Toward Example-Driven Program Synthesis of Story Sifting Patterns

Max Kreminski, Noah Wardrip-Fruin, Michael Mateas
University of California, Santa Cruz
{mkremins, nwardrip, mmateas}@ucsc.edu

Abstract

Unlike in *strong story* narrative systems, where the creation of narrative is orchestrated from the top down by a drama manager or similar agent, stories in emergent narrative systems emerge bottom-up from the behavior of autonomous characters in a simulated storyworld. As a result, emergent narrative systems do not always know what stories they're telling, and a perennial challenge for these systems involves recognizing and showcasing emergent stories as they unfold. *Story sifting* technologies can enable a narrative system to automatically recognize emergent stories that match certain *sifting patterns*, but hand-authoring these sifting patterns can be difficult and time-consuming. To support users in authoring these sifting patterns, we present an interactive example-driven program synthesizer that can generate realistic sifting patterns given a few examples of target event sequences and refine the resulting patterns based on further user feedback.

Introduction

In emergent narrative systems, story emerges organically from interaction between characters in a simulated storyworld (Louchart et al. 2015). Past research in player retellings of their play experiences in simulation-driven emergent narrative games has shown that these games often function as storytelling partners for the player (Eladhari 2018; Kreminski and Wardrip-Fruin 2019; Kreminski et al. 2019). To support this style of play, we would like to make these games better at mixed-initiative co-creative (Liapis et al. 2016) storytelling.

One difficulty in doing this is that it's hard to get emergent narrative games to understand the emergent stories they're producing. Unlike in *strong story* narrative systems (Mateas and Stern 2000; Riedl and Bulitko 2013), where the creation of narrative is orchestrated from the top down by a drama manager or similar agent, emergent stories are not necessarily recognized by the system that produces them. However, the more understanding the system has of the stories that are emerging, the more effectively it can surface these stories to the player and support narrative development. In particular, understanding of the story that is emerging from the human interactor's perspective enables the narrative system to present the interactor with *offers* of various ways in which the story might be further developed (Louchart et al. 2008). As a result, it's often desirable to provide emergent narrative systems with more sophisticated ways of understanding the stories they're creating.

Story sifting (Ryan, Mateas, and Wardrip-Fruin 2015; Ryan 2018) is one way to build narrative systems that understand emergent stories. So far, implementing story sifting has involved hand-writing lots and lots of small *story sifting patterns* that recognize certain kinds of interesting emergent microstories. While this approach has its advantages, thinking up and writing these sifting patterns can be both time-consuming and error-prone. The Felt system (Kreminski, Dickinson, and Wardrip-Fruin 2019) attempts to mitigate some of the difficulties of authoring sifting patterns by introducing an approachable domain-specific language intended to improve authorial leverage and open up the authoring of sifting patterns to writers as well as dedicated narrative system programmers, but substantial difficulties remain.

In order to partially address the difficulties of hand-writing sifting patterns, we propose an approach to synthesizing sifting patterns from user-provided examples of target event sequences: an interactive, domain-specific *example-driven program synthesizer* for story sifting patterns represented as Datalog-equivalent queries in the Felt story sifting DSL. Our system leverages inductive logic programming, along with other program synthesis techniques, to generate plausible story sifting patterns from few examples and present these patterns to the user for interactive refinement.

Related Work

Our work draws on existing literature in program synthesis, and specifically the synthesis of Datalog programs, to which Felt sifting patterns are equivalent. Much past work on synthesizing logic programs has been done under the banner of *inductive logic programming* (ILP). ILP systems—including FOIL (Quinlan 1990), GOLEM (Muggleton and Feng 1992), and Progol (Muggleton 1995)—attempt to learn a logic program that correctly characterizes the distinction between a set of positive and a set of negative examples (Muggleton and De Raedt 1994). Most existing ILP systems aspire to generality, limiting the extent to which they can use domain knowledge about what kinds of programs are likely wanted to avoid exhaustively searching a relatively unrestricted space of potential programs. However, the Metagol system (Muggleton, Lin, and Tamaddoni-Nezhad 2015) compromises by allowing the user to provide a set of meta-rules that encompass a sort of domain theory, rendering the problem of predicate invention more computationally tractable by cutting down the search space.

More recently, a number of other techniques from the wider program synthesis literature have been applied to Datalog program synthesis, including constraint-based (Albarghouthi et al. 2017), syntax-guided (Si et al. 2018), and provenance-guided (Raghothaman et al. 2019) approaches. These newer systems have improved on both the performance and the expressivity of classical ILP approaches to logic program synthesis: they can either produce the same programs substantially more quickly than older systems; produce programs that older systems could not have produced at all, for instance by introducing new predicate invention techniques; or both.

To the best of our knowledge, program synthesis has not yet been applied in an interactive or generative narrative research context. However, there has been some limited use of program synthesis in the adjacent procedural content generation and game generation communities. For instance, the ILP system Leda (Summerville 2018) synthesizes AnsProlog inference rules describing the relationship between low-level game mechanics and higher-level aspects of player experience. Meanwhile, Butler, Siu, and Zook (2017) apply non-ILP program synthesis techniques to the generation of boss monster behaviors for digital games.

Background

Our goal is to synthesize Felt sifting patterns. As background, to showcase the features of the Felt sifting DSL, we first present a complete hand-authored Felt sifting pattern, adapted from the `arson-revenge` pattern documented in (Ryan 2018):

```
(eventSequence ?e1 ?e2)
[?e1 eventType hatchRevengeScheme]
[?e2 eventType setFire]
(contributingCause ?e1 ?e2)
[?e1 actor ?arsonist]
[?e2 actor ?arsonist]
(not [?e2 tag accidental])
```

This pattern matches a sequence of two events: a `hatchRevengeScheme` event, followed by a non-accidental `setFire` event with the same protagonist and to which the first event was a contributing cause. Tokens beginning with a question mark, such as `?e1` and `?arsonist`, denote logic variables, whose values are unified with one another across the whole sifting pattern. Square-bracketed clauses (such as `[?e2 eventType setFire]` and `[?e1 actor ?arsonist]`) indicate assertions of the form `[entity attribute value]`, which can be read as stating that the entity on the left has an attribute with the name in the middle whose value is the string, number, or entity ID on the right. Parenthesized clauses (such as `(eventSequence ?e1 ?e2)` and `(contributingCause ?e1 ?e2)`) denote invocations of domain-specific Datalog inference rules, defined as part of the simulation. Clauses can also be negated with the special `not` syntax, as in `(not [?e2 tag accidental])` here.

From a program synthesis perspective, the problem of generating story sifting patterns has several unusual aspects that informed our approach.

- Our past experience authoring sifting patterns has given us a strong domain theory about which relations between events, characters, and other storyworld entities are worthwhile to explore when sifting, enabling us to aggressively prune the search space of candidate programs.
- We don't want our synthesis algorithm to generate relations involving certain properties. For instance, in a sifting pattern, relationships between the numeric IDs of involved entities shouldn't be considered, nor should relationships including characters who aren't directly involved in the events under consideration. Further, because we want to create interpretable sifting patterns that can be presented to an end user directly, to explore these irrelevant relationships would run the risk of overwhelming the user with extraneous information.
- Ideally, for easy interoperability with other tools (including Felt), we want our synthesis algorithm to be fast enough to run interactively in a web browser—a task for which state-of-the-art general-purpose ILP systems, in spite of their significant performance improvements in recent years, are still not well-suited.
- We don't need to tackle the difficult ILP problem of inventing reusable intermediate predicates ourselves, as we can generally rely on the authors of the simulation domain to have provided relevant intermediate predicates for us.

For these reasons, instead of leveraging an existing off-the-shelf ILP system or other synthesizer, we elected to build a small domain-specific ILP system of our own, intended for interactive use as a human-in-the-loop authoring tool.

Approach

Given a database of storyworld events, we first present the user with an interface for selecting *positive examples*: sequences of events that the synthesized sifting pattern should match. Initially, each positive example is represented as a sequence of one or more numeric event IDs. Once the user has selected a set of positive examples, we *enrich* each example with background information by running several queries against the database to fetch and cache information about the events and characters that are directly involved in this example. What background information needs to be fetched depends on how events and characters are represented in the simulated storyworld at hand; for the simple storyworld we used to evaluate this technique, the information fetched by this process includes the type and tags of each event in the sequence, as well as the numeric IDs of the characters that participated in each event as an actor or target.

We then pick an example at random and generate a set of *properties* for this example. As in Odena and Sutton (2020), a property is a small program that takes in an example and returns a boolean value indicating whether this property holds for this example. Intuitively, a property can be thought of as a straightforwardly answerable yes/no question about the example: for instance, "Does the first event in the sequence have event type `betray`?" or "Is the protagonist of the first event the same person as the target of the last event?" Once a set of properties intended to be applied to all of the user-provided examples has been generated, we can test these properties against an example to produce the example's *property signature*: the subset of all generated properties that hold for this example.

Our current approach generates two kinds of properties, based on the most commonly used features in an existing library of sifting patterns developed for the emergent narrative game *Why Are We Like This?* (Kreminski et al. 2020a; 2020b). First, we examine the distinguishing attributes of each event in the example sequence and generate *event attribute properties*. In our test simulation, events are distinguished by their `eventType` and `tag` attributes, so we generate properties of the form `eventType_eN_type` for each event's type and properties of the form `eventTag_eN_tag` for each tag on each event. For instance, if the first event in the example sequence is an `insultDismissively` event with the tags `unfriendly` and `highStatus`, the example's property signature would include the following properties:

- `eventType_e1_insultDismissively`
- `eventTag_e1_unfriendly`
- `eventTag_e1_highStatus`

Next, we generate *entity relationship properties* based on a simulation-specific library of Datalog inference rules relating involved entities to one another. These rules include character/character relationships, e.g. `likes`, `dislikes`, and `ancestor`; character/event relationships, e.g. `eventAdvancesHeldValue`; and event/event relationships, e.g. `indirectCause`. These rules form a significant part of the domain theory about which relationships might be relevant to story sifting in a particular simulation, and our system is able to translate these rules directly into properties, allowing the domain theory to evolve as new rules are defined.

Once we've generated a complete property signature for the first example, we test these properties against the second example and remove any that don't hold. We then iteratively repeat this process for all the other examples, winnowing down the set of properties to just those that hold for all positive examples.

The resulting set of properties may need to be pruned to eliminate redundancies. First, we prune properties that logically duplicate other properties, for instance by asserting a relationship between two logic variables representing characters for which the same relationship has already been asserted. (If the protagonist of the first event and the target of the second event are known to be the same character, and we've already asserted that the target of the first event is friends with the protagonist of the first event, we don't need to reassert the same relationship between the target of the first event and the target of the second.) Then we prune properties that are fully logically subsumed by other properties. For instance, in our test simulation, an event's tags are always dependent on its type; therefore, we eliminate all properties of the form `eventTag_eN_TAG` if a property of the form `eventType_eN_TYPE` is also present.

At this point, we can synthesize a story sifting pattern (i.e. a Datalog-equivalent query in the Felt story sifting DSL) and run it against the storyworld state database to get any *other* matches that might exist, besides the user-selected positive examples, and show these to the user.

If the user thinks the pattern shouldn't match one or more of these examples, then they can add these examples as *negative examples*. We generate properties for each negative example in the same way as before, and identify which properties are consistently true for all negative examples and consistently untrue for all positive examples. Then we refine the synthesized sifting pattern to exclude these negative properties and re-present the matches to the user.

Usage Examples

Story sifting can be applied to event sequences from a variety of sources. At one end of the spectrum, sifting techniques are sometimes used to extract narrative from sequences of game events that were initially generated without regard for narrativity, as in many sports games (Rhodes, Coupland, and Cruickshank 2010). At the other end of the spectrum, sifting can also be applied to the output of sophisticated narrative simulations like Talk of the Town (Ryan et al. 2015) and *Dwarf Fortress* (Bay 12 Games 2006; Garbe 2018), which employ high-fidelity models of character motivation and other narrative-relevant concerns at the level of event generation, before sifting is applied. Because these event sources vary widely in the baseline level of narrative structure and sophistication they provide, we want our approach to testing to assume minimal narrative structure in the storyworld database. This allows us both to stress-test the generality of our synthesis approach (ensuring that it doesn't implicitly depend on regularities in a particular simulation) and to place the primary authoring affordance on example specification (rather than spreading it between example specification and simulation authoring).

Therefore, in order to test our system, we first generated a simple storyworld containing five characters and 200 random events. To establish a rudimentary social graph, a handful of directed `likes` and `dislikes` relationships and a handful of symmetric `coworkers` relationships were added between some pairs of characters at random. Additionally, from a pool of eight *values* (such as `authority`, `communalism`, and `comfort`), each character was randomly assigned two values that they support and one value that they oppose.

For each event, we randomly selected one of 35 possible event types, then randomly assigned two different characters to be the actor and target of this event. Events were also assigned a handful of tags based on their event type; the storyworld as a whole contained 12 total distinct event tags, with each individual event having between one and four tags. Events of certain types were also marked as supporting or opposing some of the values held by characters: for instance, a `rejectSuperiority` event might be marked as opposing the `authority` value. The resulting storyworld was then used as background for the following examples.

Simple Positive Examples

We attempted to synthesize sifting patterns for several recurring, emergent patterns of events that we found narratively compelling. First, we attempted to synthesize a sifting pattern for the microstory "romantic failure followed by romantic success", which we chose to operationalize as a sequence of two romantic failures for the same character, followed by a romantic success for that same character. We provided the system with one positive example of this pattern:

1. Sarah tried to flirt with Mira, but was rejected.
2. Sarah tried to ask Mira out on a date, but was rejected.
3. Sarah tried to ask Emin out on a date, and succeeded.

Given only this example, our system generates 35 properties. 3 of these (one per event in the example) involve event types; 10 involve event tags; 4 involve statements about the same character being involved in two different roles, for instance as the target of both the first and the second events. The remaining 18 properties involve various character/character relationships, for instance indicating that Mira dislikes Sarah; Mira likes Emin; and that Emin and Sarah are coworkers.

We then provided another positive example for the same pattern:

1. Zach tried to ask Emin out on a date, but was rejected.
2. Zach tried to rekindle a romantic relationship with Sarah, but was rejected.
3. Zach tried to flirt with Mira, and succeeded.

This second example was sufficient to reduce the set of common properties between both examples down to just nine:

- `eventTag_e1_negative`
- `eventTag_e1_romantic`
- `eventTag_e2_negative`
- `eventTag_e2_romantic`
- `eventTag_e3_positive`
- `eventTag_e3_romantic`
- `sameCharacter_e1actor_e2actor`
- `sameCharacter_e1actor_e3actor`
- `sameCharacter_e2actor_e3actor`

By prepending a set of static *setup clauses* and pruning setup clauses that introduced unused logic variables, these properties were then translated into the following complete Felt sifting pattern:

```
(eventSequence ?e1 ?e2 ?e3)
[?e1 actor ?e1actor]
[?e2 actor ?e2actor]
[?e3 actor ?e3actor]
[?e1 tag negative] [?e1 tag romantic]
[?e2 tag negative] [?e2 tag romantic]
[?e3 tag positive] [?e3 tag romantic]
[(= ?e1actor ?e2actor ?e3actor)]
```

The `eventSequence` setup clause here ensures that all of its arguments are database entities of type `event`, and that all of these events occurred in chronological order from left to right (potentially interspersed with arbitrarily many other events). Meanwhile, setup clauses of the form `[?eN actor ?eNactor]` are used to bind characters involved in these events to temporary logic variables, so that they can be referenced in later clauses.

Besides the two provided positive examples, the resultant sifting pattern also matched 41 other event sequences in the database, all of which fulfilled the intended requirements. Many of these matches included some, but not all, of the events provided in the positive examples.

Adding Negative Examples

Building on the same "romantic failure followed by romantic success" microstory, we next attempted to refine the synthesized sifting pattern to specifically require that the first event in the sequence is not a *major* romantic failure (such as a breakup or failed proposal). Some events in our test simulation had the `major` tag, but there was no corresponding `minor` tag for non-major events, so this had to be accomplished through the use of negative properties.

From the set of matches for the previous sifting pattern, we added the following match as a *negative example*:

1. Mira tried to propose to Emin, but was rejected.
2. Mira tried to ask Zach out on a date, but was rejected.
3. Mira tried to flirt with Zach, and succeeded.

The system initially proposed several candidate negative properties, all of which were true for the single negative example but false for both of the positive examples:

- `eventType_e1_propose_rejected`
- `sameCharacter_e2target_e3target`
- `likes_e1actor_e1target`

However, the first of these three properties was too specific (targeting the event type, rather than the tags, of the first event in the sequence), while the latter two were incidentally applicable, but unrelated to our intent. To further refine the pattern, we then provided a second negative example:

1. Emin broke up with Vincent.
2. Emin tried to flirt with Vincent, but was rejected.
3. Emin tried to ask Sarah out on a date, and succeeded.

This enabled the system to further filter down the set of candidate negative properties, resulting in the addition of the following (correct) clause to the sifting pattern:

```
(not [?e1 tag major])
```

Note that this negative clause was initially subsumed by the more specific `eventType` constraint when only a single negative example was provided. A second negative example was needed to show the system that it was the event *tags* (rather than the event *type*) of the first event in the sequence that we actually intended to restrict.

Incorporating Entity Relationships

Next, we attempted to synthesize a sifting pattern for a microstory in which a character's ideological rival calls them out on a hypocritical action. An example of this pattern might look something like the following sequence of events:

1. A performs an action opposed to one of their own values.
2. B criticizes A.

...where B is a character who holds a value to which A is opposed, or vice versa. Although this sequence of events is short, matching instances of this microstory requires the synthesis of a pattern that includes both a character/character relationship (i.e. that characters A and B hold opposed values) and a character/event relationship (i.e. that the first action in the sequence harms a value held by A, or advances a value to which A is opposed).

We first provided the system with two positive examples. This was sufficient to produce the following nearly-correct sifting pattern:

```
(eventSequence ?e1 ?e2)
[?e1 actor ?e1actor]
[?e2 actor ?e2actor]
[?e2 target ?e2target]
[?e2 eventType criticize]
[(= ?e1actor ?e2target)]
(likes ?e2actor ?e1actor)
(opposedValues ?e1actor ?e2actor)
(eventHarmsHeldValue ?e1 ?e1actor)
```

We then had to provide one more positive example to eliminate the spurious `(likes ?e2actor ?e1actor)` clause, introduced by a property that incidentally happened to hold for the first two examples we provided but that was not part of our intent. Alternatively, an attentive user might notice that this clause was not part of their intent and manually eliminate it from the sifting pattern without giving the system any further examples.

Currently Unsupported Negative Constraints

Besides the constraint types illustrated here, there exists one other type of constraint that's fairly common in existing Felt sifting patterns, but that our system currently doesn't make any attempt to synthesize. These are *compound negative event constraints*, which (among other uses) allow sifting patterns to avoid matching candidate event sequences that are interrupted by events with certain attributes. For instance, suppose you want to write a sifting pattern to match a "violation of hospitality" microstory, in which a traveling character enters a town, is shown hospitality by a resident of this town, and then experiences harm at the hands of this same town resident character. A naïve implementation of this sifting pattern might look like this:

```
(eventSequence ?e1 ?e2 ?e3)
[?e1 eventType enterTown]
[?e1 actor ?guest]
[?e2 eventType showHospitality]
[?e2 actor ?host] [?e2 target ?guest]
[?e3 tag harm]
[?e3 actor ?host] [?e3 target ?guest]
```

However, this sifting pattern would also match event sequences like the following, in which the intended interpretation of the matched events is invalidated by the traveler leaving the town before the final event of the intended sequence plays out:

1. Yann enters town.
2. Yann is shown hospitality by Ema.

∗ Yann leaves town.

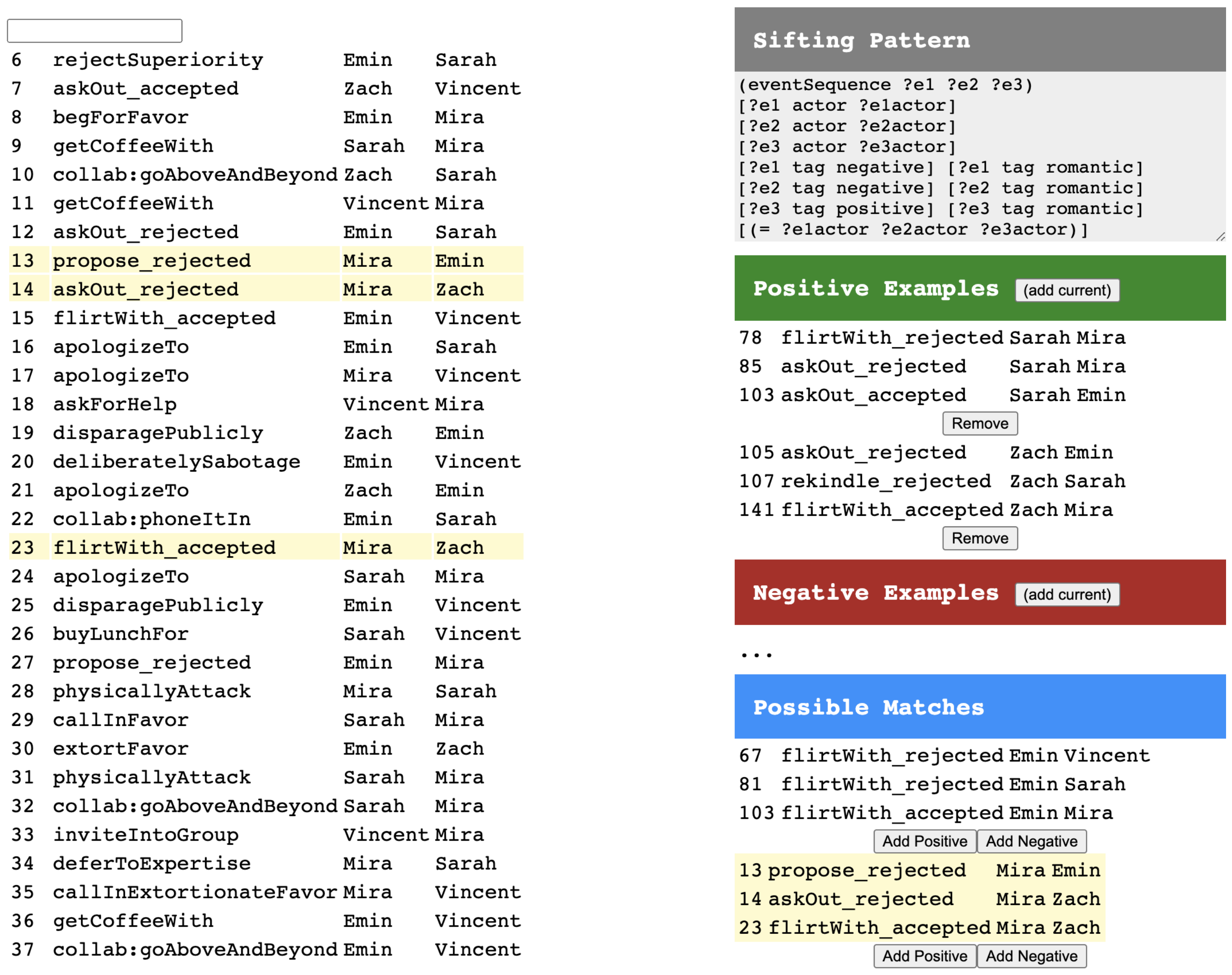
3. Ema pickpockets Yann, getting away with all their money.

To address this problem, an additional clause might be appended to the end of the sifting pattern, taking advantage of the `not-join` syntax (inherited from the DataScript library atop which the Felt sifting pattern DSL is built) to specify that the traveler must not leave town between the first and last events of the sequence. This clause essentially states that there must not exist any event with the specified characteristics (`?eMid`) between `?e1` and `?e3`:

```
(not-join [?e1 ?e3 ?guest]
  (eventSequence ?e1 ?eMid ?e3)
  [?eMid eventType leaveTown]
  [?eMid actor ?guest])
```

However, the search space of possible compound negative event clauses for any given set of user-provided examples is very large. The first difficulty lies in identifying which event or events that aren't part of the provided negative example are somehow disqualifying the negative examples from consideration. We could narrow down the search space somewhat by considering only events that occurred between the start and the end of the provided negative example, and that involved at least one of the characters involved in the example events directly, but this could still include prohibitively many events. And the second difficulty lies in determining what aspects of the disqualifying event are responsible for the disqualification: is it the event type, the tags, the actor, the target, one of the relationships between these entities to one another (or to other entities in the example), or some combination of these factors?

Perhaps the most promising way to cut down this search space is to ask the user for further guidance as to which interceding events are responsible for disqualifying each negative example. This guidance could then allow the system to suggest reasonable combinations of negative properties involving these events. However, presenting the user with an interface that allows them to easily identify the relevant events may be difficult due to the sheer volume of events generated by many emergent narrative systems. For the time being, we leave this problem to future work.



```
6  rejectSuperiority         Emin    Sarah
7  askOut_accepted           Zach    Vincent
8  begForFavor               Emin    Mira
9  getCoffeeWith             Sarah   Mira
10 collab:goAboveAndBeyond   Zach    Sarah
11 getCoffeeWith             Vincent Mira
12 askOut_rejected           Emin    Sarah
13 propose_rejected          Mira    Emin
14 askOut_rejected           Mira    Zach
15 flirtWith_accepted        Emin    Vincent
16 apologizeTo               Emin    Sarah
17 apologizeTo               Mira    Vincent
18 askForHelp                Vincent Mira
19 disparagePublicly         Zach    Emin
20 deliberatelySabotage      Emin    Vincent
21 apologizeTo               Zach    Emin
22 collab:phoneItIn          Emin    Sarah
23 flirtWith_accepted        Mira    Zach
24 apologizeTo               Sarah   Mira
25 disparagePublicly         Emin    Vincent
26 buyLunchFor               Sarah   Vincent
27 propose_rejected          Emin    Mira
28 physicallyAttack          Mira    Sarah
29 callInFavor               Sarah   Mira
30 extortFavor               Emin    Zach
31 physicallyAttack          Sarah   Mira
32 collab:goAboveAndBeyond   Sarah   Mira
33 inviteIntoGroup           Vincent Mira
34 deferToExpertise          Mira    Sarah
35 callInExtortionateFavor   Mira    Vincent
36 getCoffeeWith             Emin    Vincent
37 collab:goAboveAndBeyond   Emin    Vincent
```

Sifting Pattern

```
(eventSequence ?e1 ?e2 ?e3)
[?e1 actor ?e1actor]
[?e2 actor ?e2actor]
[?e3 actor ?e3actor]
[?e1 tag negative] [?e1 tag romantic]
[?e2 tag negative] [?e2 tag romantic]
[?e3 tag positive] [?e3 tag romantic]
[(= ?e1actor ?e2actor ?e3actor)]
```

Positive Examples (add current)

```
78  flirtWith_rejected Sarah Mira
85  askOut_rejected    Sarah Mira
103 askOut_accepted    Sarah Emin
```
Remove
```
105 askOut_rejected    Zach Emin
107 rekindle_rejected  Zach Sarah
141 flirtWith_accepted Zach Mira
```
Remove

Negative Examples (add current)

...

Possible Matches

```
67  flirtWith_rejected Emin Vincent
81  flirtWith_rejected Emin Sarah
103 flirtWith_accepted Emin Mira
```
Add Positive Add Negative
```
13 propose_rejected   Mira Emin
14 askOut_rejected    Mira Zach
23 flirtWith_accepted Mira Zach
```
Add Positive Add Negative

Figure 1: A screenshot of the system's current user interface. On the left sits a scrolling, filterable log of all events that have occurred in the storyworld so far, allowing the user to select event sequences to use as examples. On the right sits an editable view of the current synthesized sifting pattern; the sets of positive and negative examples the user has provided; and the set of additional matches for the current candidate sifting pattern, which the user can add as positive or negative examples.

Discussion

Our approach avoids many of the difficulties associated with general program synthesis by encoding a lot of information about the domain, especially in the form of hand-authored Datalog rules about relationships between characters and events. We don't need to try to discover these rules generically if we've been provided with a good set of special-case rules up front, so we don't have to do the actually-hard part of Datalog program synthesis (i.e. trying to invent these potentially-recursive rules ourselves given *just* the concrete examples.)

We avoid some of the *other* difficulties by including a human in the loop and building our synthesized programs out of relatively straightforward, individually comprehensible building blocks. Individual Datalog clauses can be presented to the human user as short, declarative natural language sentences, so even a relatively naïve user can figure out which clauses of the synthesized sifting pattern make sense and which ones don't. In this sense, program synthesis serves as an on-ramp that teaches users about the DSL used to define sifting patterns and the set of features that this DSL supports, so that users can move from exclusively defining sifting patterns via examples toward modifying existing sifting patterns and maybe even constructing new sifting patterns from scratch via the textual DSL.

We consider our approach to be an example of *human-centric program synthesis* (Crichton 2019), which deals with "what applications [of program synthesis] open up when a user has the programming skills to express specifications at a level beyond examples". For us, synthesis functions partly as a teaching tool for introducing users to the textual language in which sifting patterns are expressed and making them aware of what affordances this language provides. Our synthesized programs aren't treated as black boxes, but exposed to the user for editing, and it's our hope that users will combine the programming-by-example features that this tool affords with direct manipulation of generated sifting patterns to modify, add, and remove constraints.

Moreover, our tool remains useful even for experienced sifting pattern authors, who might not immediately notice all of the properties they intend the sifting pattern they're writing to contain. In addition, since the set of relationships that might exist between characters is likely to evolve over the course of developing a complex simulation-driven emergent narrative game, even experienced sifting pattern authors might benefit from the existence of a system that can surface new clause types to them as new Datalog rules are implemented by other developers.

Limitations

As discussed previously, our current system makes no attempt to synthesize compound negative event constraints. To support synthesis of these clauses will likely require further work on both the core synthesis algorithm and the user interface to provide the user with a way to explain why negative examples are negative by pointing out the specific interceding events that invalidate them.

Currently, our approach enforces an implicit total ordering constraint on the events matched by the synthesized sifting pattern. All examples provided for a single sifting pattern (positive and negative) must be of the same length, and it's assumed that the Nth event of each example event sequence is intended to play the same role in the target microstory as the Nth event of every other example. This strict constraint may not always be intended or desirable; for instance, it's sometimes preferable to construct a sifting pattern in which the middle events (between the first and last events of the matched sequence) are permitted to occur in any chronological order. However, assuming an implicit total ordering constraint allows us to simplify our approach to generating both properties and setup clauses. In the future, we may seek to relax this constraint, but to do so would require significant modification to our current approach.

Like many learning approaches, our approach may overfit when few examples are available. In preliminary testing with an even smaller simulated storyworld, the system was prone to including properties in synthesized patterns that were incidentally true of all examples but not part of our intent. This can be mitigated by users reviewing the synthesized sifting patterns and removing unintended clauses. Additionally, we may be able to help users debug overfitted patterns by procedurally relaxing each clause of the pattern and presenting the user with "almost matches", inspired by Writing Buddy's "almost actions" (Samuel, Mateas, and Wardrip-Fruin 2016). This could help users diagnose which clause of a synthesized pattern is responsible for overfit.

Conclusion

We present a domain-specific inductive logic programming system capable of synthesizing story sifting patterns in the Felt sifting DSL, given several user-provided examples of narratively interesting event sequences. Our system is intended for interactive use with a human user in the loop. We avoid or mitigate some of the difficulties of general-purpose program synthesis (especially the problem of intractably large search spaces) by encoding a lot of domain knowledge into the architecture of the synthesis algorithm. Our system features several key limitations, especially around the synthesis of complex negative constraints, but is nevertheless able to synthesize useful, realistic sifting patterns from a small number of example event sequences.

Potential future work includes extension of the user interface and synthesis algorithm to better support complex negative constraints; incorporation of the synthesizer into a more full-featured and user-friendly sifting pattern authoring tool; evaluation of the resulting tool with a larger number of potential users; and a more detailed comparison of our hand-rolled domain-specific synthesizer against existing general-purpose Datalog program synthesizers, to determine whether we could feasibly re-encode the problem in a way that makes it more amenable to attack by a general-purpose synthesizer.

References

Albarghouthi, A.; Koutris, P.; Naik, M.; and Smith, C. 2017. Constraint-based synthesis of datalog programs. In *International Conference on Principles and Practice of Constraint Programming*, 689–706. Springer.

Bay 12 Games. 2006. Dwarf Fortress. bay12games.com/dwarves.

Butler, E.; Siu, K.; and Zook, A. 2017. Program synthesis as a generative method. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*.

Crichton, W. 2019. Human-centric program synthesis. In *PLATEAU Workshop @ UIST*.

Eladhari, M. P. 2018. Re-tellings: the fourth layer of narrative as an instrument for critique. In *International Conference on Interactive Digital Storytelling*, 65–78. Springer.

Garbe, J. 2018. Simulation of history and recursive narrative scaffolding. project.jacobgarbe.com/simulation-of-history-and-recursive-narrative-scaffolding.

Kreminski, M., and Wardrip-Fruin, N. 2019. Generative games as storytelling partners. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*.

Kreminski, M.; Samuel, B.; Melcer, E.; and Wardrip-Fruin, N. 2019. Evaluating AI-based games through retellings. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 15, 45–51.

Kreminski, M.; Dickinson, M.; Mateas, M.; and Wardrip-Fruin, N. 2020a. Why Are We Like This?: Exploring writing mechanics for an AI-augmented storytelling game. In *Proceedings of the Electronic Literature Organization Conference*.

Kreminski, M.; Dickinson, M.; Mateas, M.; and Wardrip-Fruin, N. 2020b. Why Are We Like This?: The AI architecture of a co-creative storytelling game. In *Proceedings of the Fifteenth International Conference on the Foundations of Digital Games*.

Kreminski, M.; Dickinson, M.; and Wardrip-Fruin, N. 2019. Felt: a simple story sifter. In *International Conference on Interactive Digital Storytelling*, 267–281. Springer.

Liapis, A.; Yannakakis, G. N.; Alexopoulos, C.; and Lopes, P. 2016. Can computers foster human users' creativity? Theory and praxis of mixed-initiative co-creativity. *Digital Culture & Education* 8(2):136–153.

Louchart, S.; Swartjes, I.; Kriegel, M.; and Aylett, R. 2008. Purposeful authoring for emergent narrative. In *Joint International Conference on Interactive Digital Storytelling*, 273–284. Springer.

Louchart, S.; Truesdale, J.; Suttie, N.; and Aylett, R. 2015. Emergent narrative, past, present and future of an interactive storytelling approach. In *Interactive Digital Narrative: History, Theory and Practice*. Routledge. 185–199.

Mateas, M., and Stern, A. 2000. Towards integrating plot and character for interactive drama. In *Working notes of the Social Intelligent Agents: The Human in the Loop Symposium*, 113–118. Menlo Park: AAAI Fall Symposium Series.

Muggleton, S., and De Raedt, L. 1994. Inductive logic programming: Theory and methods. *The Journal of Logic Programming* 19:629–679.

Muggleton, S., and Feng, C. 1992. Efficient induction of logic programs. *Inductive logic programming* 38:281–298.

Muggleton, S. H.; Lin, D.; and Tamaddoni-Nezhad, A. 2015. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning* 100(1):49–73.

Muggleton, S. 1995. Inverse entailment and progol. *New generation computing* 13(3-4):245–286.

Odena, A., and Sutton, C. 2020. Learning to represent programs with property signatures. In *International Conference on Learning Representations (ICLR)*.

Quinlan, J. R. 1990. Learning logical definitions from relations. *Machine learning* 5(3):239–266.

Raghothaman, M.; Mendelson, J.; Zhao, D.; Naik, M.; and Scholz, B. 2019. Provenance-guided synthesis of datalog programs. *Proceedings of the ACM on Programming Languages* 4(POPL):1–27.

Rhodes, M.; Coupland, S.; and Cruickshank, T. 2010. Enhancing real-time sports commentary generation with dramatic narrative devices. In *Joint International Conference on Interactive Digital Storytelling*, 111–116. Springer.

Riedl, M. O., and Bulitko, V. 2013. Interactive narrative: An intelligent systems approach. *AI Magazine* 34(1).

Ryan, J. O.; Summerville, A.; Mateas, M.; and Wardrip-Fruin, N. 2015. Toward characters who observe, tell, misremember, and lie. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Ryan, J. O.; Mateas, M.; and Wardrip-Fruin, N. 2015. Open design challenges for interactive emergent narrative. In *International Conference on Interactive Digital Storytelling*, 14–26. Springer.

Ryan, J. 2018. *Curating Simulated Storyworlds*. Ph.D. Dissertation, UC Santa Cruz.

Samuel, B.; Mateas, M.; and Wardrip-Fruin, N. 2016. The design of Writing Buddy: a mixed-initiative approach towards computational story collaboration. In *International Conference on Interactive Digital Storytelling*, 388–396. Springer.

Si, X.; Lee, W.; Zhang, R.; Albarghouthi, A.; Koutris, P.; and Naik, M. 2018. Syntax-guided synthesis of Datalog programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 515–527.

Summerville, A. 2018. Towards inductive logic programming for game analysis: Leda. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*.