Formalizing the Connectivity Theorem for the Matrix Multiplication Flip Graph in ACL2

Quan Luu Zeke Medley Panagiotis Manolios

Northeastern University

ACL2 Workshop 2026

Fast matrix multiplication

- How many scalar multiplications does it take to multiply two $n \times n$ matrices?
  - Textbook: n^3. Strassen (1969): 7 instead of 8 for 2×2.
- A **scheme** is a way to compute the product; its **rank** is its number of multiplications.
- Recent records for small formats over $\mathbb{F}_2$:

format	4×4	5×5	6×6	7×7
best known rank	47	93	153	245
textbook	64	125	216	343

- Most of these were found by searching the **flip graph** (Kauers & Moosbauer, 2023).

The flip graph, and why connectivity matters

- **Vertices**: schemes.
- **Edges**: local rewrites that change a scheme but not what it computes.
- **Search**: start from a known scheme, random walk
- **Key property**: connectivity
  - theoretically, can start from any known schemes, e.g. the textbook one, and
  - can reach any schemes, even unknown ones.

Kauers & Moosbauer (2023)	flip + reduction	weakly connected
Arai et al. (2024)	flip + plus + reduction	strongly connected; proof has a gap
Medley et al. (2027)	**flip + plus**	**strongly connected**

This paper

A complete ACL2 proof of the last result: 29 books, 215 definitions, 791 theorems.

Schemes, flips and pluses

- A **rank-one tensor**: $A \otimes B \otimes C$, with A, B, C $n \times n$ matrices over $\mathbb{F}_2$.
- The **matrix multiplication tensor** for $n \times n$, with E_{ij} the unit matrix (a single 1 at row i, column j):

$$\mathcal{M}^{(n)} = \sum_{i,j,k=1}^{n} E_{ij} \otimes E_{jk} \otimes E_{ik}$$

- A **scheme**: a multiset of nonzero rank-one tensors whose sum is $\mathcal{M}^{(n)}$.
- Size of a scheme = number of scalar multiplication

Two kinds of edges, shown at position A (B and C are analogous):

flip: $A \otimes B \otimes C,\ A \otimes B' \otimes C' \ \rightarrow\ A \otimes B \otimes (C - C'),\ A \otimes (B' + B) \otimes C'$

plus: $A \otimes B \otimes C,\ A' \otimes B' \otimes C' \ \rightarrow\ A \otimes B \otimes C,\ (A' - A) \otimes B' \otimes C',\ A \otimes B' \otimes C'$

Zero summands are deleted. Both moves keep the sum.

Property of $\mathbb{F}_2$

$X + X = 0 \Rightarrow$ a flip of a summand with itself deletes both.

Lemma: spanning

Lemma (Spanning)

Let S be a scheme for $\mathcal{M}^{(n)}$. The A-components of the summands of S span the full space of $n \times n$ matrices. Likewise for B and C.

What it means for us.

- **Any** matrix X is a sum of A-components that are **already in the scheme**:

$$X = A(u_1) + A(u_2) + \cdots + A(u_k), \qquad u_i \in S.$$

- So whatever A-component we want to build, the raw material is there.

Why it holds. $\mathcal{M}^{(n)} = \sum_{i,j,k} E_{ij} \otimes E_{jk} \otimes E_{ik}$ contains every unit matrix E_{ij} in the A position; a scheme that sums to $\mathcal{M}^{(n)}$ must be able to produce each of them.

Lemma: add duplicate & add arbitrary

Lemma (Add duplicate)

If $A \otimes B \otimes C$ and $A' \otimes B' \otimes C'$ are in S, and $X = A + A'$, there is a path from

$$S \to S \uplus \{X \otimes B \otimes C,\ X \otimes B \otimes C\}.$$

Lemma (Add arbitrary)

For $n \geq 2$ and any nonzero X, Y, Z, there is a path from S to $S \uplus \{X \otimes Y \otimes Z,\ X \otimes Y \otimes Z\}$.

Proof sketch: spanning + add duplicate, iterated.

- Carry one pair and move its A-component toward X, one existing A-component at a time:

$$A_0 \ \to\ A_0 + A(u_1) \ \to\ A_0 + A(u_1) + A(u_2) \ \to\ \ldots \ \to\ X$$

- Each step: duplicate-sum makes the next pair, a self-flip deletes the old one.
- Spanning says such a chain $u_1, \ldots, u_k$ exists.
- Then do the same for B toward Y, and for C toward Z.

Lemma: zero-summing

Lemma (Zero-summing)

For $n \geq 2$, if $R \subseteq S$ and the summands of R sum to zero, there is a path from

$$S \to S \setminus R.$$

Proof sketch.

- Break every summand of R into **unit summands** $E_{ab} \otimes E_{cd} \otimes E_{ef}$ (using the previous lemma and a flip).
- Unit summands are linearly independent: if they sum to zero, each occurs an **even** number of times.
- Now R is made of identical pairs, delete them via flips.

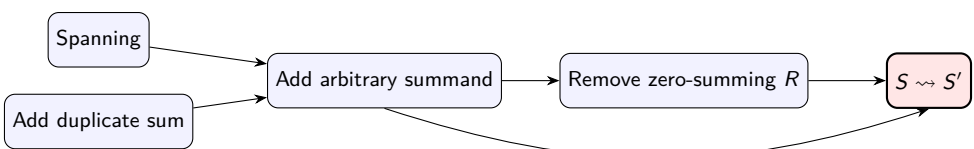
Putting it together

Theorem (Flip-plus connectivity, Medley et al.)

Over $\mathbb{F}_2$, for every $n \geq 2$ and any two schemes S, S', there is a path from S to S' using only flips and pluses.

$$S \xrightarrow{\text{add each } s \in S' \text{ twice}} S \uplus S' \uplus S' \xrightarrow{\text{remove } S \uplus S'} S'$$

- $S \uplus S'$ sums to $\mathcal{M}^{(n)} + \mathcal{M}^{(n)} = 0$, so the removal lemma applies.



Bit lists and matrices: two kinds of predicate

A vector is a list of bits; a matrix is a **list of rows**: `'((1 0) (0 1))`. Addition is `bitxor`.

Structural: says nothing about size.

```
(define bit-list-p (x)
  (if (atom x)
      (null x)
    (and (bitp (car x))
         (bit-list-p (cdr x)))))
```

Matrices: `bit-list-listp`

Type: fixes the dimensions.

```
(define bit-listnp (x (n natp))
  (if (zp n)
      (null x)
    (and (consp x)
         (bitp (car x))
         (bit-listnp (cdr x) (1- n)))))
```

Matrices: `(bit-matp a m n)`

Summands and schemes

```
(define summandp (s (n natp))
  (and (mat-triplep s)          ; (A B C)
       (bit-matp (car s) n n)
       (bit-matp (cadr s) n n)
       (bit-matp (caddr s) n n)
       (not (summand0p s n))))  ; no zero factor
```

```
(define schemep (x (n natp))
  (and (obag::bagp x)
       (summand-listp x n)))
```

- A summand $A \otimes B \otimes C$ is the **triple** `(A B C)`.
- A scheme is an **ordered bag** from `std/obags`: a list sorted by $\ll$, repeats allowed.
  - two schemes are the same multiset iff they are `equal`
- Every insertion goes through `insert-all-nonzero`, which drops a summand with a zero factor.

```
(define insert-all-nonzero ((l mat-triple-listp) (s obag::bagp) (n natp))
  (if (atom l)
      s
    (if (summand0p (car l) n)
        (insert-all-nonzero (cdr l) s n)                    ; drop it
      (obag::insert (car l) (insert-all-nonzero (cdr l) s n)))))
```

Moves: flip and plus

A **move** is data: `(:flip p s1 s2)` or `(:plus p s1 s2)`, with $p \in \{0, 1, 2\}$ the A, B, or C position.
A **path** is a list of moves.
flip and **plus** are functions: two summands and a position in, the two output summands out.

```
(define flip ((s1 mat-triplep) (s2 mat-triplep) (p natp))
  (let ((q (pos-next p))
        (r (pos-prev p)))
    (list (summand-set-pos s1 r (bit-mat-add (summand-get-pos s1 r)
                                             (summand-get-pos s2 r)))
          (summand-set-pos s2 q (bit-mat-add (summand-get-pos s2 q)
                                             (summand-get-pos s1 q))))))

(define plus ((target mat-triplep) (pivot mat-triplep) (p natp))
  (list (summand-set-pos target p (bit-mat-add (summand-get-pos target p)
                                               (summand-get-pos pivot p)))
        (summand-set-pos target p (summand-get-pos pivot p))))
```

- `summand-get-pos` / `summand-set-pos` read and replace the factor at a position; `pos-next` / `pos-prev` step through A, B, C cyclically.

Applying moves and paths

```
(define apply-move ((move move-p) (sch obag::bagp) (n natp))
  (case (move-kind move)
    (:plus (insert-all-nonzero (plus (move-s1 move) (move-s2 move) (move-pos move))
                               (obag::delete (move-s1 move) sch)
                               n))
    (:flip (insert-all-nonzero (flip (move-s1 move) (move-s2 move) (move-pos move))
                               (obag::delete (move-s1 move)
                                             (obag::delete (move-s2 move) sch))
                               n))
    (t sch)))
```

- `apply-move`: delete the summands the move consumes, insert its outputs (zeros dropped).
- `apply-moves`: apply a path, move by move.
- `move-validp`: both summands are in the scheme (twice, if they are the same summand), and for a flip they agree at p.
- `path-validp`: every move is valid on the scheme produced by the moves before it.

Tensors and correct schemes

A tensor over $n \times n$ matrices is a **flat bit list of length** n^6, so the bit-list operations apply to it directly. `(mm-tensor n)` is $\sum_{i,j,k} E_{ij} \otimes E_{jk} \otimes E_{ik}$, by three nested recursions.

```
(define tensor ((a bit-list-listp) (b bit-list-listp) (c bit-list-listp))
  (bit-list-outer (mat-flat a)                          ; A (x) B (x) C
                  (bit-list-outer (mat-flat b) (mat-flat c))))

(define scheme-sum ((l mat-triple-listp) (n natp))      ; sum of the summands' tensors
  (if (atom l)
      (bit-listn0 (* n n n n n n))
    (bit-list-add (summand-tensor (car l))
                  (scheme-sum (cdr l) n))))

(define correct-schemep (sch (n natp))                  ; the paper's notion of scheme
  (and (schemep sch n)
       (equal (scheme-sum sch n) (mm-tensor n))))

(defrule correct-schemep-of-apply-moves                 ; a valid path preserves it
  (implies (and (correct-schemep sch n) (path-validp moves sch n) (natp n))
           (correct-schemep (apply-moves moves sch n) n)))
```

The main theorem

```
(defrule flip-plus-connectivity
  (implies (and (correct-schemep sch n)
                (correct-schemep sch2 n)
                (natp n) (<= 2 n))
           (b* ((path (main-path sch sch2 n))
                (res  (apply-moves path sch n)))
             (and (path-validp path sch n)
                  (equal res sch2)))))
```

- "There is a path" becomes a **function** that returns the path. No existential quantifier.

Schemes are algorithms, and algorithms are schemes

Are bags of matrix triples really **multiplication algorithms**? A small language of **bilinear algorithms**:

- a **linear form** in the entries of one matrix: `(:var i j)`, `(:add e1 e2)`, `(:scale b e)`
- a **product** `(:mul d-form f-form)`: a form in D times a form in F
- an **algorithm**: a list of products, and an $n \times n$ table listing which products are summed into each output entry

```
(:mul (:add (:var 0 0) (:var 1 1)) (:add (:var 0 0) (:var 1 1)))   ; (D00+D11)(F00+F11)
```

Soundness, **completeness**, and **existence** (so the main theorem is not vacuous):

```
(defrule scheme-algo-correct              ; correct scheme -> correct algorithm
  (implies (and (correct-schemep sch n) (natp n) (< 0 n))
           (algo-correctp (scheme-algo sch n) n)))

(defrule algo-to-correct-scheme           ; correct algorithm -> correct scheme
  (implies (and (algop alg n) (natp n) (< 0 n) (algo-correctp alg n))
           (correct-schemep (algo-scheme alg n) n)))

(defrule correct-schemep-of-naive-algo-scheme       ; the textbook algorithm
  (implies (natp n) (correct-schemep (algo-scheme (naive-algo n) n) n)))
```

Who did what

The proof was developed with **Claude** (Anthropic) through Claude Code: mostly Claude Fable 5, with Opus 5 and Fable 5.1 for some sessions.

1. We wrote the base: bit lists, bit matrices and their operations.
2. Claude read the companion paper and proposed the representations of summands, schemes, moves and paths.
3. Along the way, we built up a set of conventions for it to follow.
4. Claude stated and proved the main theorem and every intermediate lemma, within two days.
5. We reviewed the whole proof, and ordered changes where the explanation showed a problem.

Conventions 1: the basics

- **`define` and `defrule`** everywhere
  - 137 `defines` and 527 `defrules` in the connectivity books; zero `defun`/`defthm`
- **Definitions are disabled by default.** Enable one only in the proofs that need it.
  - about 100 of the 137 functions are disabled; a proof never opens a recursive definition by accident
- **Guards verification** for every function.
- **Two predicates**: structural in guards and `:returns`, plus a separate theorem with the type predicate.

```
(define bit-list-add ((x bit-list-p) (y bit-list-p))
  :returns (z bit-list-p)                    ; structural, proved by define
  ...)
(defrule bit-listnp-of-bit-list-add          ; type, what later proofs use
  (implies (and (bit-listnp x n) (bit-listnp y n))
           (bit-listnp (bit-list-add x y) n)))
```

Conventions 2: how to prove things

- **No new induction schemes.** At the start, Claude defined a new induction scheme for almost every proof that needed induction.
- **An invariant theorem at every step.** The idea is to build up a theory around every newly defined function

```
(defrule add-duplicate                    ; the theorem for add-duplicate-path
  (implies (and (schemep sch n)
                (obag::in (summand a1 b1 c1) sch) (obag::in (summand a2 b2 c2) sch))
           (let ((path (add-duplicate-path a1 b1 c1 a2 b2 c2 n))
                 (x (bit-mat-add a1 a2)))
             (and (path-validp path sch n)                          ; valid
                  (equal (apply-moves path sch n)                   ; lands here
                         (insert-all-nonzero (list (summand x b1 c1) (summand x b1 c1))
                                             sch n))))))
```

What the model came up with itself

- **Compute, don't quantify**: every "there exists" becomes a function that returns the witness.
- **Run before you prove**: every function is executable, so it ran each construction at $n = 2$ first.

Spanning in ACL2

```
(defruled a-components-span-everything
  (implies (and (correct-schemep sch n) (natp n) (< 0 n) (bit-matp y n n))
           (in-spanp y (a-components sch) n n)))
```

Different from the paper proof:

- The paper proof showed that there exists a linear combination
- For ACL2, the linear combination is **explicit**: `a-witness-sum` adds the A-factors of the summands with $B[j,k] \cdot C[i,k] = 1$, and the result is E_{ij}.
- For each matrix in the scheme S, we either add it to the spanning set or ignore it.
- `span-witness` returns the coefficients, one bit per matrix.

Adding summands in ACL2

```
(defrule add-arbitrary
  (implies (and (correct-schemep sch n) (natp n) (<= 2 n)
                (bit-matp x n n) (not (equal x (bit-mat0 n n)))
                (bit-matp y n n) (not (equal y (bit-mat0 n n)))
                (bit-matp z n n) (not (equal z (bit-mat0 n n))))
           (let* ((path (add-arbitrary-path sch x y z n))
                  (s    (summand x y z))
                  (sch3 (apply-moves path sch n)))
             (and (path-validp path sch n)
                  (equal sch3 (obag::insert s (obag::insert s sch)))
                  (correct-schemep sch3 n)))))
```

Same statement as the paper. `add-duplicate`: the paper's four moves, proved by symbolic execution.

Different from the paper proof:

- Every partial sum $A_0 + A(u_1) + \ldots$ must be **nonzero**, or the carried summand vanishes.
- We pick the chain from an **independent** pool of A-components.

The experiment

Setup. A fresh session with Claude Fable 5.1.

- Given: the companion paper (Medley et al.) **and nothing else**.
- Task: formalize the paper in ACL2.
- No human feedback, only ACL2's.

Result. 24 certified books, no `skip-proofs`, `defaxiom` or trust tags.

- The **connectivity theorem**, with all four lemmas.
- And the **rest of the paper**, which we did not formalize

It made different choices

	guided (this paper)	unassisted
matrix	list of rows	flat vector of length n^2
scheme	ordered bag	list, up to `perm`
main theorem ends with	`(equal res sch2)`	`(perm res S2)`
events	`define` / `defrule`	`defun` / `defthm`
definitions	disabled by default	enabled by default

```
(defthm flip-graph-strongly-connected          ; the unassisted main theorem
  (implies (and (<= 2 n) (schemep n S) (schemep n S2))
           (and (path-ok-p (sq n) (connect-path n S S2) S)
                (perm (apply-path (sq n) (connect-path n S S2) S) S2))))
```

A different way to use the spanning lemma

Both need a chain $A_0 \to \cdots \to X$ with **nonzero partial sums**.

Guided

- thin the A-components to an **independent** pool
- solve for coefficients over the pool
- independence $\Rightarrow$ nonzero sums

Unassisted

- take the A-components that sum to $A_0 + X$
- **cancel duplicate pairs**: now a set
- walk **greedily**: always pick an element different from the running sum

- About 600 lines, against our 791-line span theory plus a 2505-line book.

Performance, and checking the two agree

connectivity proof	books	lines	functions	theorems	certification (serial)
guided (this paper)	19	6363	137	527	**51 s**
unassisted	9	3005	108	409	**1240 s**

- Half the lines, but **24 times slower** to certify.
- Cause: enabled definitions. Proofs open recursive functions and split into many cases. Six theorems and two termination proofs take 90% of the time.

Do they prove the same theorem? Tested with ACL2s (`defdata`, `test?`):

- a translation between the two representations; random correct schemes for $n = 2, 3$; 300 trials per property
- correct schemes agree, valid moves correspond, and each development's path works under the other's definitions
- one difference found: the flip at position B has its two summands in the other order

Conclusion

- The matrix multiplication flip graph over $\mathbb{F}_2$ is strongly connected with flips and pluses: fully proved in ACL2.
  - 29 books, 215 definitions, 791 theorems; no trust tags, `skip-proofs` or `defaxiom`
  - also: correct schemes are exactly the bilinear multiplication algorithms, up to a natural equivalence
- With an AI assistant: we set the conventions, the model wrote definitions, theorems and proofs.
- Unassisted, the model can do the whole proof, and found a simpler argument. What it lacked was how to use ACL2 well.

Future work

- Other fields, e.g. $\mathbb{Q}$: no proof for flip and plus edges is known yet.
- Give a model only the definitions and the theorem statement, without the paper proof. Does it find the lemmas?

Thank you!