Last Lab: Pentomino

Advanced Algorithms and Computation

Title: Pentomino Lab Report
Name: Robert Rice
Date: December 2, 2025
Student ID: 657588340
Language: Java

---

1. Problem Statement

The goal of this lab is to implement a general-purpose solver for the pentomino tiling puzzle. We will achieve this by reducing to the *Exact Cover* problem and solving that with Knuth's Algorithm X using the Dancing Links (DLX) data structure.

The puzzle has twelve free pentominoes (F, I, L, N, P, T, U, V, W, X, Y, Z), each covering five unit squares. A valid tiling place requires all twelve pieces on a board so that

- each piece is used exactly once,
- the board is fully covered, and
- no two pieces overlap.

Since there are 12 pieces of area 5, any board that can be tiled must have an area 60. In this assignment, we focus on the four rectangular boards 3×20, 4×15, 5×12, and 6×10, which are known to be tileable in

$$2,\ 368,\ 1010,\ 2339$$

distinct ways, respectively. When rotated and flipped, these copies of the same tiling are not counted separately.

The program is required to

- use *exactly* the encoding of pentomino shapes and orientations given in the assignment file,
- reduce the tiling problem for a given board to an instance of Exact Cover, and
- use the supplied `DLX.java` skeleton to implement Algorithm X and count solutions.

The input in the program is a single integer $n \in \{3, 4, 5, 6\}$. The output is the number of solutions for the corresponding rectangle: $n = 3 \Rightarrow 3{\times}20$, $n = 4 \Rightarrow 4{\times}15$, $n = 5 \Rightarrow 5{\times}12$, $n = 6 \Rightarrow 6 \times 10$.

2. Results

After implementing the solver and running it on all required inputs, I obtained the following results:

Board	Input n	Expected # of solutions	Program output
3×20	3	2	2
4×15	4	368	368
5×12	5	1010	1010
6×10	6	2339	2339

Initially, when I used all the orientations for the F pentomino exactly as listed in the shape file, my program reported 8 solutions for the board 3×20. This matches the "un-normalized" counting where rotated and flipped copies of a tiling are considered distinct. After applying the normalization suggested in the assignment (restricting F to two canonical orientations), the counts changed to the expected values in the table above.

The final outputs match the known counts for all four rectangular boards, which provides a strong sanity check that both the reduction and the implementation of Algorithm X are correct.

Testing Strategy

I used a staged testing approach:

1. First, I implemented `DLX.search` and tested it on small, hand-constructed exact cover instances (for example, 4 columns with a few rows where I knew there were exactly 2 covers). This verified that the Dancing Links operations and recursion were working as expected.
2. Second, I verified that the pentomino encoding produced a reasonable number of rows in the exact cover matrix (each legal placement generates one row, always with exactly six entries: one piece column and five cell columns).
3. Finally, I ran the solver on the four required boards and compared the solution counts with the reference counts provided in the assignment notes.

3. Algorithm Framework

3.1 Encoding of Pentomino Pieces

I followed the encoding specified in the assignment and the provided shape file:

- Each oriented pentomino is defined relative to an origin in the top-left grid square inside the shape, with coordinates $(0, 0)$.
- The other four squares of the piece are stored as four $(\Delta x, \Delta y)$ pairs in an array of eight lengths. For example,

  $$[1, -1,\ 2, -1,\ 1, 0,\ 1, 1]$$

  means that in addition to the origin, the four squares are at $(1, -1)$, $(2, -1)$, $(1, 0)$, and $(1, 1)$ relative to the origin.
- All orientations provided (rotations and reflections) are grouped into arrays like `static final int[][] F`, `I`, ..., `Z`, and then assembled into

  ```
  static final int[][][] SHAPES = {F, I, L, N, P, T, U, V, W, X, Y, Z};
  ```

in that exact order.

To avoid counting trivial symmetric duplicates, I applied one of the suggested normalizations from the homework: I restricted the `F` piece to two canonical orientations (the normalized F given in the handout) and left all other pieces with the full set of orientations from the shape file. With this choice, the solver counts each essentially different tiling once.

3.2 Generating Legal Placements

For a given board of size `rows` × `cols`, I generated all legal placements of every oriented piece using the nested loops recommended in the assignment:

```
for each piece p in SHAPES:
    for each orientation O of p:
        for row = 0 .. rows-1:
            for col = 0 .. cols-1:
                place origin at (row, col)
                compute positions of the other four squares
                if any square is out of bounds:
                    placement is illegal
                else:
                    record the 5 board cells as one placement
```

Each placement is stored as:

- a piece index (0–11), and
- five cell indices in row-major order, where cell (r, c) corresponds to index $r \cdot \texttt{cols} + c$.

This data goes directly into the sparse matrix representation for Exact Cover.

3.3 Reduction to Exact Cover

The reduction is standard for pentomino tilings:

- **Columns** represent constraints:
  - 12 columns for the pieces (each pentomino must be used exactly once).
  - 60 columns for the board cells (each cell must be covered exactly once).

  Thus, there are $12 + 60 = 72$ columns in total.
- **Rows** represent possible placements: each legal placement of a piece corresponds to one row with exactly six 1s: one in the column of the piece itself and one in each of the five board-cell columns it covers.

I implemented this as a sparse matrix: each row is stored simply as a list of column indices where that row has value 1. This representation is passed directly to the `DLX` constructor:

- The column indices $0 \ldots 11$ are piece constraints.
- The column indices $12 \ldots 71$ are cell constraints.

3.4 Algorithm X and Dancing Links

The `DLX.java` file implements the Dancing Links data structure as a network of doubly-linked nodes. My main task was to implement the `search(int k)` method following Knuth's Algorithm X:

1. If there are no remaining columns (i.e., `head.R == head`), a solution has been found. Increment the solution counter and return.
2. Choose the column with the smallest number of nodes using `selectColumn()`.
3. Cover that column.
4. For each row that has a 1 in that column:
   (a) Add this row to the partial solution (push it onto a `LinkedList<DataNode>`).
   (b) Cover all other columns that contain a 1 in this row.
   (c) Recursively, call `search(k+1)`.
   (d) In reverse order, uncover those columns and remove the row from the partial solution.
5. Uncover the chosen column and return.

The advantage of Dancing Links is that each cover / uncover operation runs in time proportional to the number of nodes actually affected, and those operations are reversible in constant time per node. This makes Algorithm X much more efficient than a naive implementation of backtracking on a dense matrix or on an explicit adjacency structure.

4. Complexity Analysis

4.1 Generation of Placements

Let $R \times C$ be the size of the board (here $R \cdot C = 60$). For each of the 12 pieces and each of its orientations (at most 8), we try to place the origin at every board position. The complexity of generating placements is, therefore, the complex.

$$O(12 \cdot \text{orientations} \cdot R \cdot C),$$

which is $O(RC)$ up to a constant factor, since the number of orientations per piece is fixed and small.

4.2 Algorithm X / DLX

The Exact Cover problem is NP-complete, so in the worst case the running time of Algorithm X is exponential in the number of pieces. The theoretical complexity of the worst-case can be expressed as $O(b^d)$ where b is the branching factor (choices at each step) and d is the depth of the search tree (up to 12 in this problem).

However, several factors make the algorithm practical in this setting:

- The heuristic of always choosing the column with the smallest number of nodes tends to significantly reduce the branching factor.

- The Dancing Links representation allows for very fast cover/uncover operations, so the constant factors in the backtracking are small.
- The structure of pentomino tilings is highly constrained, so many branches die early when a partial placement cannot possibly extend to a full cover.
- Normalizing the F pentomino reduces redundant symmetric solutions, which trims the search tree and also aligns the counts with the assignment's convention.

Empirically, on a modern machine the solver finds all solutions for the four rectangles in a reasonable amount of time (from fractions of a second to a few seconds, depending on the board).

The spatial complexity is $O(N)$ in the number of nodes in the Dancing Links structure. Each legal placement creates six nodes (one for the piece column and five for cell columns), and the total number of placements is bounded, so overall memory usage remains manageable.

5. Lessons Learned, Challenges, and Conclusion

5.1 Lessons Learned

- **Reductions in practice:** This lab was a concrete example of how an unfamiliar puzzle (pentomino tiling) can be reduced to a known hard problem (Exact Cover) and then solved by a general algorithmic framework instead of an ad-hoc search.
- **Backtracking and pruning:** Implementing Algorithm X clarified how systematic backtracking with a good heuristic (minimum column size) can dramatically reduce search space compared to a naive exhaustive search.

5.2 Challenges Faced

The most error-prone parts were:

- Getting the indexing of the board cells correct when converting (r, c) to a single index and then back again mentally when debugging.
- Ensure that each placement generated exactly five distinct cell indices in addition to the origin and that each row in the exact cover matrix always had length 6.
- Debugging the discrepancy between 8 solutions and 2 solutions on the 3×20 board: this ultimately came down to understanding the counting convention and correctly applying the F-piece normalization.

Testing on small, artificial Exact Cover instances before running the full pentomino solver was very helpful in isolating mistakes in the `search` implementation and in building confidence before trusting the large cases.

5.3 Feedback and Conclusion

In general, the structure of the assignment (encoding the $\rightarrow$ reduction $\rightarrow$ Algorithm X) made the connection between theory and implementation very clear. Having the DLX

skeleton available, let me focus on understanding Algorithm X and the reduction rather than on low level pointer bugs.

Compared to earlier labs, this assignment felt like a more "real-world" use of an algorithmic framework: the same Exact Cover + DLX engine could solve many different constraint problems just by changing the matrix. That was a useful lesson about how to structure problem-solving in algorithms and in software engineering.

6. References

- Donald E. Knuth, "Dancing Links," 2000.
- Wikipedia: "Exact Cover" (for small test examples and basic definitions).
- Standard references on pentominoes and known solution counts for the 3×20, 4×15, 5×12, and 6×10 rectangles.