Fast, Declarative, Character Simulation Using Bottom-Up Logic Programming

Ian Horswill, Samuel Hill[1]

[1] *Northwestern University, 2233 Tech Drive, Evanston, IL, 60208, USA*

Abstract

Declarative programming offers several advantages in terms of compactness and modularity. Logic programming and rule-based systems are often chosen for tasks such as social simulation because their use of declarative rules and predicates map well to rules of social engagement. Unfortunately, declarative programming is often quite slow, making it inappropriate for large systems or high-frequency updates. This is partly because of its use of search algorithms, but also because of its heavy use of pointer chasing, dynamic allocation, garbage collection, and runtime type-checking.

In this paper, we discuss how bottom-up execution of logic programs can be implemented without these issues. We argue that character simulation is a "sweet spot" for bottom-up logic programming, allowing character behavior to be specified in terms of declarative rules, while offering performance competitive with Python systems such as Talk of the Town. We present a language, TED, which offers very good performance and has been used both in research and in an unannounced commercial game.

Keywords

Social simulation, logic programming, declarative programming

1. Introduction

Simulations, including games, involve iterating through data structures representing the world state, updating the various components of the world and looking for pairwise interactions between them.

For example, needs-based AI [1], such as in *The Sims* [2] involves finding for each character c an object o in the world that best satisfies its various needs n using some variant of the one-line formula:

$$B(c) = \arg\max_{o \in O} \sum_{n \in N} S(c, o, n)$$

The naïve implementation of this involves three nested loops running over the sets of characters C, objects O, and needs N. :

```
For each character c in C
   If character idle
      Best score = 0
      For each object o in O
         Score = 0
         For each need n in N
            Score += S(c, o, n)
         If score > best score
            Best object = o
            Best score = score
      Interact with best object
```

This has running time $O(CON)$. In an effort to optimize it, the programmer might maintain separate, dynamic lists of just the characters that need to be updated, just the objects that are available to satisfy a specific need, etc. This comes at the cost of having to modify other parts of the program to maintain these lists, increasing development costs and dependencies between modules.

Ideally, one would be able to specify the fundamental computation being performed (maximization) separately from the choice of data structures, as one does in relational databases: queries are expressed in terms of a set of logical data structures (relational tables). The physical layout of the data can be adjusted independently to best support the mix of queries needed. As those queries inevitably change, the physical data structures can be changed to support them. Similar arguments have been made in the context of entity-component systems for massively online games [3].

In this paper, we describe a high-performance declarative programming language, TED that can compactly express the algorithm above in two lines of code (see figure 1). Moreover, it can be optimized declaratively as in a relational database, by providing annotations about how to index the data. TED is highly performant, running with minimal dynamic allocation, type checking, or pointer chasing. It also supports parallel execution. We also briefly describe a city simulator built using TED.

AIIDE Workshop on Experimental Artificial Intelligence in Games, October 08, 2023, University of Utah, Utah, USA

ian@northwestern.edu (I. Horswill);
samuelhill2022@northwestern.edu (S. Hill)

```
// Assume C[x], O[x], N[x] mean x is a character, object, or need, respectively

// Score[c, o, t] means t is the total score for object o and character c
var Score = Definition("Score", c, o, t).If(O[o], t==Sum(s, And[N[n], s==S[c,o,n]));

// B[c,o] means c is a character whose best object is o.
var B = Predicate("B", c, o).If(C[c], Maximal(o, t, Score[c, o, t]));
```

Figure 1: Needs-based action selection in TED.

2. Logic programming

Logic programming is a family of declarative programming techniques that involve describing a program in terms of a set of predicates (relations) and rules for computing them.

A rule gives a set of conditions implying the truth of a predicate. For example, siblinghood could be defined in terms of shared parentage:

```
Sibling[x,y].If(Parent[x,p], Parent[y,p]);
```

This states that for any *x*, *y*, and *p*, the sibling relationship holds if both parent relationships hold.

Although not always expressed in the form of logic programming, symbolic rules have frequently been used to describe character behavior and social physics in systems such as Inform 7 [4], Comme Il Faut [5], Versu [6], MKULTRA [7], and City of Gangsters [8].

1.1. Top-down execution

Classical logic-programming languages such as Prolog [6][7] execute queries "top-down" using SLD resolution [11]. The user "calls" a predicate such as `Sibling`, with argument values, and the system tries to prove the predicate true of those arguments using one of the predicate's rules.

Using a rule involves matching the rule's variables to the arguments specified in the call. For example, the call `Sibling["Bill", s]`, i.e. "who is a sibling *s* of Bill?", matches the rule above to yield the substitution:

```
Sibling["Bill",s]
       .If(Parent["Bill",p], Parent[s,p]);
```

The system then recursively executes the call `Parent["Bill",p]`. If there was another rule that stated Jenny was one of Bill's parents, then executing this query would set the variable *p* to "Jenny", meaning that the second call is really `Parent[s,"Jenny"]`. If there is a rule that says Jenny is a parent of Christine, then this call would set *s* to Christine. The rule proves `Sibling["Bill", s]` is true and provides Christine as the value of *s*. Further solutions (further siblings) can be found by backtracking the proof process.

Top-down execution computes one solution at a time, which is an advantage when only one is needed. On the other hand, if the same call is performed repeatedly, e.g. by different subgoals of a call, then the entire work of that call is repeated. Moreover, the matching process (unification) is somewhat expensive. For example, just looking up the value of a matched variable requires a loop chasing forwarding pointers.

1.2. Bottom-up execution

Suppose we had already computed the full extension (all the child/parent pairs) of the `Parent` relation and stored them in an array. In that case, the `Sibling` rule above could be computed with the following loop:

```
for each (x,p) in Parent
   for each (y, p2) in Parent
      if p == p2
         return (x,y)
```

Indeed, we could compute an array of all the Siblings, with a small modification:

```
for each (x,p) in Parent
   for each (y, p2) in Parent
      if p == p2
         Sibling.Add( (x,y) )
```

After execution of this loop, `Sibling` contains all the sibling pairs. This has the obvious disadvantages that:

- There may be quite a lot of sibling pairs
- You may only care about the siblings of Bill, in which case the effort to compute the other families is wasted.[2]

However, it also has several advantages:

- Unification can be replaced with if's and assignments to C-like variables.
- Subsequent calls to `Sibling` can simply check the array; rules are executed only once.
- Indexing can be used to speed access to the array.
- What to index can be decided after the rules are written and evolve during the life cycle of the game.

This suggests an alternative execution strategy: compute the complete extensions of each predicate, and place them in arrays, ensuring before executing a rule, we first make sure the predicates it references have been computed. This is known as bottom-up execution, and is the strategy used in Datalog [13]. We

[2] For database queries, it is possible to transform a query into an equivalent query that, when executed bottom-up is as efficient as top-down execution using the so-called magic sets algorithm [12].

However, it's unclear what this would look like in a game engine use-case.

```
// Base table: holds the state of the grid
var Grid = Predicate("Grid", loc.Key, occupied.Indexed);
// Derived table: the number of cells neighboring a given location
var NeighborCells = Predicate("Neighbors", loc.Key, count.Indexed)
  .If(Grid[loc, __], count==Count(And[Neighbor[loc, neighbor], Grid[neighbor, true]]));
// Update table: cell dies if over/underpopulated neighborhood
Grid.Set(loc,occupied,false).If(Grid[loc,true],Neighbors[loc,count],(count<2|count>3));
// Update table: cell born if empty and 3 neighbors
Grid.Set(loc, occupied, true).If(Neighbors[loc, 3]);
```

Figure 2: Conway's Game of Life in TED.

believe character simulation is a "sweet spot" for bottom-up logic programming: it is often defined in terms of rules, and the engine generally does have to compute complete extensions anyway.

3. Declarative simulation

There are many reasons why it's appealing to be able to treat a simulation as a database that one can query using a query language. One reason is it makes the state of the simulation easily inspectable, and so hopefully debuggable, because everything is already stored in tables that can be queried by the user.

The motivating example for this work is Ryan's [14] argument for generating stories by running a city-scale character simulation and then searching ("sifting") its output to find interesting stories. Story sifting effectively requires a query language that can be run against the simulation. Indeed, Datalog has been used for story sifting in the past [15]. However, it involved logging everything in the simulation to a file and then reading the file into a separate application for sifting.

This paper began from the question: *what if the query language could also be the simulation language?* That is, can we write the character simulation logic for games like *The Sims* [2], *Bad News* [16], [17], or *Prom Week* [5], [18] declaratively in some language akin to Datalog? And if so, how performant would it be?

The basic structure of such a simulation is as follows. In the foregoing we will generally use the terms predicate and table interchangeably, since most predicates are represented at runtime as tables. Simulations use three main types of tables:

- **Base tables** store the state of the simulation. They retain their data from one simulation step to the next, except insofar as they're modified by update tables, below.
- **Derived tables** are defined in terms of other tables (base or derived) using rules. Derived tables are recomputed on every step of the simulation.
- **Update tables** list modifications to be made to different base tables at the end of the current simulation step. They are themselves a kind of derived table defined by rules.

As with relational database systems, tables can optionally be indexed by different columns (predicate arguments), making it faster to perform lookups.

4. The TED language

TED is a high-performance, bottom-up logic programming language intended for character simulation in AI-heavy games. It is strongly-typed, supports higher-order predicates, metaprogramming, and parallel execution. It includes an optional run-time parser-evaluator that allows users to make live queries against an executing simulator.

TED is embedded in C#, meaning that TED code is C# code that builds the TED program in memory. This has several advantages:

- Good interoperability between TED and C#. TED code can easily call into C#, and C# code can easily access the contents of TED tables.
- IDE support for C#, such as type and syntax checking, colorization, and refactoring automatically extends to TED.
- C# can be used as a macro language for metaprogramming; higher-level abstractions can be written as C# code that builds the necessary TED tables and rules.

TED by itself is strictly less expressive than Prolog: it does not allow recursion (see section 5.3) and it does not allow Prolog "functors" (composite objects are opaque to TED's pattern matcher). Unlike datalog, TED can call arbitrary C# code, which is Turing-complete. But it still would not be a natural choice to use to implement algorithms such as symbolic integration or natural language generation, that require manipulating tree structures representing symbolic expressions. In exchange for this limitation, we get high performance and parallelizability.

4.1. Trivial example

Figure 2 shows Conway's Game of Life [19] implemented as a short TED program. The program consists of a series of C# statements that build the parse tree of the TED program to be executed.

The first statement creates a new predicate object (table) and stores it in the C# variable `Grid`. It holds the state of the board. The predicate has two arguments, location and whether the location is occupied by a cell, and its table representation has two corresponding columns. It's a base table; it retains its state from tick to tick except as specified by the `Set()` methods at the end. We will talk about `.Set()` and the the `.Key` and `.Indexed` annotations shortly.

The second statement also defines a table and stores it in a C# variable, `NeighborCells`. This table maps locations in the grid to the number of cells surrounding them. Unlike `Grid`, this statement calls the predicate's `.If()` method to add a rule to the predicate: `NeighborCells[loc,count]` is true if:

- `Grid[loc, __]`, i.e. `loc` is a location on the board (`__`ignores that argument), and
- `count==Count(And[…])`, i.e. count is the number of solutions to the `And[…]` query, which finds neighbors of `loc` that have cells.

Since `NeighborCells` has a rule, it's a derived table; its table is erased and recomputed on each tick.

The last two statements call the `.Set()` method of the `Grid` predicate. The `Set()` method creates and returns a new predicate, which is a table of rows to be updated. The two calls create two such tables. The `.If()` method called at the ends of the statements adds rules to those tables, causing it to recompute those tables, and hence the grid locations to modify, on each tick, based on their respective rules. The first rule says to set the `occupied` column of any location with less than 2 or more than 3 neighbors to false; it "kills" the cell. The second rule says to set the `occupied` column for locations with exactly 3 neighbors to true; it spawns cells.

4.1.1. Naïve execution

The control flow of the overall program is as follows:

```
repeat forever
   recompute NeighborCells based on Grid
   recompute table of cells to destroy
   recompute table of cells to create
   update Grid based on update tables
```

Let's look in detail at the rule in the third statement, which says to kill a cell at location `loc` if:

- `Grid[loc,true]`
  There's a cell at `loc`,
- `Neighbors[loc,count]`
  It's neighbor count is `count`,
- `(count<2|count>3)`
  And the count is outside the desired range

Remember this is making a table of cells to kill that will be rebuilt each tick. The naïve execution algorithm for this would be:

```
clear the table
for each (loc1, occupied) in Grid
   if occupied == true
      for each (loc2, count)
               in NeighborCells
         if loc1==loc2
            if (count<2||count>3)
              add loc1 to the table
```

On the positive side, the `.If()` rule is considerably more compact than the equivalent code above, which is generally a good thing. However, it exhaustively searches the `NeighborCells` table each time it tries to look up the count for a specific location; for g grid cells, the algorithm is $O(g^2)$. So on the face of it, the logic program is a terrible idea.

4.1.2. Declarative optimization

We can speed this rule up by indexing the tables so they don't need to be scanned. The `.Key` and `.Indexed` annotations in the declarations of `Grid` an `NeighborCells` tell the system to index the tables by the specified columns. In the `.Key` case, the annotation also promises that the values in that column will be unique; no two rows can have the same key. Key indices map column values to single rows. Non-key indices map values to sets of rows. Both rows list location as a key; given a location, we can find its row in $O(1)$ time. They also index their second column; we can get the sets of rows with/without cells or rows with a given number of neighbors, in $O(1)$ time. Using indexing, the rule above effectively executes as:

```
clear the table
foreach (loc, _) in Grid.Index[1][true]
   (_, c) = NeighborCells.Index[0][loc]
   if (c<2||c>3)
      add loc to the table
```

Here, *predicate*`.Index[`*columnNumber*`]` is the index for the specified column. It's a hash table mapping column values to rows (key index) or linked lists of rows (non-key). So `Grid.Index[1][true]` is the list of all rows whose second column is true, and `NeighborCells.Index[0][loc]` is the unique row that has `loc` as its first column. Whereas, the previous version ran in $O(g^2)$ time, this runs in $O(c)$ time where c is the number of cells, a dramatic speedup.

While still not the preferred way to implement *Life*, it's efficient enough to run at 120Hz on a single core of an i9-9900K, including Unity's graphics code.

Logic programming lets us write loops declaratively: the iteration structure is implicit in the conditions listed in a rule. By precomputing results and storing them in tables, bottom-up logic programming lets us **optimize** declaratively. Indices can be added to tables incrementally during development, as new access patterns are introduced. Crucially, adding indices requires only adding an annotation to the predicate declaration; no other action is required. Rules need no modification.

4.2. Structure of a TED program

As discussed, a TED program is a C# program builds the run-time representation of the TED program, then calls into it as necessary. Declarations create syntax trees representing the code, then preprocess them to create the run-time representation used by the interpreter. Predicates, calls, rules, etc. are all represented at run-time as typed C# objects.

TED relies on liberal use of operator overloading to make TED code look as natural as possible, even though it is "really" a series of constructor calls for

syntax trees. Square brackets denote calls to predicates (C# allows the bracket operator to be overloaded, but not the call operator).

4.2.1. Terms and variables

Following the terminology used in logic and logic programming, the expressions used as arguments to predicates are known as terms.

Since, again, TED code is really C# code that builds the syntax tree for the TED code to be executed, a term such as `x+1` is represented as a data structure such as:

```
var x = new Var<int>("x");
new FunctionalExpression<int>(Add, x,
    new Constant<int>(1))
```

However, overloading allows the programmer to type `x+1` and have it converted to the constructor call above. The programmer can largely ignore the internal representation.

The one place the programmer does need to be aware of terms and their data types is with variables. As shown above, a TED variable is represented as a C# object of type `Var<T>` where `T` is the type of the variable's value. Before using a variable in a rule:

```
P[x].If(Q[x]);
```

which states that P is true of a value x if Q is true of it, the programmer needs to first define `x` as *a C# variable containing a TED variable*, as with the declaration above. Since this is somewhat cumbersome, it can be shortened to:

```
var x = (Var<int>)"x";
```

The declaration specifies both the name of the variable and its type. Once a variable is defined, it can be used in multiple rules, but is treated as a separate local variable for each rule. While the declaration syntax is annoying, one can keep the number of variable declarations to a manageable level.

Note that the program fragment shown in figure 2, uses the variables `loc`, `neighbor`, `occupied`, and `count`. The declarations for these were withheld until now and are as follows (`Vector2Int` is Unity's standard data type for grid locations):

```
var loc = (Var<Vector2Int>)"loc";
var neighbor = (Var<Vector2Int>)"…";
var occupied = (Var<bool>)"occupied";
var count = (Var<int>)"count";
```

4.2.2. Predicates

Predicates are C# objects of type `Predicate<`T_1`, `T_2`, …, `T_n`>`, where n is the number of arguments to the predicate and the T_i are their respective types. The predicates we've discussed so far are **table predicates**. They are represented as tables, and queries to them are implemented as searches on the table and its indices. Table predicates have a `.If()` method to add rules to them. Table predicates are created using the Predicate method:

`Predicate(`*name*, arg_1, arg_2, …, arg_n`)`

where *name* is a string used for identifying the predicate in error messages, and the arg_i are variables with the desired types for each argument (these jointly define the type of the predicate). Thus the declaration:

```
var Grid=Predicate("Grid",loc,occupied);
```

sets `Grid` to a `Predicate<Vector2Int, bool>`, i.e. a predicate with `Vector2Int` and a `bool` arguments.

In addition to table predicates, there are **primitive predicates**, which are directly implemented as C# methods. TED includes many built-in primitives, such as the `<` operator, used in figure 2.

Finally, TED allows **definitions**, predicates defined by rules that are inlined into any calls. For example,

```
var CellAt = Definition("CellAt", loc)
                .If(Grid[loc, true])
```

states that `CellAt` is a predicate over a `Vector2Int`, but queries of the form `CellAt[`x`]` should be replaced by the query `Grid[`x`, true]`.

4.2.3. Goals and rules

Procedure calls in Prolog and TED are called goals. Applying a predicate to a set of terms of the correct types returns a `Goal` object: the syntax tree for a call. Goals for table predicates include an `If(Goal…)` method that takes a series of other `Goals` as arguments, constructs a rule from them, and adds the rule to the original `Goal`'s predicate. Our example declaration:

```
P[x].If(Q[x]);
```

creates a rule object stating that $\forall x.\, Q(x) \rightarrow P(x)$, and adds it to the list of `P`'s rules. For convenience, table predicates also have their own `If` methods, allowing rules to be combined with predicate definition. Thus:

```
var P = Predicate("P", x).If(Q[x]);
```

is equivalent to:

```
var P = Predicate("P", x);
P[x].If(Q[x]);
```

4.2.4. Higher-order predicates

Higher-order predicates are predicates parameterized by goals or other predicates. TED includes a number of these, as well as facilities for defining one's own. Table predicates cannot be higher order.

Definitions and primitive predicates can be made higher order, simply by having one of their arguments be of type `Goal`. TED includes a number of built-in higher-order primitives:

- **Logical connectives**: `And[]`, `Or[]`, and `Not[]`
- **Optimization primitives**: `Maximal` and `Minimal`, as used in Figure 1.
- **Flow-control** predicates that execute the goal in some modified manner, such as `Once[]`.
- **Aggregation functions:** `Count`, `Sum`, and `Aggregate` as used in Figure 2.

User-defined higher-order primitive predicates are allowed but are currently more involved to write than other user-defined primitives.

4.2.5. Table operators

Operators map tables to tables. They encapsulate algorithms that execute over a table as a whole, returning a new table as a result. One example is `CountsBy`, which makes pivot tables. If the table `Population` is a table of characters in the game and it has a column called `sex`, then the declaration:

```
var Demographics =
      CountsBy("Demographics",
               Population, sex, count);
```

defines a new `Predicate<sex,int>`, listing the number of characters with each sex in the current step of the simulation.

Most other operators encapsulate graph algorithms. For example, if `R` is a table predicate of type `Predicate<T,T>` for some `T`, for example, representing the edges in a graph whose vertices are objects of type `T`, then:

```
var RStar = Closure("R*", R);
```

makes a new predicate, `RStar`, also of type `Predicate<T,T>`, such that `RStar[`a`,`b`]` holds iff b is reachable from a via edges in `R`.

A number of operators implement different forms of graph matching. If `Interested` is a `Predicate<Person,Person>` describing who is interested in dating whom, then:

```
var D = MatchRandomly("D", Interested);
```

makes a new table, `D`, that on any step of the simulation contains a subset of the rows of `Interested` such that no person is listed in two different rows. If we add another column to `Interested` specifying a level of interest, then:

```
var D = MatchGreedily("D", Interested);
```

will attempt to choose a matching with the highest interest levels possible (although it not necessarily globally optimal).

If `Interested` is a relation not between people, but between people and jobs (so it is of type `Predicate<Person,Job,float>`), and if `Capacity` is a `Predicate<Job,int>` listing how many openings there are for each `Job`, then

```
var E = AssignGreedily("A",
                       Interested, Capacities);
```

makes a new table, `E`, matching people to jobs with the highest possible interest level, without assigning more to a job than there are openings.

4.2.6. Table update

Base tables can be updated by providing tables of changes to perform.

If B is a base table of type `Predicate<`T_1`, …, `T_n`>`, then B`.Add` is a table of the same type whose rows are appended to B at the end of each simulation step. Thus, B`.Add.If(…)`, which adds a rule to B`.Add`, effectively specifies a rule for when to add a row to B. There is not presently a B`.Delete`, but there are plans to add it.

Individual columns can be changed by providing tables of changes to make. B`.Set(`*key*`,` *updateColumn*`)`, where *key* and *updateColumn* are columns of B, i.e. variables specified in B's definition, returns a table with the columns *key* and *updateColumn*. At the end of each update step, the system will iterate through the rows of the B`.Set` table, and for each row, use B's index to find the row in B with the specified key, then change the value of that row's *updateColumn* to the value listed in the B`.Set` table. Thus, the rule:

```
Population.Set(who, status)
     .If(Died(who), status=Status.Dead)
```

Would update the `status` column of the `Population` table for someone who dies to `Status.Dead`.

4.2.7. Invariant checking

Every TED program has two built-in base tables. `Exceptions` lists all the exceptions that have been thrown while running rules, together with the tables and rules that threw them. `Problems` lists invariants and other assertions that have been violated. To declare an invariant, simply write a rule of the form:

```
Table.Problem.If(…);
```

If the rule ever succeeds, it will add a line to the `Problems` table listing *Table*, the rule, and the values of all variables in the rule. Problem rules are like assertions in other languages in that checking of them can be enabled and disabled and they impose no run-time penalty when disabled. Unlike most languages, however, TED problem rules can be enabled and disabled at run-time without recompilation.

5. Implementation

TED is highly optimized. Most code can run without run-time type checks or storage allocation, apart from the initial allocation or reallocation of the tables themselves.

5.1. Table representation

Table data for a `Predicate<T1, …, Tn>` is stored in a packed array of tuples of type `(T1, …, Tn)`, one per table row. Tuples are value types; they are stored in-line in the array, rather than separately represented in the heap. This means table data is stored as a contiguous sequence of bytes, without boxing. Moreover, tables are almost always scanned in order, so table operations have best-case cache locality.

Table operations are optimized to avoid copying of value types. Generics are used for most table operations, to avoid the need for boxing or run-time type checking. Compiled code therefore looks largely like what one would get with hand-written C code, apart from the use of out-of-line calls for things like equality comparison.

For derived tables, which are recomputed on each simulation step, the array is reused from step to step. When table data overflows the array, the array size is doubled, guaranteeing amortized $O(1)$ performance.

Indices are implemented as custom hash tables mapping keys to row numbers. To maximize cache performance, the current implementation uses direct addressing with linear probing. To avoid clustering, the hash tables keep their load factors below 0.5, making clustering unlikely. Indices use 8 bytes per row per index, for key indices, 12 for general indices.

Tables can also be declared to have unique rows (i.e. they are sets rather than multisets). They maintain a hash set of all rows and ignore duplicate additions.

5.2. Rule execution

A rule, created using the `If()` method, is specified by a `Goal` object, known as its **head**, that forms its conclusion, and a sequence of `Goals`, known as its **body**, that form its conditions. The actual internal representation of a rule consists of a series of iterators for each goal in the body. Rules are transformed into their iterators through a preprocessing mechanism.

5.2.1. Unification: read-mode and write-mode

Goals, such as `P[x, y, 7]`, are matched against tuples in tables using unification [20], which computes the solution to a set of simultaneous equations over terms. In its usual form in languages such as Prolog, unification can equate variables to other variables. Thus, Robinson's original unification algorithm [21] dynamically computes an equivalence relation over variables using the disjoint set partition algorithm [22]. This requires variables to store forwarding pointers when they are equated to some other variable. Looking up the value of a variable involves looping over forwarding pointers until an unaliased variable is found. Moreover, equations between variables must be backtrackable, requiring a mechanism for undoing the aliasing.

In Datalog-like languages, including TED, unification is only performed between goals and table tuples that cannot contain variables. Variables can therefore only be unified with data values, not other variables, removing the need to track aliases.

Variables therefore behave much like C variables; they are simply typed locations in memory. The first time a variable is unified with a value, that value is stored in the location. This is referred to as **write mode**. Subsequent uses of the variable later in the rule, where the variable is unified with other data values, is implemented by testing equality between the previously stored value and the new value. This is referred to as **read mode**.

For example, the goal `P[x, x, 1]` can be unified with a tuple (a, b, c) from `P`'s table by first storing a in `x`, then testing if that stored value is equal to b, and finally testing whether $c = 1$. It's essentially equivalent to the C# code:

```
bool CanUnifyXX1(int a, int b, int c) {
   x = a;
   return x == b && c == 1;
}
```

Running this against a particular a, b, c, tests if they're unifiable, while updating `x` to its new value. It requires no binding lists, pointer chasing, or type checking. It's much faster than the general unification algorithm.

This is the general approach used for unification in TED. The body of a rule is scanned and the first (leftmost) occurrence of each variable is found and marked write mode. All other variable occurrences, as well as constants, are read mode. A goal unifies with a tuple if each goal argument unifies with its respective tuple element. A write-mode variable aways successfully unifies, and updates the variable. A read-mode variable or constant unifies with a value only if they are equal.

5.2.2. Body normalization

The first step of preprocessing is to reduce the rule body to a normal form in which:

- Functional expressions are hoisted out of calls to predicates other than the built-in primitive predicate `Eval[]`. Thus, `P[x+1]` is transformed to: `And[Eval[t,x+1],P[t]]`
- Calls to `And[]` and `Or[]` are flattened; that is, `And[a, And[b, c]]` is simplified to `And[a, b, c]`.
- Any goals that can be partially evaluated are reduced to simpler goals. If their truth values are known at preprocessing time, they are replaced with `true` or `false`. This includes simplifying `And`, `Or`, and `Not`. It is possible for a body to simplify down to just `false`, in which case an warning is printed.

5.2.3. Iterator selection

Finally, the `Goals` of the body, which again are just syntax trees, are mapped to iterators to be executed at runtime. For goals involving primitive predicates, the primitive implements its own custom iterator. The preprocessor leaves it to the primitive to choose it.

For table predicates, the iterator must iterate through the rows of the table, unify them with the

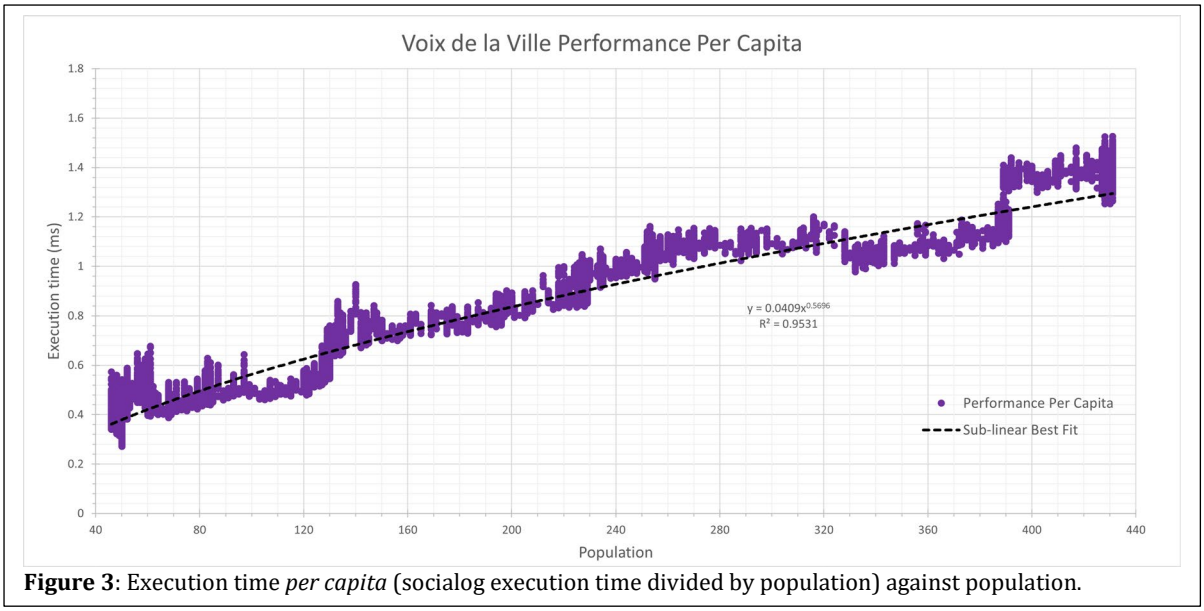

Figure 3: Execution time *per capita* (socialog execution time divided by population) against population.

arguments in the goal, and generate only the ones that match. The preprocessor attempts to choose iterators that use indices when possible. In decreasing order of preference, these are:

- If the table is declared to have **unique rows** and all goal arguments are read-mode, then the goal is implemented as a single test against the table's hash-set of rows, with no iteration or unification.
- If the table has a **key index** for a read-mode argument in the goal, the goal is implemented as a lookup of this row and a single test of whether it unifies with the arguments.
- If the table has a **non-key index** for a read-mode argument, the goal is implemented as lookup of the linked list of rows with the specified column value, followed by an iteration over those rows, unifying them with the arguments.
- If **no index** is available, the goal is implemented as an iteration over all rows of the table, attempting to unify each with the goal arguments.

There is no hinting mechanism to allow the programmer to advise the system on how to choose between multiple non-key indices at present. Nor is there a way to make combined indices over multiple columns. However, these are planned additions.

5.2.4. Iterator sequencing

Execution consists of running the iterators in order. When an iterator succeeds, it updates any write-mode variables it unified with the selected table row, and execution proceeds to the next iterator. When an iterator fails, execution backs up to the previous iterator which generates its next match, if any.

When the last iterator succeeds, all variables have been matched to values during the unification process. The system forms a tuple from the arguments in the head goal, fills it in with the values of the relevant variables, and appends it to the table. It then returns to the last goal's iterator to generate the next tuple, which may involve the last iterator failing, and asking for the next solution from the previous iterator, which may fail, etc.

When the first iterator fails, the rule has generated all its tuples for the predicate and execution is complete.

5.3. Simulation control flow

Simulation proceeds by repeatedly recomputing derived predicates, then updating base predicates. Recomputation of a derived predicate works by recursively updating the predicates used its rules, if they haven't already been updated, then recomputing the table for the derived predicate. Updates of base tables are specified by update tables, which are derived tables defined by rules, as discussed previously.

Bottom-up logic programming can support recursive rules at the cost of a significantly more complicated algorithm (a fixed-point iteration [23]). The classical use case for recursion in Datalog is translative closure a binary relation (e.g. reachability in a graph). However, TED can compute reachability in an undirected graph in $O(V + E)$ time and $O(V)$ space using its `EquivalenceClass` operator, whereas the classical recursive solution in Datalog requires at least $O(V^3)$ time and $O(V^2)$ space, depending on the algorithm used and the ability of the system to index the relevant tables. We have deferred supporting recursion until we have a compelling use case.

5.4. Parallel execution

Alternatively, TED programs can be executed in parallel. Each predicate is updated in a separate task. Once a table is updated, all further access are read-only, so parallel update requires no locking or other mutual exclusion. The "continuation" feature of the C# task parallelism library is sufficient to guarantee tasks are not scheduled until the tasks they depend on have been completed, so there is no explicit blocking.

Since each predicate is updated by a single task, the level of parallelism depends on the number of

predicates in the program and the length of their serial dependencies.

6. Performance

TED is currently used in two projects. It is being used for consistency-checking the asset database of an unannounced commercial game. It was chosen over Unity Prolog [24] because of its strong typing, and because its embedded design made interoperability with C# easier. However, it is not being used for character simulation thus far.

The main project using TED is *Voix de la Ville*, a declarative reimplementation of a subset of Ryan's *Talk of the Town* city simulator [14]. The current system supports small hundreds of characters interacting with one another in a city with 44 different kinds of buildings and 62 different kinds of professions. Single-core execution times on a single core of an i7-7700K running at 4.8GHz are show in figure 3. These are per capita execution times, i.e. the execution time of a simulation tick, divided by the population. They represent a combination of character updates, which are fixed cost per character, and per-relationship updates, which are inherently quadratic. The core simulation is ~500 lines of code, including comments.

7. Related work

Many AI-based games have used symbolic rules for character control. *MKULTRA* [7] was written primarily in Prolog, save for the graphics and UI code. *City of Gangsters* [8] used another top-down logic programming language, albeit with an exotic implementation. Many other games have used some kind of rule engine. *Façade* [25] was implemented primarily in a reactive planning language, *ABL* [26], however its internal working memory included a forward-chaining production system. Similarly, *Prom Week* [18] used a forward-chaining production system implemented in Javascript. *The Sims 3* [27] also used a rule-based system to script the interactions between situations, personality traits, and actions available to a given character [28].

Several game development frameworks and social simulation middleware have used symbolic rules, particular for interactive narrative. One of the earliest and most influential such systems is the Nelson's *Inform 7* language [4], [29], which allows designers to build interact narrative systems, particularly simulationist systems, using declarative statements. The *Versu* simulationist narrative system [30], [31] used a custom logic programming language, *Praxis*, which was based on an exotic modal logic called eremic logic (aka exclusion logic) [6]. More recently, the *Lume* system [32] made extensive use of Prolog's definite clause grammars [33], [34] for text generation. Lapeyrade has also used Prolog for better character decision making [35]

Several systems have used forward-chaining rule-based systems, including *ABL*, *Comme Il Faut* [5], the social simulation engine upon which *Prom Week* was built, and the *Ensemble Engine* [36], *CiF*'s successor.

To our knowledge, bottom-up logic programming has not previously been used to implement social simulations. However, Datalog has been used for story-sifting [15], the process of searching the output of a social simulator for interesting narrative content.

Bottom-up logic programming, and Datalog in particular, has received the most attention in the database community [13], [37], [38], where its appeal came partly from the ability to compile it into relational algebra operators for efficient execution on classical database architectures, and because it can be extended to recursive rules using a fixed-point evaluation algorithm. This allows it to compute transitive closure (e.g. reachability in a graph), which standard relational algebra cannot. This is where most of the original research on the language and its implementation was done. More recently, it has seen extensive use for the semantic web [39].

Games involving large-scale social simulation are relatively rare. The best known is *Dwarf Fortress* [40], which supports real-time simulator of small hundreds of characters. Achieving this level of performance requires implementation in C++ and significant programmer effort to optimize cache locality and minimize the number of pointer indirections.[3] *RimWorld* is very similar game that also involves social simulation for the purpose of storytelling [41]. In the research literature, the best known system is Ryan's *Talk of the Town*, [14], which was used in the award-winning game *Bad News* [16], [17]. TotT was a batch simulation of the growth of a small American town over the course of 140 years, ending with population around 400 people using a time-varying level of detail. It was implemented in Python and required many minutes to simulate a city. More recently, Johnson-Bey has developed *Neighborly* [42], a more modular and modifiable implementation based on an entity-component-system architecture [3]. With the possible exception of *RimWorld*, these systems run the simulation in a single thread.

Kismet [43] is a rapid-prototyping system for social simulations intended for casual users. It used answer-set programming (a type of logic programming) internally. However, its focus was on allowing casual users to build social simulations, rather than on trying to maximize performance.

8. Conclusion

TED is a high-performance, embedded, parallelizable, logic programming system that allows game designers to quickly and conveniently implement large-scale social simulations and run them quickly on modern, multi-core architectures. It allows story-sifting and simulation to be written in the same language. The use of tables to store all intermediate results aids debugging by making all intermediate results inspectable and queryable at run-time.

[3] Tarn Adams, personal communication.

References

[1] R. Zubek, "Needs-Based AI," in *Game Programming Gems 8*, Cengage Learning PTR, 2010.

[2] W. Wright, "The Sims." MAXIS/Electronic Arts, 2000.

[3] "Entity Systems are the future of MMOG development – Part 1 – T-machine.org," Jul. 31, 2013. https://new.t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/ (accessed Jul. 18, 2023).

[4] G. Nelson, "Inform 7." 2006.

[5] J. McCoy, M. Treanor, B. Samuel, N. Wardrip-Fruin, and M. Mateas, "Comme il Faut: A System for Authoring Playable Social Models," in *Proceedings of the 7th AI and Interactive Digital Entertainment*, V. Bulitko and M. O. Riedl, Eds., Stanford, CA: AAAI Press, 2011.

[6] R. Evans, "Introducing Exclusion Logic as a Deontic Logic," in *Deontic Logic in Computer Science, Proceedings of the 10th International Conference, DEON 2010, Lecture Notes in Computer Science Volume 6181*, Fiesole, Italy: Springer, 2010, pp. 179–195.

[7] I. Horswill, "Postmortem: MKULTRA, An Experimental AI-Based Game," *AIIDE*, vol. 14, no. 1, pp. 45–51, Sep. 2018, doi: 10.1609/aiide.v14i1.13027.

[8] SomaSim, "City of Gangsters." Chicago, 2021.

[9] D. H. D. Warren, L. M. Pereira, and F. Pereira, "PROLOG - The Language and its implementation compared with LISP," in *Symposium on AI and Programming Languages*, ACM, 1977, pp. 109–115. doi: 10.1145/800228.806939.

[10] W. F. Clocksin and C. S. Mellish, *Programming in Prolog: Using the ISO Standard*. New York, NY: Springer, 2003.

[11] M. H. Van Emden and R. a. Kowalski, "The Semantics of Predicate Logic as a Programming Language," *Journal of the ACM*, vol. 23, no. 4, pp. 733–742, 1976, doi: 10.1145/321978.321991.

[12] "Bottom-up beats top-down for datalog | Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems." https://dl.acm.org/doi/10.1145/73721.73736 (accessed Jul. 21, 2023).

[13] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about Datalog (and never dared to ask)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146–166, Mar. 1989, doi: 10.1109/69.43410.

[14] J. Ryan, "Curating Simulated Storyworlds," University of California Santa Crus, 2018.

[15] M. Kreminski, M. Dickinson, and N. Wardrip-Fruin, "Felt: A Simple Story Sifter," R. E. Cardona-Rivera, A. Sullivan, and R. M. Young, Eds., in Lecture Notes in Computer Science, vol. 11869. Cham: Springer International Publishing, 2019, pp. 267–281. doi: 10.1007/978-3-030-33894-7_27.

[16] J. O. Ryan, B. Samuel, and A. Summerville, "Bad News : A Game Of Death And Communication," pp. 160–163, 2016.

[17] B. Samuel, J. Ryan, A. J. Summerville, M. Mateas, and N. W. Fruin, "Bad news: An experiment in computationally assisted performance," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2016. doi: 10.1007/978-3-319-48279-8_10.

[18] J. McCoy, M. Treanor, B. Samuel, and A. A. Reed, "Prom Week." Expressive Inteligence Studio at UC Santa Cruz, Santa Cruz, California, 2012.

[19] M. Gardner, "Mathematical Games - The fantastic combinations of John Conway's new solitaire game 'life,'" *Scientific American*, no. 223, Oct. 1970, doi: doi:10.1038/scientificamerican1070-120.

[20] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2009.

[21] J. A. Robinson, "Computational logic: The unification computation," in *Machine Intelligence 6*, Edinburgh University Press, 1971, pp. 63–72.

[22] T. H. Cormen, C. E. Leiserson, and R. R. L., *Introduction to Algorithms*. MIT Press, 1990.

[23] M. Alvarez-Picallo, A. Eyers-Taylor, M. Peyton Jones, and C.-H. L. Ong, "Fixing Incremental Computation," in *Programming Languages and Systems*, L. Caires, Ed., in Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 525–552. doi: 10.1007/978-3-030-17184-1_19.

[24] I. Horswill, "Unity Prolog." Dec. 20, 2022. Accessed: Jul. 21, 2023. [Online]. Available: https://github.com/ianhorswill/UnityProlog

[25] M. Mateas and A. Stern, "Façade." 2005.

[26] M. Mateas and A. Stern, "A Behavior Language for Story-Based Agents," *IEEE Intelligent Systems*, vol. 17, no. 4, pp. 39–47, 2002.

[27] Maxis, "The Sims 3." 2009.

[28] R. Evans, "AI Challenges in Sims 3," in *Artificial Intelligence and Interactive Digital Entertainment*, Stanford, CA: AAAI Press, 2009.

[29] G. Nelson, "Natural Language, Semantic Analysis, and Interactive Fiction." Unpublished white paper, Cambridge, UK, 2006.

[30] R. Evans and E. Short, "Versu." Linden Lab, San Francisco, CA, 2013.

[31] R. Evans and E. Short, "Versu - A Simulationist Storytelling System," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 2, pp. 113–130, 2014.

[32] S. Mason, C. Stagg, N. Wardrip-fruin, and M. Mateas, "Lume: A System for Procedural Story Generation," in *The Fourteenth International Conference on the Foundations of Digital Games (FDG '19)*, San Luis Obispo, CA, USA, 2019.

[33] F. C. N. Pereira and D. H. D. Warren, "Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks," *Artificial Intelligence*, vol. 13, no. 231–278, 1980.

[34] F. C. N. Pereira and S. Shieber, *Prolog and Natural Language Analysis*. Brookline, MA: Microtome Publishing, 1987.

[35] S. Lapeyrade, “Reasoning with Ontologies for Non-player Character’s Decision-Making in Games,” *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 18, no. 1, Art. no. 1, Oct. 2022, doi: 10.1609/aiide.v18i1.21980.
[36] B. Samuel, A. A. Reed, P. Maddaloni, M. Mateas, and N. Wardrip-Fruin, “The Ensemble Engine: Next-Generation Social Physics”.
[37] J. D. Ullman, *Ullman: Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.
[38] J. D. Ullman, *Ullman: Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies*. Computer Science Press, 1989.
[39] G. Gottlob, G. Orsi, A. Pieris, and M. Šimkus, “Datalog and Its Extensions for Semantic Web Databases,” in *Reasoning Web. Semantic Technologies for Advanced Query Answering: 8th International Summer School 2012, Vienna, Austria, September 3-8, 2012. Proceedings*, T. Eiter and T. Krennwallner, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 54–77. doi: 10.1007/978-3-642-33158-9_2.
[40] T. Adams and Z. Adams, “Slaves to Armok: God of Blood Chapter II: Dwarf Fortress.” Bay 12 Games, 2006.
[41] T. Sylvester, “RimWorld.” Ludeon Studios, Oct. 2018.
[42] S. Johnson-Bey, M. J. Nelson, and M. Mateas, “Neighborly: A Sandbox for Simulation-based Emergent Narrative,” in *2022 IEEE Conference on Games (CoG)*, Beijing, China: IEEE, Aug. 2022, pp. 425–432. doi: 10.1109/CoG51982.2022.9893631.
[43] B. S. Samuel, “Kismet: A Small Social Simulation Language,” Jan. 2021, Accessed: Jul. 24, 2023. [Online]. Available: https://www.academia.edu/101779750/Kismet_A_Small_Social_Simulation_Language