For each N-gram ng,
            store result of e compared to ng.Predict()
            ng.UpdateWithNewEvent(e)
output accuracy results
output N value of most accurate N-gram based on results
```

or not, and provide feedback on its running accuracy for the current sequence. If the accuracy is performing below expectations, your AI can find out about it and switch to plan B. Alternatively, you might ignore predictions until the accuracy reaches an acceptable level.

48.3.6 Which N-Value Provides the Best Prediction?

In practice, choosing the right order for an N-gram is often done by starting with a guess, experimenting, and then adjusting [Millington 06]. With the help of some trustworthy event sequences, we can automate the process of finding which *N* value is right for you. Listing 48.1 shows an example for how to algorithmically find the best *N* value to use based on observed data.

Remember that larger *N* values require increased processing time and memory storage. It's probably best for a human to review the outcomes in order to pick the best compromise of lowest *N* value and good prediction accuracy. You could have each N-gram output its memory usage at the end of each sequence to aid your decision. Upon close inspection, you may find that the most accurate *N* value changed over certain stretches of the sequence. In this case, using multiple N-grams in parallel may offer the best prediction accuracy in your game.

The sequences used must be good representations of real gameplay, so you'll want to write a method to record event sequences from a designer's playthrough. If the new N-gram will alter the next playthrough, you have a feedback loop. Be sure to run through the process again until you're sure your *N* value is well chosen.

The N-gram code provided with this book includes an implementation of the algorithm in Listing 48.1.

48.4 N-Grams in Your Game

Our N-gram is optimized and ready, but there are lots of considerations for integration in a game.

48.4.1 An Event by Any Other Name

N-grams work with events without caring what they really are. Only your game knows what each event actually stands for. They aren't really going to be Rock, Paper, and Scissors.

It's up to you to decide which types of incidents should constitute a unique event in an N-gram. A well picked list will lend itself well to pattern forming, so this is an important task. You may need to play with different approaches to get good results. Consider the different ways you can view events in your game.

For traditional fighting games using complex button combinations, N-grams can use raw controller input to easily predict which combo the player is going to execute. For fighting games without complex button combos, raw controller input may not suffice. It may be better to tokenize these inputs into completed actions, which then become events. Don't lose too much information while tokenizing. A sequence of "Jump, Kick, Land" allows the AI to intervene, while a sequence of only "Jumpkick" does not.

Players often interrupt their course of action in order to respond to the AI. A well-timed kick by the AI may cause the player to perform "Dash, Block" instead of "Dash, Attack." Adding certain AI actions into the event sequence may provide more context to pattern learning. Consider how AI actions affected the sequence.

For RTS games, issued commands can work well as events. Consider grouping commands by type and using separate N-grams for each group. A sequence of "Tease_Oliphaunt, Release_Oliphaunt" should not be separated by several Redundantly_Keep_Miner_Busy commands. Even if the player was spam-clicking on miners, that doesn't need to interrupt the important oliphaunt-angering pattern. You might decide to ignore mining actions altogether. Instead, you could fire an event whenever the player becomes rich enough to build a nontrivial unit. Think about which actions really make a difference.

For uses beyond player prediction, such as procedural generation, only you can know best. Look for branch points and designate events for choices that could define a style. For data using floating-point numbers, try to quantize the data into differentiable thresholds. Turning 2.2, 3.2, 4.2, 6.2 into "Increase1, Increase1, Increase2" creates a pattern you could use.

Your AI needs to determine when an event has occurred. Decide whether to observe state transitions, or rely on messages being received, or some other mechanism.

48.4.2 Predicting the Impossible

In practice, there are event patterns that will never happen. "Sheath_Sword, Draw_Sword" is possible, but "Sheath_Sword, Sheath_Sword" is not. Based on your selection algorithm, your N-gram may predict an impossible pattern. Modifying your N-gram implementation to circumvent this is probably unnecessary. If absolutely needed, your AI can check the prediction returned from the N-gram against a list of invalid patterns before using it.

48.4.3 Dealing with Time Delays

In fighting games like *Street Fighter II*, some special moves require crouching for two seconds before attacking. If the player's last move was crouch, your N-gram must decide between predicting a special move or a normal crouching move. If normal moves are more common, your AI will probably fall victim to the special move every time. Your players will pick up on this very quickly; a golden path is to low-kick twice, then special move once.

This situation might benefit from using time delays as an event. A player doing "Crouch, TimeSecond1, TimeSecond1" will help your N-gram predict that a special attack is forthcoming. Use time events with care, as many time delays can occur unintentionally in normal gameplay. Irrelevant time events sprinkled throughout a sequence will break apart meaningful patterns and wreak havoc on prediction accuracy.

48.4.4 Let's Take This Offline

N-gram learning can take place online or offline. Offline means an event sequence is created during development and the N-gram starts the game with patterns already observed. If you don't allow the N-gram to learn at runtime, it will stick to the style you trained it with during development. The storage requirement will also be fixed, and no update searches are needed.

Online means new events are added to the sequence at runtime. This allows your N-gram to learn new patterns from gameplay. As with any runtime learning technique, this gives your AI the ability to adapt to situations that you haven't anticipated, which should excite you. This includes situations that were never tested, which should scare the crap out of you. Worse still, as shown by the *Street Fighter II* example above, if your players figure out how the AI works then they may be able to find ways to exploit it.

Be sure to keep tight control over the extents of runtime learning. Players are accustomed to outwitting AI; taking advantage of NPCs is a common game mechanic. We want players to do this within the limits we set, so that we can be sure their experience is entertaining. Unearthing a golden path or a crash bug is not the entertainment you want. Your testers should be informed about the learning so they can hammer on it with the appropriate variety. In addition, they should test the learning in very long sessions, to catch problems such as your statistics overflowing, your sequence growing too long, or your pattern storage using too much memory.

48.4.5 Mimicking Style and Procedural Generation

We've mentioned that N-grams can give style to AI, but what does that really mean? Until now, all our examples have been based on recognizing styles in order to predict events, and indeed, this seems to be the most common use in games. N-grams can be trained to control AI according to a consistent style. For example, N-grams can perform procedural generation. Predicting events and choosing actions from a style are functionally equivalent for an N-gram. The real differences are in how and when we use the predictions, event sequence, and window.

We've defined the window to be a set of the last events in the sequence, but this need not be the case. Once an N-gram learns from an event sequence, it can accept any set of N–1 events as a window and predict the next event. We can treat the prediction as a creative choice or action. Decoupling the event sequence from the window allows us to separate learning and predicting, like studying a role and then performing it.

We can think of the event sequence as a cookbook that our N-gram memorized, the window as a set of ingredients on hand, and the prediction as the N-gram's completed dish. Present the N-gram with different sets of ingredients, and it will whip up dishes based on how it learned to cook from the event sequence. We could even feed a dish back as an ingredient in the next window, allowing the N-gram to feed off its own output and drive its own patterns.

Say we are building trees for our game using a collection of building blocks, including Start, Trunk, Fork, Branch, Twig, Leaf, and Done. We pick an ordered sequence of: "Start, Trunk, Fork, Branch, Twig, Done, Branch, Twig, Done," and a symmetrical tree appears with a split trunk and a branch and twig on either side. If we were to use this sequence to train a bigram, it would know how to create a tree identical to ours. We could give it the window, “Start,” and it would choose, “Trunk.” We could return, “Trunk,” and it would choose, “Fork.” By repeatedly returning its choice as the next window, the bigram would clone our tree. If we built a few different trees with these options, the bigram would create one tree with the most common traits we used. If we added a bit of randomness to its prediction accuracy, it would create a variety of trees in our style, which could populate our game. Accurate and deterministic predictions can work against the N-gram’s creativity. There is a time to be creative, and a time to be accurate.

If we restricted prediction of unseen events, the bigram would never create trees with leaves. If unrestricted, it would gently toy with our style by adding a few leaves. It might also try absurd variations such as trunks growing on twigs, so we’d need to watch its output for invalid choices.

Many types of procedural generation can be performed, such as playing music or writing text. You can train an N-gram with the complete text of a book, such as *Green Eggs and Ham*. In this case the text is called the *corpus*. Afterwards, your N-gram can write text to mimic the unique style learned from the corpus. You can play with this in a demo provided on the book’s website. You can use N-grams in a game or in a development tool. You can use them here or there. You can use them anywhere.

It’s important to note that in the examples in this section, the N-gram is not predicting and learning at the same time. Once your N-gram has learned the style you want, it’s time to perform. It can improvise when needed, but it shouldn’t be changing its act. Train it up in the way it should behave, and then lock it down so that it won’t stray from its style.

48.4.6 Weighting Old versus Recent Events

Thus far our N-grams have remembered each event occurrence equally. For living beings, some actions are more memorable than others. Will it increase prediction accuracy to weight some event occurrences more than others? Applying weights to specific event types will require game-specific information. Weighting events based on time of occurrence could be useful in general.

Applying more weight to older data creates a prediction bias toward the earliest events seen. If an “Approach, Kick” pattern occurred at the beginning of the sequence, an N-gram would still be suspicious of another kick, even if multiple “Approach, Hug” patterns occurred later. This might be utilized as a technique for learning unique styles, such as a puppy whose first impression of someone may forever influence further interaction. If we want accurate predictions, however, then this won’t do it. Applying more weight to newer events sounds more promising.

48.4.6.1 Changing with the Times

With longer event sequences, N-grams can better learn the patterns that identify a style. If that style changes, a well-trained N-gram will have difficulty catching up with the times. An N-gram won’t predict a new trend until the new patterns obtain higher occurrences than the past patterns. The longer the sequence, the more confident and less flexible an N-gram becomes. This situation arises often when performing player prediction, since players often change their play style. Their style might change upon obtaining a new item, gaining competency in a new ability, or upon simply learning to play the game better. We need a solution that can keep N-grams focused on recent events.

48.4.6.2 Limiting Memory: You Must Unlearn What You Have Learned

If you give your N-gram a limited sequence memory, it will forget old events. This is an all-or-nothing form of weighting recent events, by discarding them once they become stale. You can do this by keeping the event sequence in a queue with a limited size. When the queue is full, each new event will push an old event out. When an old event leaves the queue, you decrement the occurrence counts of matching patterns, thereby forgetting that the event ever occurred. If the entire event sequence is already being stored by another system, your N-gram can keep pointers into it rather than storing its own deep copied queue. This limited memory technique is easy to implement, and can improve prediction accuracy for many games.

When using this technique, picking a good queue size is important. Patterns that occur less frequently than the queue size will not accumulate occurrence counts. This creates a weakness in which uncommon event patterns are never expected. For example, in an RTS, a player might always target the AI's factory with a nuke. Nukes take a long time to build, during which many other events occur. By the time the player's nuke is ready for launch, the N-gram will have forgotten the result of the last nuke, and it won't have a clue where to expect the blast. You might solve this problem by keeping minimum occurrences for all patterns, which would also keep the N-gram from repeatedly adding and removing patterns to and from storage.

48.4.6.3 Focusing on Recent Activity: What's Trending Now

Instead of forgetting old events, another option is to focus on new trends. Your N-gram can keep a small queue of recently completed patterns. This hot-list can be referenced when making predictions, in order to favor the probability of recent activity. When calculating a pattern's probability, a bonus is given if that pattern is currently in the hot-list. This affects caching, since the pattern with the most occurrences may not be the most probable after hot-list bonuses are factored in.

Patterns leave the hot-list over time. If patterns occur when they are already on the hot-list, they are reinserted to remain longer and they get bigger probability bonuses. This helps avoid the problem of forgetting old data altogether, but the extra calculations and hot-list lookups increase processing time during predictions and learning. It's a trade-off you'll only want to explore if needed for your game.

48.4.7 With Great Power Comes Great Responsibility

Memorizing boss patterns and learning their reactions have been the key to victory in many games for over 20 years. N-grams give your AI the power to throw all that back in a player's face. Imagine playing a version of *Punch-Out!!* in which the AI opponents could start predicting your counter-attacks and changing their combos in the middle of the bout. Exciting, yes, but talk about brutal!

N-grams can easily become too powerful [Millington 06]. If your game naturally lends itself well to N-gram prediction, you will probably run into this problem. Rather than trying to hinder the N-gram's prediction prowess, you can have another AI layer intercept the predictions and decide how best to use them. The benefit of the interception layer is that you can still depend on accurate predictions. Rather than being unsure if the player is going to punch or dodge, you can be sure of the punch and have the AI opponent lower its defenses just in time. Accurate predictions are useful for balancing difficulty in either direction.

48.5 Future Research

The following areas are ripe for future research.

48.5.1 Probabilities That Make Sense

We've taken some extreme liberties with probability calculations. Ours will correctly order patterns from least to most probable, but their actual values aren't very meaningful. A good addition would be to factor in error estimation in order to provide meaningful probabilities. Error estimation will likely use the *N* and *E* values, sequence length *L*, and additional information.

48.5.2 Time Exploration

We mentioned the use of a time delay as an event, in order to differentiate patterns that rely on time. The big problem is that unintentional time events occur often, and spamming the sequence with irrelevant time events wreaks havoc on learning. Further exploration on how to use time information would be helpful. A good starting point might be to take a parameter when a new event is added to the sequence, or whenever a prediction is requested. These timestamps might be used in conjunction with feedback to internally discover patterns that rely on a time delay.

48.5.3 What's in an Event?

This article has focused on N-gram implementation and game integration. Event selection is an important consideration that is very game-specific. A survey of various games with real event lists would be a great complement, and would provide more insight on when N-grams are appropriate.

48.5.4 Player Imitation: Jockin' My Style

In the *Forza* series, the Drivatar mode is able to mimic the player's driving style with neural nets. However, for games that can use it, N-grams offer a simpler statistical method to mimic a player's style.

After recording an adequate number of player events, the player's style can be stored as an N-gram's statistics. The N-gram can then be used to control the AI, by using predictions as move choices. This could provide many fun gameplay experiences.

Imagine a game that features cooperative AI. Rather than configure the AI with parameters, a player could perform a playthrough that teaches the AI how to act. If the game allowed sending N-gram statistics to other players, they could play the game with an AI that mimics the experience of playing alongside their friend.

Super Street Fighter IV: 3D Edition allows players to form teams of fighters in the game's Figurine Mode. These teams are sent to other players via the Nintendo 3DS's StreetPass function, after which each player is notified of a battle outcome decided by fighter selection. Such a mode could benefit from N-grams: battles could be decided by actually playing out AI fights using each player's fighting style. In addition, players could then fight the opponent AI directly. Such a feature would allow players a glimpse of how they might fare against someone they'll never get to challenge, such as a game designer, a celebrity, a tournament champion, etc. Of course, N-grams cannot fully copy the experience of playing against a real person, but they are capable of mimicking some of the most easily identifiable patterns. The trick is determining which aspects are most identifiable.

Games particularly suitable for this technique are those in which the move selection itself identifies a style. In *Pokémon Ruby Version* and *Pokémon Sapphire Version*, players can set up their own secret base and send it to their friends. Inside a friend's secret base will be an AI-controlled version of that friend, whom the player can battle against. This AI clone already captures many identifying characteristics of the player, such as a unique avatar, salutation, and Pokémon team selection. The AI version will use the same moves as the player it is personifying, but it does so in unexpected patterns, such as charging up an attack and then not unleashing it. This scenario practically begs for the use of N-grams. By simply repeating the move patterns observed from the original player, the AI's imitation would become drastically more accurate of the player's battling style. In well-matched scenarios like this one, N-grams can provide a social experience that is largely untapped in today's games.

48.6 Conclusion

N-grams find patterns in sequences of events. They are good at learning to predict a player's actions at runtime, enabling your AI to compete or cooperate effectively, and balancing difficulty. Further, N-grams can be taught to act according to a style. They can perform procedural generation with consistency. They require little processing to choose their course of action. Lastly, N-grams are easy to implement. Using them in your game usually requires more work than implementing them alone. Properly designating events is of critical importance to an N-gram's pattern recognition, while identifying new types of events in your game will provide new ways for you to leverage N-grams.

References

[Laramée 02] F. D. Laramée. "Using N-gram statistical models to predict player behavior." In *AI Game Programming Wisdom*, edited by Steve Rabin. Hingham, MA: Charles River Media, 2002.

[Millington 06] I. Millington. *Artificial Intelligence for Games*. San Francisco, CA: Morgan Kaufmann, 2006, pp. 580–591.