EQ677 Model Searcher - User Manual

Overview

The eq677 program is a specialized model searcher designed to find finite algebraic structures (magmas) that satisfy equation 677 from the Equational Theories project. This tool was developed to investigate the remaining open implication 677 → 255 in the study of equational theories.

The program implements multiple sophisticated search strategies, including constraint-based DPLL solvers, specialized algebraic searches, and database enumeration methods. It is written in Rust and optimized for performance with parallel execution capabilities.

Mathematical Background

Equation Definitions

The program searches for models of the following equations:

Equation 677: For all elements x and y in the magma, the following identity must hold:

```
x = y * (x * ((y * x) * y))
```

More formally, denoting the binary operation as f, this states:

```
x = f(y, f(x, f(f(y, x), y)))
```

Equation 255: This equation states that every element is idempotent with respect to a specific 4-fold composition:

```
x = f(f(f(x, x), x), x)
```

The program can search for models that satisfy equation 677, and it can optionally filter to find only those models that do not satisfy equation 255 (the anti-255 condition), which is relevant to the open mathematical question of whether 677 implies 255.

Model Representation

Models (magmas) are represented as multiplication tables stored in matrix form. For a magma of size n, the multiplication table is an n×n matrix where entry (x, y) contains the result of f(x, y). The program uses canonical forms to avoid discovering duplicate models that differ only by relabeling of elements.

Building and Compilation

The program requires the Rust nightly compiler due to its use of experimental features, particularly

explicit tail calls. To build the program:

```
bash

cargo +nightly build --release
```

The release build includes debug symbols for profiling purposes, as specified in the Cargo.toml configuration.

Dependencies

The program relies on several key dependencies including egg (for equality saturation), nauty-pet (for graph automorphisms and canonicalization), z3 (for SMT solving in some search methods), rayon (for parallel execution), and various utility libraries for collections and randomization.

Program Architecture and Entry Points

Main Function Modification Required

The program does not currently implement a traditional command-line interface. Instead, users must modify the main function in src/main.rs to specify which search operations to execute. The default main function contains a sample call:

```
rust

fn main() {
    setup_panic_hook();
    let _timer = Timer::new();

    M(89, 0).get().cycle_dump();
}
```

To perform searches, you would replace this with calls to the various search functions described below.

Search Function Overview

The program provides numerous search strategies, each accessible through dedicated functions. These can be called individually or in combination. The primary search entry point that runs all search methods in parallel is:

```
rust

search::all()
```

This function spawns parallel threads for all available search methods, including linear_search, linmat_search, affine_search, affmat_search, poly_search, bij_plus_search, bij_mul_search, c_search,

semitinv_search, tinv_search, db_search, db_cart_search, complex_linear_search, and complex_affine_search.

Search Methods

The program implements multiple specialized search strategies, each targeting different structural properties that models satisfying equation 677 might possess.

Linear and Affine Searches

Linear Search: This method searches for models where the binary operation has the form f(x, y) = (ax + by) mod p for some prime p and coefficients a and b. The constraints imposed by equation 677 reduce to algebraic conditions on these coefficients. The search systematically explores all primes and coefficient combinations.

Affine Search: Similar to linear search but with an additional constant term, searching for operations of the form f(x, y) = (ax + by + c) mod p.

Linear Matrix Search (linmat_search): Searches for models where the operation can be expressed using matrix operations in modular arithmetic, providing a more general framework than simple linear forms.

Affine Matrix Search (affmat_search): Extends the matrix approach to include affine transformations.

Complex Number Searches

Complex Linear Search: Explores models defined over complex arithmetic, searching for operations that can be expressed as linear combinations in the complex plane.

Complex Affine Search: Extends complex linear search to include affine transformations in complex arithmetic.

Translation-Invariant Searches

T-Invariant Search (tinv_search): This is one of the most sophisticated search methods. It searches for models where the operation has the special form f(x, y) = x + h(y - x), where h is a permutation function. This structure enforces translation invariance. The search uses a constraint-based DPLL (Davis-Putnam-Logemann-Loveland) solver to efficiently explore the space of possible h functions.

Parameters can be controlled through constants in src/tinv_dpll/run.rs:

- **ANTI_255**: When set to true, filters out models that satisfy equation 255, searching specifically for counterexamples to the 677 → 255 implication.
- **SELF_INVERSE**: When enabled, restricts the search to models where h is self-inverse (h(h(x)) = x).
- **H0_0_OR_1**: Forces h(0) to be either 0 or 1, reducing the search space through symmetry.

The search also supports parallel execution with configurable threading depth controlled by the threading_depth function.

Semi T-Invariant Search (semitinv_search): A variation that relaxes some of the t-invariant constraints while maintaining related structural properties.

Bijection-Based Searches

Bij Plus Search (bij_plus_search): Searches for models where each left multiplication map $x \mapsto f(a, x)$ is a bijection, and these bijections can be expressed using addition operations.

Bij Mul Search (bij_mul_search): Similar to bij_plus_search but using multiplication-based bijections.

Polynomial and Specialized Searches

Polynomial Search (poly_search): Explores models where the operation can be expressed as polynomial functions over finite fields.

Fake Linear Search (fakelin): Searches for models that appear linear in certain respects but may violate full linearity.

DivT-Invariant Search (divtinv): A specialized search for models with divisibility-based translation invariance properties.

Database Searches

Database Search (db_search): Iterates through all models stored in the embedded database (located in the db/ directory), verifying that they satisfy equation 677 and presenting them again. This is useful for verification and analysis of known models.

Database Cartesian Search (db_cart_search): Generates new models by taking Cartesian products of models from the database. For models m1 and m2, the Cartesian product $m1 \times m2$ is a model of size $|m1| \times |m2|$. The search automatically filters products that would be too large (exceeding 1000 elements) and skips trivial cases.

Equality-Based DPLL Searches

EQ-DPLL Search: Uses an equality-based DPLL solver that reasons about equalities between terms rather than assignments of values. This can be more efficient for certain classes of problems. The search builds a term DAG representing equation 677 constraints and uses congruence closure for propagation.

Symmetry DPLL (sym_dpll): A DPLL-based search that explicitly accounts for symmetries in the problem, using graph automorphism detection to prune symmetric branches.

C-DPLL Search (c_search): A specialized DPLL variant with custom constraint propagation rules optimized for the structure of equation 677.

Extended and Composite Searches

Extend Search: Given partial models or smaller models, this search attempts to extend them to larger

sizes while maintaining the equation 677 property.

Composite Analysis: The program includes functionality for decomposing models into irreducible components and analyzing whether models can be expressed as combinations of simpler structures.

Size Parameters and Search Ranges

Most search functions iterate indefinitely through increasing sizes, starting from size 0 or 1. The typical pattern is:

```rust
pub fn search_name() {
    for n in 0.. {
        // search for models of size n
    }
}
```

To limit searches to specific sizes, users must modify the relevant search function. For example, to search only for models of size 10 to 20 with the t-invariant search:

```rust
pub fn tinv_limited() {
    for n in 10..=20 {
        tinv_run(n);
    }
}
```

There is no built-in command-line mechanism for specifying size ranges; this must be done by editing the source code.

Termination Conditions

The program does not implement automatic termination conditions by default. Search functions will run indefinitely unless manually terminated or until they exhaust their search space (which for most methods is infinite).

To implement timeout functionality, users would need to add timer checks. The program includes a Timer type in src/timer.rs that measures elapsed time. Custom termination logic could be added like:

```rust
```

```rust
fn limited_search_with_timeout(max_seconds: u64) {
    let start = std::time::Instant::now();
    for n in 0.. {
        if start.elapsed().as_secs() > max_seconds {
            break;
        }
        tinv_run(n);
    }
}
```

Similarly, to stop after finding a specific number of models, one would need to maintain a counter and check it during execution.

Model Database System

The program includes an embedded database of previously discovered models. This database is stored in the db/ directory, with subdirectories organized by model size (e.g., db/25/ contains models of size 25). Each file contains the multiplication table of one model in a parseable text format.

Database Structure

Models are referenced using the notation M(n, k), where n is the size of the model and k is an index distinguishing different models of that size. For example, M(25, 5) refers to the sixth model of size 25 in the database (using zero-based indexing).

The database is compiled into the binary during build time through the build.rs script, which reads all model files and generates Rust code to include them. This approach ensures fast access to known models without requiring file I/O during runtime.

Database Operations

Models can be retrieved from the database programmatically:

```rust
let model = M(25, 0).get();  // Get first model of size 25
model.dump();                // Display the multiplication table
```

The db() function returns all models as a vector of (name, magma) pairs:

```rust
for (name, magma) in db() {
    println!("Model {}: size {}", name, magma.n);
}
```

Database models are guaranteed to satisfy equation 677, and the current database contains only models that also satisfy equation 255 (all tests verify this). The models are stored in canonical form to avoid duplicates.

Output Format and Model Presentation

When a new model is discovered, the program outputs information in several formats.

Standard Output

The present_model function generates output following this pattern:

```
[New ]Model "M(n, k)" [non-right-cancellative] [non-idempotent] found by search_method:
```

The "New" prefix appears only if this model was not previously in the database. Additional annotations indicate special properties like non-right-cancellativity or non-idempotence.

Multiplication Table Display

For models of size 100 or smaller, the full multiplication table is displayed in matrix form:

```
0 1 2 3
1 2 3 0
2 3 0 1
3 0 1 2
```

Each row x shows the results of f(x, 0), f(x, 1), ..., f(x, n-1).

Cycle Notation Display

The program can also display models in cycle notation using the cycle_dump method. For each row x, it shows the permutation defined by y ↦ f(x, y) in standard cycle notation. For example:

```
0: (0)(1 2 3)
1: (0 1 2 3)
2: (0 2)(1 3)
3: (0 3 2 1)
```

This format is particularly useful for understanding right-cancellative models where each row defines a bijection.

Column Cycle Notation

For right-cancellative models, column_cycle_dump displays the column permutations, showing the bijection x ↦ f(x, y) for each y.

Symmetry and Canonicalization

The program implements sophisticated canonicalization to detect when two multiplication tables represent the same model up to relabeling of elements. This uses the nauty-pet library for graph automorphism detection.

The canonicalize2() method transforms any model into its canonical form. Two models are considered equivalent if their canonical forms are identical. The database stores only canonical forms.

Automorphism Analysis

For models of moderate size (currently less than 25 elements when SHOW_AUTOM is enabled in src/present.rs), the program can display the automorphism group:

```
rust

magma.autom_dump();
```

This shows the symmetries of the model, which can provide insights into its algebraic structure.

Performance Optimization Options

The program includes several mechanisms for optimizing search performance.

Parallel Execution

Most search methods support parallel execution through Rust's rayon library. The into_par_for_each function distributes work across available CPU cores. Threading depth can be controlled through parameters like threading_depth in the DPLL solvers.

For maximum parallel utilization, use the search::all() function, which runs all search strategies simultaneously on separate threads.

Compilation Flags

Always compile with the release profile for serious searches:

```
bash

cargo +nightly build --release
```

The release build enables optimizations and removes debug assertions. However, debug symbols are still included for profiling.

Constraint Pruning

The DPLL-based searches use sophisticated constraint propagation and scoring heuristics to prune the search tree. The scoring functions in files like tinv_dpll/run.rs determine the order in which variables are assigned, which can dramatically affect performance.

Heuristic parameters can be tuned by modifying constants like C11_SCORE, C12_SCORE, C21_SCORE, and C22_SCORE in the constraint definition files.

Symmetry Breaking

Many searches incorporate symmetry-breaking constraints. For example, the H0_0_OR_1 flag in tinv_dpll limits h(0) to reduce redundant exploration of symmetric cases. Such constraints can provide order-of-magnitude speedups.

Additional Features and Interesting Capabilities

Conjectures and Automated Theorem Proving

The conj module implements automated conjecture generation. When a model is discovered, the program can automatically test various algebraic properties and generate conjectures about relationships with other equations. This is implemented through integration with automated theorem provers.

Knuth-Bendix Completion

The kb module provides Knuth-Bendix completion functionality for generating term rewriting systems from equational specifications. This can be used to analyze the equational theory generated by discovered models.

Twee Integration

The program integrates with the Twee theorem prover (twee.rs and twee_sys.rs) for advanced equational reasoning. This can verify properties of discovered models or search for proofs of relationships between equations.

Composite Structure Analysis

When CHECK_COMPOSITE is enabled in src/present.rs, newly discovered models are automatically analyzed to determine whether they decompose into smaller models via direct product constructions. The decompose function identifies all possible factorizations.

Property Testing

The program automatically checks several algebraic properties:

- **Right-cancellativity**: Whether f(x, y) = f(x, z) implies y = z for all x
- **Idempotence**: Whether f(x, x) = x for all x
- **Self-inverse permutations**: For t-invariant models, whether h(h(x)) = x

These properties are reported in the output when models are presented.

One-Orbit Search

The one_orbit module implements specialized searches for models where certain group actions have a

single orbit, indicating high degrees of symmetry.

Extended Equation Analysis

The analysis module provides tools for analyzing relationships between equation 677 and various other equations in the equational theories database. This includes automated verification of implications and counterexample searches.

Programmatic Usage Examples

Since the program lacks a traditional CLI, here are examples of how to modify main.rs for common tasks:

Example 1: Search for models of size exactly 30

```rust
fn main() {
    setup_panic_hook();
    let _timer = Timer::new();

    tinv_run(30);
}
```

Example 2: Search for non-255 models using t-invariant with timeout

```rust
fn main() {
    setup_panic_hook();
    let _timer = Timer::new();

    // Modify ANTI_255 to true in src/tinv_dpll/run.rs first
    let start = std::time::Instant::now();
    let timeout_secs = 3600; // 1 hour

    for n in 10..100 {
        if start.elapsed().as_secs() > timeout_secs {
            println!("Timeout reached");
            break;
        }
        tinv_run(n);
    }
}
```

Example 3: Run all search methods in parallel

```rust
fn main() {
    setup_panic_hook();
    let _timer = Timer::new();

    search::all();
}
```

Example 4: Analyze all database models

```rust
fn main() {
    setup_panic_hook();
    let _timer = Timer::new();

    for (name, magma) in db() {
        println!("=== {} ===", name);
        magma.cycle_dump();
        println!("Right-cancellative: {}", magma.is_right_cancellative());
        println!("Idempotent: {}", magma.is_idempotent());
        println!();
    }
}
```

Example 5: Search with specific symmetry constraints

```rust
fn main() {
    setup_panic_hook();
    let _timer = Timer::new();

    // First modify H0_0_OR_1 and SELF_INVERSE in tinv_dpll/run.rs
    // to enable desired constraints, then:

    for n in 5..50 {
        tinv_run(n);
    }
}
```

Advanced Configuration

Most advanced configuration requires modifying constants and flags in the source code. Key configuration files include:

- **src/tinv_dpll/run.rs**: ANTI_255, SELF_INVERSE, H0_0_OR_1, threading_depth
- **src/semitinv_dpll/run.rs**: Similar flags for semi-t-invariant search
- **src/c_dpll/run.rs**: Configuration for the c-dpll search variant
- **src/present.rs**: CHECK_COMPOSITE, SHOW_AUTOM, and output formatting
- **src/matrix/mod.rs**: FIXED_WIDTH for table formatting

After modifying these files, recompile the program for changes to take effect.

Troubleshooting and Limitations

Memory Consumption

Large searches, particularly those using DPLL solvers, can consume substantial memory. The constraint graphs for models of size 100+ may require several gigabytes of RAM. Monitor memory usage and adjust size ranges accordingly.

Infinite Searches

Most search functions iterate indefinitely. Always plan for manual termination or implement custom stopping conditions. The program does not include built-in resource limits.

Compilation Requirements

The program requires Rust nightly due to explicit tail call features. Ensure you're using a recent nightly build. Standard Rust stable will not compile this code.

Z3 Dependency

Some search methods rely on the Z3 SMT solver. Ensure Z3 is properly installed on your system, as the z3 Rust crate requires Z3 libraries to be available.

Database Modifications

If you add models to the db/ directory, you must rebuild the entire program (not just recompile) to regenerate the db_sources.rs file. A simple `cargo clean` followed by `cargo build --release` ensures the build script runs.

Summary

The eq677 model searcher is a sophisticated mathematical research tool implementing numerous specialized search strategies for finding algebraic structures satisfying equation 677. While it lacks traditional command-line conveniences, it provides powerful and flexible programmatic access to cutting-

edge search algorithms. The program excels at parallel exploration of large search spaces and includes extensive support for symmetry exploitation, constraint propagation, and automated analysis of discovered models. Users should be prepared to modify source code for specific research needs and should carefully manage computational resources when conducting large-scale searches.