Proceedings of the Twentieth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2024)

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

Ian Horswill, Samuel Hill
Northwestern University, 2233 Tech Drive, Evanston, IL, 60208, USA
ian@northwestern.edu, SamuelHill2022@u.northwestern.edu

Abstract

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, they are often quite slow, due in part to its heavy use of pointer chasing, dynamic allocation, garbage collection, and runtime type-checking, making it difficult to use for large numbers of characters or high-frequency updates.

For appropriate tasks, bottom-up execution of logic programs can provide the declarativity of logic programming without its performance issues. We argue that large-scale character simulations are a "sweet spot" for bottom-up LP. We present a language, TED, that combines declarativity with excellent performance. TED can be used with any game engine supporting C#. In head-to-head comparisons TED code was 2-3 orders of magnitude faster than Prolog, 2-5 times more compact than C#, and only 25% slower than C#. It is used in both the research game *Voix de la Ville* and the upcoming commercial game *Rise of Industry 2*.

Introduction

Game loops and other simulations often involve iterating through data structures representing world state. For example, the needs-based AI (Zubek 2010) of *The Sims* (Wright 2000) involves finding for each character the object that best satisfies its various needs:

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

A naïve implementation of involves three nested loops running over the C characters, O objects, and N needs, with running time $O(CON)$. This can be improved upon by maintaining separate lists of just the characters that need to be updated, just the objects that are available to satisfy a specific need, and so on. But this requires modifying unrelated parts of the program to update those lists, increasing inter-module dependencies and development costs.

Ideally, one would specify the maximization separately from the data layout, as one does in relational databases. Layout could be tuned independently to best support the mix of queries needed. As those queries inevitably change, the layout could be retuned without modification of the code. Similar arguments have been made in the context of entity-component systems for massively online games (Martin 2013).

In this paper, we describe a high-performance logic programming language, TED, designed for rapid development of large-scale character simulations such as *Talk of the Town* (J. Ryan, 2018) and *Dwarf Fortress* (Adams and Adams 2006). TED can compactly express the need maximization algorithm in two lines of code (see Listing 1). It can also be used with any game engine that supports C# (e.g. Unity, Godot).

In direct comparisons, TED code is 2-5 times shorter than the equivalent C# code. Interpreted, its performance easily exceeds Python's, and compiled, it is competitive with hand-written C#. It also supports parallel execution without modification. Its performance is due in large part to its bottom-up execution strategy, discussed below.

TED has been used in the research game *Voix de la Ville* (Hill and Horswill, Ian 2023) and the forthcoming commercial game *Rise of Industry 2* (Viglione 2025).

Related Work

Many AI-based games have used logic or other symbolic rules for character control. Many other games have used some kind of rule engine. *Façade* (Mateas and Stern 2005) used a reactive planner, *ABL* (Mateas and Stern 2002) that incorporated a forward-chaining production system. *Prom Week* (Josh McCoy et al. 2012) also used a forward-chaining production system. *The Sims 3* (Maxis 2009) used a rule system to script the interactions between situations, personality traits, and actions available to a given character (Evans 2009). *MKULTRA* (Horswill 2018) and *City of Gangsters* (SomaSim 2021) both used logic programming to implement character behavior.

Listing 1: Needs-based action selection in TED

```
var Score = Definition("Score", c, o, t).If(O[o], t==Sum(s, And[N[n], s==S[c,o,n]));
var B = Predicate("B", c, o).If(C[c], Maximal(o, t, Score[c, o, t]));
```

Several game development and social simulation frameworks have used symbolic rules. One of the earliest and most influential is the Inform 7 language (Nelson 2006a; 2006b) that allows designers to build interact narrative systems using declarative statements. The *Versu* simulationist narrative system (Evans and Short 2013; 2014) used a custom logic programming language, Praxis, based on an exotic modal logic called eremic logic (aka exclusion logic) (Evans 2010). The *Lume* system (Mason et al. 2019) made extensive use of Prolog's definite clause grammars (Pereira and Warren 1980; Pereira and Shieber 1987) for text generation. Lapeyrade has also used Prolog for better character decision making (Lapeyrade 2022)

Several systems have used forward-chaining rule-based systems, including *ABL*, *Comme Il Faut* (Joshua McCoy et al. 2011), the social simulation engine upon which *Prom Week* was built, and the *Ensemble Engine* (B. Samuel et al., n.d.), *CiF*'s successor.

Datalog has been used for story-sifting (Kreminski, Dickinson, and Wardrip-Fruin 2019), the process of searching the output of a social simulator for interesting narrative content. Bottom-up LP, and Datalog in particular, has received the most attention in the database community (Ceri, Gottlob, and Tanca 1989; Ullman 1988; 1989), 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. More recently, it has seen extensive use for the semantic web (Gottlob et al. 2012).

Games involving large-scale social simulation are relatively rare. The best known is *Dwarf Fortress* (Adams and Adams 2006), which supports real-time simulator of small hundreds of characters. Achieving this level of performance requires implementation in C/C++ and significant programmer effort to optimize cache locality and minimize pointer chasing. *RimWorld* is very similar game that also involves social simulation for the purpose of storytelling (Sylvester 2018).

In the research literature, the best known large-scale character simulation is Ryan's *Talk of the Town*, (J. Ryan 2018), which was used in the award-winning game *Bad News* (J. O. Ryan, Samuel, and Summerville 2016; B. Samuel et al. 2016). *TotT* simulates a small 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* (Johnson-Bey, Nelson, and Mateas 2022), a more modular and modifiable implementation based on an entity-component-system architecture (Martin 2013).

Kismet (B. S. Samuel 2021) 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 or expressivity.

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.

A rule gives a set of conditions implying the truth of a predicate. For example, siblinghood can be defined by:

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

which states two people are siblings if they share a parent.

Top-Down Execution

Classical logic-programming languages execute queries "top-down" using SLD resolution (Van Emden and Kowalski 1976). They compute a single solution, avoiding unnecessary work when not all solutions are needed, but duplicating it when the same call is repeated. Unification in these systems is expensive, requiring extensive use of forwarding pointers.

Bottom-Up Execution With Tables

An alternative is to compute the full extensions of each predicate (all values for which it's true) before they are used, and then store them in tables. If the full extension (all the child/parent pairs) of the `Parent` relation is already stored in a table, we can compute the table for the full extension of the `Sibling` relation with the loop:

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

Listing 2: Cellular automaton in TED

```
var Grid = Predicate("Grid", loc.Key, occupied.Indexed);
var NeighborCells = Predicate("Neighbors", loc.Key, count.Indexed)
  .If(Grid[loc, __], count==Count(And[Neighbor[loc, neighbor], Grid[neighbor, true]]));
Grid.Set(loc,occupied, false).If(Grid[loc,true],Neighbors[loc,count],(count<=2|count>3));
Grid.Set(loc,occupied, true).If(Neighbors[loc, 3]);
```

This is undesirable when we don't need the full extension. However, it's extremely efficient when we do. Rather than using Prolog-style "logic variables", unification is implemented through simple loads, stores, and equality tests. Moreover, if we index the Parent table by its second argument, we optimize it:

```
for each (x,p) in Parent
   for each (y, p2) in Parent.Index[2][p]
      Sibling.Add( (x,y) )
```

The decision of what columns of what tables to index can be made separately and tuned at one's leisure. The compiler compiles the `If()` rule above to the most efficient loop given the indices available.

This is the "bottom-up" execution strategy: for each predicate, compute its full extension as a table. To compute that extension, first ensure that any other predicates referenced by its rules have been computed, the run the rules. This strategy and variations on it are used in Datalog (Ceri, Gottlob, and Tanca 1989).

Declarative Simulation

We believe character simulation is a "sweet spot" for bottom-up logic programming. Large scale character simulations are often defined in terms of rules, and the engine generally does have to compute complete extensions anyway. Moreover, the entire state of the game is then visible to queries, for programming, visualization, and debugging.

This work was motivated by Ryan's argument for generating stories by running a city-scale character simulation and then searching ("sifting") its output to find interesting stories (J. Ryan 2018). Story sifting requires a query language that can be run against the simulation.

We begin from the question: *what if the query language could also be the simulation language?* That is, can we write character simulation logic declaratively in the query language? If so, can it be performant?

Declarative simulations can be built using three main types of predicates/relations, referred to henceforth as tables:

- **Base tables** store the state of the simulation. They retain their data from one tick (simulation step) to the next, except insofar as they're modified by update tables.
- **Derived tables** are defined in terms of other tables using `If()` rules. Derived tables are recomputed on each tick.
- **Update tables** are lists of modifications to be made to base tables. They are themselves a kind of derived table defined by rules. The engine applies these updates to the base tables at the end of each tick.

The TED Language

TED is a high-performance, bottom-up logic programming language intended for character simulation in AI-heavy games. Originally a typed, embedded version of Datalog (hence the acronym TED), it is strongly-typed, supports higher-order predicates, metaprogramming, parallel execution, optional native code compilation, and an optional runtime parser-evaluator permitting interactive queries within the game.

TED is embedded in C#, meaning that TED code is C# code that builds the TED program in memory. This means C# interoperation is largely transparent, and TED code inherits IDE support such as type checking from the underlying environment. It also allows C# to be used as a macro language for metaprogramming, see (Hill and Horswill, Ian 2023) for an example.

TED is less expressive than Prolog: it does not allow Prolog "functors" or recursion within a given simulation step. Recursion can be implemented across simulation steps, or by calling C#, but this has not been necessary for the kinds of games TED is designed for. In exchange for these limitations, we get high performance and parallelizability.

Trivial Example

Listing 2 shows a cellular automaton, Conway's Life (Gardner 1970), implemented as a 4-line 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/table, `Grid`, to hold the board state. It has two arguments/columns: location and whether the location is occupied by a cell. It's a

base table: it retains its state from tick to tick except as specified by the `Set()` methods at the end.

The second statement defines `NeighborCells`, mapping locations in the grid to the number of cells surrounding them. The `.If()` call adds a rule: `NeighborCells[loc,count]` is true if

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

The last two statements define modifications to make to `Grid` at the end of each tick. The first clears the occupied column of any location with less than 2 or more than 3 neighbors, removing those cells. The second sets it to true for locations with exactly 3 neighbors, spawning cells.

It should be noted that there more efficient algorithms for *Life.* Nonetheless even the interpreted version of this TED code is fast: it runs at 120FPS on a single core of an i9-9900K, including Unity's graphics code.

Declarative Data Structures

The `.Key` and `.Indexed` annotations tell the system to index the tables by the specified columns. The `.Key` annotation declares values in that column are unique, so key indices map column values to single rows. Non-key indices map values to sets of rows.

Both predicates are keyed by location. Given a location, we can find its occupancy or neighbor count in $O(1)$ average time. They also index their second column; we find rows with/without cells or rows with a given number of neighbors, in $O(1)$ time.

Structure of a TED Program

Being embedded in C#, TED makes liberal use of operator overloading to allow code to look as natural as possible, even though it is "really" a series of constructor calls for syntax trees. C#'s limitations on overloading lead to some unfortunate syntactic conventions, most notably the use of square brackets for calls to most (but not all) predicates.

Terms and Variables

Following logic and logic programming convention, arguments to predicates are known as *terms*. Constants largely behave as in C#. However, variables must be represented as objects in the parse tree and so must be declared using declarations of the form:

```
var name = (Var<Type>)"name";
```

Once a variable is defined, it can be used in multiple rules but it 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.

The fragment in Listing 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";
```

Predicates

Predicates are C# objects of type `Predicate<T₁ … Tₙ>`, where n is the number of arguments to the predicate and T_i is the type of the ith argument. The predicates we've discussed so far are **table predicates**. They are stored as tables (arrays); queries to them are implemented as searches on the table and its indices. They are created using the `Predicate` method:

```
Predicate(name, arg₁, arg₂, …, argₙ)
```

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). The declaration:

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

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

In addition to table predicates, there are primitive predicates directly implemented as C# methods, such as the `<` operator, used in Listing 2. Finally, TED allows predicates defined by rules that are inlined into their calls:

```
var CellAt = Definition("CellAt", loc)
                .Is(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]`. Definitions provide a way of defining predicates that are executed top-down. Their full extensions are not computed.

Goals and Rules

Following logic programming convention, applications of predicates to arguments are referred to as *goals*. Rules combine a conclusion goal, known as the rule's *head*, with a set of conditions, its *body*. The statement:

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

defines a rule stating that $\forall x. Q(x) \rightarrow P(x)$, and adds it to the list of P's rules. For convenience, rules may be combined with predicate definitions, as in:

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

Higher-Order Predicates

Higher-order predicates are predicates parameterized by goals or by other predicates. TED includes a number of these, as well as facilities for defining one's own. Built-in higher-order primitives include:

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

Table Operators

Table operators map tables to tables. They encapsulate algorithms that execute over a table's contents, returning the contents of a new table as a result.

Many operators encapsulate graph algorithms. For example, the `Closure` operator computes transitive closure. If `Edge` is a table of edges in an undirected graph, then:

```
var Connected=Closure("Connected", Edge);
```

defines a new table, `Connected`, that is true when its arguments are connected in the original graph.

Several operators implement graph matching. If `Interest` is a `Predicate<Person,Person>` describing who is interested in dating whom, then:

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

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

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

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

If `Interest` is a relation not between people, but between people and jobs (type `Predicate<Person,Job,float>`), and if `Cap` specifies how many openings there are for each Job (`Predicate<Job,int>`), then

```
var E = AssignGreedily("A",Interest,Cap);
```

matches people to jobs with the highest possible interest level, while honoring the limits on numbers of openings.

Table Update

Base tables can be updated by providing tables of changes to perform. If *`B`* is a base table, 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`*.

Individual columns can be changed by providing tables of changes to make. *`B`*`.Set(`*`key, updateColumn`*`)` 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, performing the specified updates. 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 `Dead`.

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 with *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, with no run-time penalty when disabled. Unlike most languages, however, TED problem rules can be enabled and disabled at run-time without recompilation.

Implementation

TED is highly optimized. Queries use minimal pointer dereferencing and generally run without run-time type checks or storage allocation.

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 stored in-line in the array, rather than as separate heap objects. Table data is thus a contiguous sequence of bytes. Tables are almost always scanned in order, so table operations have excellent cache locality.

Indices are stored as custom hash tables mapping keys to row numbers. Indices use direct addressing with load factors below 0.5. This provides good cache performance while avoiding clustering, at a cost of 8 bytes per row per index, for key indices or 12 for general indices.

Rule Representation

Rules are stored as a sequence of iterators, one per body goal. Rules are transformed into their iterators by a preprocessor, beginning with a normalization pass that performs partial evaluation and algebraic simplification.

Unification

Goals, such as `P[x, y, 7]`, are matched against tuples in tables using unification (Russell and Norvig 2009), which computes the solution to a set of simultaneous equations over terms. Since variables can be unified with other variables, the classic unification algorithm (Robinson 1971) uses forwarding pointers to build an equivalence relation over variables. Finding the variable's value requires a loop to dereference forwarding pointers until a ground value is found. In bottom-up LP, goals are only unified with (variable-free) table tuples, making forwarding unnecessary.

For example, `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`. The equivalent C# code is:

```
x = a; if (x != b || c != 1) goto fail;
```

Running this against a particular `a`, `b`, `c`, tests if they're unifiable, while updating `x` to its new value, but is dramatically faster than the general unification algorithm.

The first (leftmost) use of a variable within a rule sets the variable and so is known as *write mode*. All other references match against the stored value and are said to be *read mode*.

Iterator Selection

After normalization, body goals are mapped to iterators. Primitive predicates have their own custom iterators. For calls to table predicates, the preprocessor searches for a read-mode argument for which the table has an index, choosing the most desirable such index:

- 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.
- A **key index** is the next most desirable. Its "iterator" finds the single row (if any) with the key and unifies against it. It does not actually iterate, since there can be at most one row that matches it. Key indices on multiple columns are preferred over single-column indices
- If only a **non-key index** is available, the iterator iterates over the linked list of rows with the specified column value. Again, multi-column indices are preferred when available.
- If **no index** is available for a read-mode argument, the iterator matches against all rows of the table.

Iterator Sequencing

The iterators effectively form a state machine. Each iterator corresponds to two starts: a start state, and a restart state. The start state finds the first solution, the restart state, subsequent ones. Upon success, control passes to the start state of the following iterator. Upon failure, it passes to the restart state of the previous iterator.

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 jumps to the restart state of the final iterator to generate the next solution.

When the first iterator's restart state fails, the rule has generated all its solutions and execution proceeds to the next rule.

Serial Update

For single-core execution, each derived predicate's table is cleared, then its contents are regenerated by executing its rules. Update honors dependencies between tables: tables are updated after those tables referenced in their rules. Finally, updates are applied to base tables.

Parallel Update

For multicore execution, each predicate is updated in a separate task in the worker thread pool. Tasks are launched when their last dependency completes. No other mutual exclusion is necessary, so the only penalty paid is the general overhead of the .NET task library.

Native Code Compilation

Although the default execution mode is the interpreter discussed above, TED includes an experimental compiler that regenerates C# source code from those iterators. The result is a .cs file that contains compiled state machines for each inferred predicate, to be used the next time the system is run. Although run-time code generation would be preferable, most mobile and console platforms forbid it.

The current compiler inlines all unification and table reads, as well as hash lookups for key indices. As a result, compiled rules consist primarily of array references, equality comparisons, loads, stores, and loops for hash lookups. The only out-of-line calls are to the hash functions, some indices, and table writes. The latter two will be inlined in future versions.

Functionality	TED	Native C#	Ratio
Update logic	11	52	4.7
Person data	2	30	15
Location data	2	3	1.5
Total	**15**	**85**	**5.6**
TED variables	7		
TED wrappers	14		
Total w/overhead	**36**	**85**	**2.3**

Table 1: Code length, excluding blank lines, comments, and code shared between the versions.

Implementation	Avg. update (ms)	% C#
C#	6.01	100
TED Compiled	7.55	125
TED Interpreted	17.47	290

Table 2: Single-core execution times in milliseconds for 100 locations and 2000 characters for interpreted TED, compiled TED, and native C#.

Evaluation

The core claims of this work are that (1) large scale character simulations can be written more compactly with bottom-up logic programming than traditional languages, with (2) competitive performance, but that (3) this requires careful attention to language implementation. We expect more compact code to be easier to write, debug, visualize, and modify.

Fair direct comparisons are difficult because they require implementing identical functionality in both systems. The larger the system, the more difficult it is to ensure identical functionality. *Voix de la Ville* is an attempt to reimplement the bulk of *Talk of the Town* in TED, but it's effectively impossible to know all the functional differences between the systems.

As a more controlled experiment, we implemented three core features found in *Talk of the Town* and other "townlikes". Each was implemented in C# and TED, built on a common core of C# code. This code implements (1) characters selecting locations to go to, (2) characters interact with each other at those locations, and (3) updating inter-character affinities based on the interactions. Each maximizes an objective function based on affinities with the character's personality. Code for generating and comparing personalities is in the common C# core. Annotated source code for the TED version, and repos for both versions are given in the appendix.

Compactness

Table 1 shows the sizes of the two implementations. By line count, TED's update logic is smaller by a factor of 5. However, this does come with some caveats. First, TED lines tend to be denser than C# lines. Second, this excludes one-time costs for the TED code: variable definitions, and wrappers required to make code written for the C# implementation callable by the TED implementation. A relatively small amount of such overhead code gets shared by the whole of the TED project, growing only modestly with the size of the project. Because of the shortness of this benchmark, the overhead dominates the "real" code. However, even counting this boilerplate, TED is less than half the size of native C#.

Performance

Table 2 compares execution times for each implementation in a large world (2000 characters and 100 locations). Tests were performed on an Apple M3 Pro running at 4.06 GHz with 18GB of RAM using standalone builds of each system under Unity version 2021.3.11f1. Each system was run for 1000 ticks to warm the caches, then reset and immediately run again for 1000 ticks and timed. Interpreted TED was roughly 2.9 times slower than compiled native C#, while compiled TED was 25% slower than native C#. This is a single-core comparison. Large simulations with many tables will likely benefit from multicore execution.

Although exact byte counts for particular data structures are unobtainable and can vary between C# implementations, the asymptotic space complexities of the two implementations are identical. We would expect memory footprints to be within constant factors. TED may be slightly smaller because it packs the data more tightly into a smaller number of objects.

Comparison to Other Languages

It would arguably have been better to compare to Python, since that is most commonly used for Townlikes in the academic research community (J. Ryan 2018; Johnson-Bey, Nelson, and Mateas 2022). However, this would considerably complicate the comparison, since we could not build on a shared code base. Since C# is both preferred for shipping games and has a higher bar for performance, we chose to compare to it.

That said, Python performance is variously listed as between 10 and 100 times slower than C#. Good benchmarks are difficult to find but Benchmarksgame puts different Python implementations between 25 and 71 times slower than C# (Benchmarksgame 2024). So even interpreted TED is unlikely to incur a performance penalty compared to Python.

It has also been suggested that rather than C#, we compare to other logic programming languages such as Prolog, Answer-Set Programming, or non-TED Datalog variants.

This is somewhat fraught for a few reasons. First, none of those languages are designed for mutable data, which is the preferable way of implementing simulations. In the case of ASP and pure Datalog, the standard approach would be to add a time argument to every predicate and then write the update rules to compute the data for time $t+1$ based on the data for time t. This would have the effect of archivally storing the complete game state for every tick of the game, with memory consumption growing linearly with the length of the game.

Although implementing the full benchmark in ASP is impractical, we ran the one-line ASP program that declares "one person's affinity for another has a unique value for each timepoint." For 100 people, 100 ticks, and 10 quanta of affinity (standard ASP doesn't handle floating-point), the Clingo ASP solver required 151 seconds to ground, 3 seconds to solve, and 10GB of RAM. The parameters used in our benchmark tests would result in a grounded form that was 4000 times larger and so un-runnable on our machines.

Alternatively, one could transfer the complete game state from the game engine to the ASP/Datalog program, have it compute a new state, and then export it back to the game. However, this would mean serializing and deserializing the complete game state twice per tick, which would also be impractical.

Prolog does allow mutable state, although it strongly discourages it. However, this brings up the issue that logic programming languages don't generally define the time complexities of basic operations.

In our benchmark, both the TED and C# implementations use dynamic arrays and self-expanding hash tables as their low-level data structures. These provide $O(1)$ average, amortized access and update. For P people and L locations, the C# and TED implementations have identical time complexities: $O(P^2/L)$ per tick.

In Prolog, one stores mutable state in the clause database using the assert and retract primitives. The semantics of Prolog specify that when a predicate is called, its clauses are tried exhaustively in order unless the implementation can prove *a priori* that some clauses can't match. So, looking up the affinity between two characters, a $O(1)$ operation in C# and TED, is now $O(P^2)$. That, by itself, raises the time complexity of the simulation to $O(P^4/L)$, although, a naïve implementation be $O(P^5/L)$ because it would run goals P times that TED ran only once.

In principle, Prolog could be implemented to have the same $O(1)$ performance as C# and TED, although we have not been able to find an implementation that is documented to do so. Most Prolog implementations index the clause database to reduce the complexity of lookups. The standard indexing scheme used by most systems indexes on the functors (types) of arguments, which would not be helpful in our benchmark. So-called deep indexing systems can do much better, but these would need to do deep indexing on two arguments at once to attain $O(1)$ performance on our benchmark. SWI Prolog, which as of this writing does deep indexing but only on single arguments, was 2-3 orders of magnitude slower than TED on our benchmark.

Conclusion

Simulations consist in large part of loops over large tables of data. Bottom-up logic programming lets us write those loops declaratively. By storing results in tables whose indexing patterns can be controlled independently of the algorithms operating on them, we can optimize declaratively.

The use of tables as a uniform representation has several advantages. It aids debugging by making all intermediate results inspectable and queryable at run-time. A small number of general table visualizers can then be applied to the entire game state. For example, in *Voix de la Ville* any table can be visualized as spreadsheets, graphs, or tilemaps. This means that when debugging, rather than poking around the C# stack or building new custom visualizers, the entire game state is always interactively visualizable in a live game. Uniform use of tables also makes snapshotting, reloading, and logging game state substantially easier.

TED is a high-performance, embedded, parallelizable, bottom-up, logic programming implementation that allows game designers to quickly and conveniently implement large-scale social simulations that map efficiently onto modern, multi-core architectures. In head-to-head comparisons with native C#, TED is 2-5 times more compact with only a 25% speed penalty in the current implementation.

Acknowledgements

We would like to thank the reviewers for their thoughtful comments. We would also like to thank the people of the Chicago Game Lab for their helpful suggestions, and Rob Zubek and Jason Liu for their work with early versions of the system.

Appendix: Annotated TED Benchmark

The source code used for the benchmarks may be found at github.com/SamuelHill/TEDBenchmarks-{TED,Csharp}. In the code below, a predicate called without arguments is shorthand for calling it with same variables names used in its definition. Thus, within an `If()` rule, `Person` without brackets, is syntactic sugar for `Person[person]`.

The first two lines define predicates of one argument, `Person` and `Location`:

```
var Person = Predicate("Person", person);
var Location = Predicate("Location",
                         location);
```

These are true when the argument is a person (character) or location (business), respectively. The are implemented internally as tables (arrays) of all persons, and locations in the game, respectively. Calls to them test whether the argument is a person/location, or allow enumeration of person/characters in the game, depending on context. For our tests, the Person and Location tables are preloaded at the start of the simulation, both in the TED and C# versions.

Next, we define a predicate that is true when a given person works at a given location. Again, this is implemented as table of (person, location) pairs, and is preloaded with random data before the simulation:

```
var WorkLocation = Predicate
      ("WorkLocation", person, location);
```

Finally, we define a predicate (table) of affinities (degrees of liking) between people. This table is initially empty, and filled in as characters meet:

```
_affinity = Predicate("Affinity", person,
                      other, affinity)
           .JointKey(person, other);
_affinity.Overwrite = true;
```

Next we state that the affinity between two people is the value listed in the `_affinity` table, if any, otherwise the result generated by the C# function `PersonPersonAffinity`, which is shared with the C# implementation:

```
var Affinity = Definition("Affinity",
               person, other,affinity)
    .Is(FirstOf[_affinity,
        PersonPersonAffinity[person, other,
                             affinity]]);
```

Now a table of character locations. It is recomputed each tic by two rules: if it's daytime, they're at work, if it's nighttime, they're at the place that best matches their current mood:

```
_whereTheyAre = Predicate("WhereTheyAre",
   person, location.Indexed)
   .If(IsAM[false], Person,
       RandomMood[mood],
       Maximal(location, affinity,
               Location
               & PersonLocationAffinity
                 [person, mood, location,
                  affinity]))
   .If(IsAM[true], WorkLocation);
```

Now, a predicate stating that a given person interacted with a given other person, that their affinities for each other are the specified affinities, and the outcome of the interaction. It is computed by the rule stating that a person interacts with the other person in their current location with whom they have the highest affinity. `Interact` is another C# function shared with the C# benchmark:

```
_interactedWith=Predicate("InteractedWith",
                person, other, affinity,
                otherAffinity, outcome)
 .If(_whereTheyAre,
     Maximal(other, affinity,
             _whereTheyAre[other, location]
              & Affinity[person, other,
                affinity]),
     Affinity[other, person,otherAffinity],
     Interact[person, other, affinity,
              otherAffinity, outcome]);
```

Finally, the rules for updating the table of inter-character affinities. They state that if `person` interacts with `other` with a resulting `outcome`, their affinities for each other are incremented by the (signed) delta of that outcome for each of them:

```
_affinity.Add
         [person, other,
          affinity + ActorOutcome[outcome]]
  .If(_interactedWith);
_affinity.Add
         [person, other,
          affinity + OtherOutcome[outcome]]
  .If(_interactedWith[other, person, __,
                      affinity, outcome]);
```

References

Adams, Tarn, and Zach Adams. 2006. "Slaves to Armok: God of Blood Chapter II: Dwarf Fortress." Bay 12 Games.

Alvarez-Picallo, Mario, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. 2019. "Fixing Incremental Computation." In *Programming Languages and Systems*, edited by Luís Caires, 525–52. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-17184-1_19.

Benchmarksgame. 2024. "The Computer Language 24.06 Benchmarks Game." June 2024. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html. Accessed: 7-15-2024

Ceri, S., G. Gottlob, and L. Tanca. 1989. "What You Always Wanted to Know about Datalog (and Never Dared to Ask)." *IEEE*

Transactions on Knowledge and Data Engineering 1 (1): 146–66. https://doi.org/10.1109/69.43410.

Clocksin, William F, and Christopher S Mellish. 2003. *Programming in Prolog: Using the ISO Standard*. New York, NY: Springer.

Evans, Richard. 2009. "AI Challenges in Sims 3." In *Artificial Intelligence and Interactive Digital Entertainment*. Stanford, CA: AAAI Press.

Evans, Richard. 2010. "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*, 179–95. Fiesole, Italy: Springer.

Evans, Richard, and Emily Short. 2013. "Versu." San Francisco, CA: Linden Lab.

Evans, Richard, and Emily Short. 2014. "Versu - A Simulationist Storytelling System." *IEEE Transactions on Computational Intelligence and AI in Games* 6 (2): 113–30.

Gardner, Martin. 1970. "Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game 'Life.'" *Scientific American*, no. 223 (October). https://doi.org/doi:10.1038/scientificamerican1070-120.

Gottlob, Georg, Giorgio Orsi, Andreas Pieris, and Mantas Šimkus. 2012. "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*, edited by Thomas Eiter and Thomas Krennwallner, 54–77. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-642-33158-9_2.

Hill, Samuel, and Horswill, Ian. 2023. "An Executable Ontology for Social Simulation." In *Workshop on Experimental Games in AI (EXAG-23)*. Salt Lake City, UT.

Horswill, Ian. 2018. "Postmortem: MKULTRA, An Experimental AI-Based Game." *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 14 (1): 45–51. https://doi.org/10.1609/aiide.v14i1.13027.

Johnson-Bey, Shi, Mark J. Nelson, and Michael Mateas. 2022. "Neighborly: A Sandbox for Simulation-Based Emergent Narrative." In *2022 IEEE Conference on Games (CoG)*, 425–32. Beijing, China: IEEE. https://doi.org/10.1109/CoG51982.2022.9893631.

Kreminski, Max, Melanie Dickinson, and Noah Wardrip-Fruin. 2019. "Felt: A Simple Story Sifter." In , edited by Rogelio E. Cardona-Rivera, Anne Sullivan, and R. Michael Young, 11869:267–81. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-030-33894-7_27.

Lapeyrade, Sylvain. 2022. "Reasoning with Ontologies for Non-Player Character's Decision-Making in Games." *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* 18 (1): 303–6. https://doi.org/10.1609/aiide.v18i1.21980.

Martin, Adam. 2013. "Entity Systems Are the Future of MMOG Development – Part 1 – T-Machine.Org." July 31, 2013. https://new.t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/. Accessed: 7-15-2024.

Mason, Stacy, Ceri Stagg, Noah Wardrip-fruin, and Michael Mateas. 2019. "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.

Mateas, Michael, and Andrew Stern. 2002. "A Behavior Language for Story-Based Agents." *IEEE Intelligent Systems* 17 (4): 39–47.

Mateas, Michael, and Andrew Stern. 2005. "Façade."

Maxis. 2009. "The Sims 3." Redwood City, CA: Electronic Arts.

McCoy, Josh, Mike Treanor, Ben Samuel, and Aaron A. Reed. 2012. "Prom Week." Santa Cruz, California: Expressive Intelligence Studio at UC Santa Cruz.

McCoy, Joshua, Mike Treanor, Ben Samuel, Noah Wardrip-Fruin, and Michael Mateas. 2011. "Comme Il Faut: A System for Authoring Playable Social Models." In *Proceedings of the 7th AI and Interactive Digital Entertainment*, edited by Vadim Bulitko and Mark O. Riedl. Stanford, CA: AAAI Press.

Nelson, Graham. 2006a. "Inform 7."

Nelson, Graham. 2006b. "Natural Language, Semantic Analysis, and Interactive Fiction." Cambridge, UK: Unpublished white paper.

Pereira, Fernando C. N., and Stuart Shieber. 1987. *Prolog and Natural Language Analysis*. Brookline, MA: Microtome Publishing.

Pereira, Fernando C. N., and David H. D. Warren. 1980. "Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks." *Artificial Intelligence* 13 (231–278).

Robinson, John Alan. 1971. "Computational Logic: The Unification Computation." In *Machine Intelligence 6*, 63–72. Edinburgh University Press.

Russell, Stuart, and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Ryan, James. 2018. "Curating Simulated Storyworlds." University of California Santa Cruz.

Ryan, James Owen, Ben Samuel, and Adam Summerville. 2016. "Bad News : A Game Of Death And Communication," 160–63.

Samuel, Ben, Aaron A Reed, Paul Maddaloni, Michael Mateas, and Noah Wardrip-Fruin. n.d. "The Ensemble Engine: Next-Generation Social Physics."

Samuel, Ben, James Ryan, Adam J. Summerville, Michael Mateas, and Noah Wardrip Fruin. 2016. "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)*. Vol. 10045 LNCS. https://doi.org/10.1007/978-3-319-48279-8_10.

Samuel, Ben Selvan. 2021. "Kismet: A Small Social Simulation Language," Joint Proceedings of the ICCC 2020 Workshops (ICCC-WS 2020), September 7-11 2020, Coimbra (PT).

SomaSim. 2021. "City of Gangsters." Chicago, IL.

Sylvester, Tynan. 2018. "RimWorld." Ludeon Studios.

Ullman, Jeffrey D. 1988. *Ullman: Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press.

Ullman, Jeffrey D. 1989. *Ullman: Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies*. Computer Science Press.

Van Emden, M. H., and R. a. Kowalski. 1976. "The Semantics of Predicate Logic as a Programming Language." *Journal of the ACM* 23 (4): 733–42. https://doi.org/10.1145/321978.321991.

Viglione, Matt. 2025. “Rise of Industry 2.” SomaSim, LLC. CHicago, IL.

Warren, D H D, L M Pereira, and F Pereira. 1977. “PROLOG - The Language and Its Implementation Compared with LISP.” In *Symposium on AI and Programming Languages*, 12:109–15. ACM. https://doi.org/10.1145/800228.806939.

Wright, Will. 2000. “The Sims.” MAXIS/Electronic Arts. Redwood City, CA

Zubek, Robert. 2010. “Needs-Based AI.” In *Game Programming Gems 8*. 8. Cengage Learning PTR.