NYT Strands Solver

Jaden Zhou, Alexander Rees, Saran Jagadeesan Uma, Ananya Hegde

CS5100 Foundations of Artificial Intelligence Spring 2025

Abstract

Solving word puzzles like the New York Times Strands[5] requires balancing semantic relevance with spatial constraints while efficiently navigating combinatorial possibilities. This project aimed to develop an intelligent solver that combines natural language understanding with constraint optimization to generate accurate solutions and solve a given Strands game. The system employs a three-stage pipeline: a trie + DFS approach which extracts all possible valid words from the given word grid, following which a Sentence Transformer (SBERT) ranks them by thematic coherence. Finally, a constraint satisfaction problem solver selects the optimal subset that satisfies the spatial rules. We also explored alternative methods including Dancing Links (DLX), and Genetic Algorithms, which each had trade-offs in efficiency and constraint enforcement. Despite challenges such as theme ambiguity, niche vocabulary, and computational bottlenecks, the solver showed strong performance across multiple real-world puzzles and highlighted the effectiveness of combining linguistic models with spatial reasoning.

1 Introduction

The New York Times Strands is a daily word puzzle that challenges players to identify interconnected words within a grid, all linked by a central theme. One of the puzzle's unique features is the *spangram*, a special word that spans from one side of the grid to the opposite side, encapsulating the theme and anchoring the overall solution.

Solving Strands puzzles manually can often be a tedious and time-consuming task, especially given the complexity of interpreting thematic clues and navigating the spatial layout of potential words. This inherent challenge, paired with our interest in natural language processing and optimization techniques, motivated us to design an automated solver capable of tackling Strands puzzles with speed and accuracy.

The topic was both intellectually stimulating and genuinely enjoyable to explore. Our goal was to combine semantic understanding with spatial reasoning, ultimately creating a system that mimics the human thought process involved in solving such puzzles. This report details the architecture, implementation, and evaluation of our Strands Solver.

2 Related Work

To date, there is no end-to-end solver in the literature that both enforces the Strands puzzle's spatial constraints exactly and ranks candidates by thematic coherence. Instead, our approach stitches together ideas from several adjacent areas. We found several similar approaches to projects that worked with our problem.

2.1 Depth First Search

One very similar problem to finding all the words in a grid based puzzle is for Boggle. It is a game played on a 4 by 4 grid of letters. You try to find as many words as possible by moving to all letters, also including the diagonals. This is very similar to our DFS problem of finding all the words in the grid. We looked online and found some github repositories of people building solvers for this. They generally had the approach of solving with a prefix tree and a DFS algorithm to find the words.[3] Based on this we built the algorithm for our specific constraints. An example repository of a boggle solver is linked in the references.

2.2 Transformer

Another similar project relevant to our constraint-based stage is Dr.Fill, Matthew Ginsberg's automated solver for crosswords. Dr.Fill

converts every crossword into a weighted CSP, each across/down entry becomes a variable, cross-checks form hard constraints, and a learned probability $p(fill|clue)$ supplies soft weights.[1] We mirror that split, exact cover for board legality, learned scores for thematic fit, scaled down to Strands.

2.3 Constraint Satisfaction

Covering a geometric board with fixed-shape pieces is a classic CSP. Exact-cover tilings have previously been efficiently solved by Knuth's Algorithm X with Dancing Links (DLX). DLX variants power Blokus bots that enumerate millions of piece placements while pruning most of the tree[2]. Our Python backtracker echoes DLX's matrix idea: each candidate word path is a row, each grid cell a column, and the solver selects a subset covering every cell exactly once.

3 Methods

In this section we describe the three core steps that make our solver both thorough and fast. First, we load the full dictionary into a Trie so that prefix checks during our grid-wide DFS run in time proportional only to the length of the current string, not the size of the dictionary. We kick off a recursive search from every cell, pruning entire branches the moment we hit a prefix that doesn't exist, and record every valid word path. Second, we take those raw candidates and score them with lightweight NLP: a combination of semantic similarity to the theme hint, masked-language-model likelihood, and word-frequency heuristics to rank the most relevant entries at the top. Finally, we treat the ranked list as a small backtracking CSP that enforces no letter reuse and valid overlaps, selecting the highest-scoring words that form a consistent solution. Together, these stages balance exhaustive search, statistical ranking, and constraint solving to deliver accurate Strand answers quickly.

3.1 Depth First Search

A Strands word is just a path through the 6×8 grid where successive letters lie in one of the eight neighboring cells. Enumerating all such paths is a classic search-tree problem: the root is the starting cell, each level adds one adjacent letter, and the tree's branching factor is at most 8. A naive traversal would explode (8^{16}) paths at the maximum word length), but when DFS is paired with two cheap pruning tests, (i) a maximum-length cutoff and (ii) a "prefix exists in the trie?" check, we can discard dead branches almost immediately. The result is an exhaustive yet still lightweight procedure that feeds high-quality candidates to the later NLP and CSP stages.

3.1.1 Trie

A Trie is just a tree where each node represents a letter, and walking down from the root spells out a word prefix. We insert every dictionary word into this structure so that checking "does any word start with 'XYZ'?" is as simple as following three letter-nodes and seeing if we've fallen off the tree. If the path exists, there's at least one word with that prefix; if it doesn't, we know immediately we can't form any valid word from those letters, which makes our DFS pruning lightning fast. Storing words this way also lets us tag end-of-word nodes, so when a prefix exactly matches a full word we can grab it right away without any extra lookups.

3.1.2 Pseudocode

Algorithm 1: DFS Find Words & Paths

$\mathcal{C} \leftarrow \emptyset$
function DFS(r, c, word, path)
  if $|word| > M$ **or not** T.HasPrefix($word$) **then**
    return
  end if
  if $|word| \geq m$ **and** T.IsWord($word$) **then**
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{(word, path)\}$
  end if
  for $\Delta r, \Delta c \in \{-1, 0, 1\}^2 \setminus \{(0,0)\}$ **do**
    $r' \leftarrow r + \Delta r,\ c' \leftarrow c + \Delta c$
    if $0 \leq r' < R \ \wedge\ 0 \leq c' < C \ \wedge\ (r', c') \notin path$ **then**
      DFS(r', c', word+$G_{r'c'}$, $path \cup \{(r', c')\}$)
    end if
  end for
end function
function FindWords
  for $r \leftarrow 0$ **to** $R - 1$ **do**
    for $c \leftarrow 0$ **to** $C - 1$ **do**
      DFS(r, c, G_{rc}, $\{(r, c)\}$)
    end for
  end for
  return $\mathcal{C}$
end function

Algorithm 1 seeds a depth-first search from every grid cell, growing a path through the 8-connected neighbors while the partial string remains (i) no longer than the length cap M and (ii) a valid prefix in the trie T. When the current string is at least the minimum length m and T.IsWord confirms it is complete, the pair (word, path) is pushed into the candidate list $\mathcal{C}$. Because branches are cut off the instant they violate the length limit or fail the prefix test, the algorithm exhaustively enumerates all legal words and their coordinates without ever exploring useless sub-trees.

3.2 NLP Candidate Ranking

To solve the NYC Strands puzzle more effectively, we built an NLP-based system to rank candidate words based on how well they fit a given theme. Our goal was to go beyond simple string matching and instead understand the meaning behind words and how they relate to each other. To do this, we combined three different signals: semantic similarity, language model predictions, and word frequency.

3.2.1 Cosine Similarity

We start by checking how similar a candidate word is to the theme using cosine similarity. This involves turning both the candidate and the theme into vector representations (embeddings) and then measuring the angle between them. The closer the angle (or the higher the cosine similarity), the more semantically related they are.

3.2.2 Sentence Transformer

To get these embeddings, we use the all-mpnet-base-v2 model from the SentenceTransformers library. This model was able to capture the context really well, making it great for comparing phrases and understanding which words go with a given theme. We use the model to encode both the theme and the candidate, then compare them using PyTorch's built-in cosine similarity function.

3.2.3 Language Model Scoring

We also use a masked language model to see how likely it is for a word to appear in a sentence given the theme. We used the prompt: "The theme is {`theme`}. A related word is `[MASK]`." Then, using BERT, we check if our candidate word shows up in the model's top predictions. If it does, that's a good sign the word fits well in the context.

3.2.4 Word Frequency

Sometimes, a word might be a great fit semantically but is super obscure. To help avoid that, we check how common each candidate word is using the `wordfreq` library. This helps us favor more familiar and puzzle-appropriate words.

3.2.5 Ranking Function

We bring everything together using a weighted scoring system:

$$score(\text{word}) = \alpha \cdot \text{sim} + \beta \cdot \text{lm} + \gamma \cdot \text{freq}$$

Here, α, β, and γ are weights we assign to control the importance of each factor (we used 0.5 for similarity, 0.3 for the language model, and 0.2 for frequency). The final result is a ranked list of candidates, with the most thematically relevant words at the top.

3.3 Constraint Satisfaction Problem

Strands ultimately asks us to choose a set of candidate words that (i) cover every grid cell exactly once, (ii) never overlap, and (iii) satisfy the puzzle-specified word-count. Those requirements map naturally onto an Exact Cover formulation:

3.3.1 Exact Cover

Each candidate word path can be viewed as a row in a 0-1 matrix whose columns correspond to the 48 grid cells. A row has a 1 wherever the word occupies that cell. Finding a legal Strands solution is therefore an **Exact Cover** problem: select a subset of rows so that every column contains exactly one 1 (no overlaps, full coverage) and, if the puzzle specifies it, the subset size equals the mandated word-count k.

3.3.2 Pseudocode

Algorithm 2 backtracks through the score-sorted candidate list. At index i it either *chooses* the current word, adding its cell set P_i to *covered* and recursing, or *skips* it. A branch succeeds as soon as $|\textit{covered}| = RC$ (and $|S| = k$ if required), otherwise failure propagates back, undoing choices along the way. Because the list is ordered by semantic score, high-quality exact covers are typically discovered early, keeping the search shallow in practice.

Algorithm 2: Back-tracking Exact-Cover Solver

```
function SEARCH(i, S, covered)
    if |covered| = N then
        return S
    end if
    if i = |C| or (k ≠ none ∧ |S| ≥ k) then
        return ∅
    end if
    (w, P, s) ← C[i]
    if P ∩ covered = ∅ then
        S ← S ∪ {(w, P, s)}
        covered ← covered ∪ P
        sol ← SEARCH(i + 1, S, covered)
        if sol ≠ ∅ then
            return sol
        end if
        S ← S \ {(w, P, s)}
        covered ← covered \ P
    end if
    return SEARCH(i + 1, S, covered)
end function
    return SEARCH(0, ∅, ∅)
```

4 Experiments

Our experiments evaluate three alternative tiling engines, our primary CSP back-tracker, a DLX exact-cover core, and a heuristic Genetic Algorithm, on both *speed* and *solution quality*. The next subsections summarize the implementation details, strengths, and weaknesses of each approach.

4.1 Dancing Links (DLX)

Algorithm 3: DLX Exact-Cover Solver

```
function SEARCH(S)
    if all columns are covered then
        return S if k = none or |S| = k else ∅
    end if
    col ← column with fewest 1's COVER(col)
    for all row ∈ rows(col) do
        S ← S ∪ {row}
        for all col' ∈ columns(row) do COVER(col')
        end for
        result ← SEARCH(S)
        if result ≠ ∅ then
            return result
        end if
        S ← S \ {row}
        for all col' ∈ columns(row) reverse do UNCOVER(col')
        end for
    end forUNCOVER(col)
    return ∅
end function
    return SEARCH(∅)
```

As an alternative to the CSP, we used the DLX which is particularly effective for problems involving exact cover, where we need to find an arrangement of words that covers the grid without overlaps[4]. By representing the grid as a matrix with rows for word positions and columns for grid cells, the algorithm was able to efficiently explore all valid word placements while enforcing spatial constraints. Our algorithm works by recursively selecting columns with the fewest constraints, covering them, and backtracking when necessary. This approach found the optimal arrangement of words that fit the grid while ensuring all constraints, such as no overlapping and full grid coverage, but was unsuccessful at fulfilling the semantic constraints.

4.2 Genetic Algorithm[6]

Also an alternative to the CSP, we tried a genetic algorithm to find the best combination of words that fill the entire grid without overlapping. It starts by creating many random combinations of words that don't clash. The fitness function evaluates how well a chromosome covers the entire grid without overlaps while maximizing the cumulative semantic scores of the words. Over time, the algorithm keeps the best solutions and creates new ones by mixing and slightly changing them. This process is repeated over many rounds to improve the results. In the end, it returns the best set of words that fit the grid and match the puzzle's theme. While the genetic algorithm was useful for exploring different combinations, it lacked strict constraint enforcement. It struggled to guarantee full grid coverage and produced invalid solutions. In contrast, the CSP approach was better suited for our project.

5 Results

Our evaluation focuses on end-to-end runtime and solution quality on archived Strands puzzles. The next three subsections give the headline numbers for each solver.

5.1 CSP

End-to-end runs typically took around **30** min. When we provided the exact target word count up front, the solver spent closer to **45** min, reflecting the extra backtracking needed to confirm completeness under that constraint.

5.2 DLX

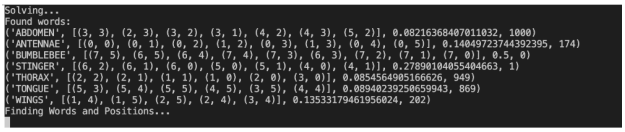
The exact-cover computation itself finishes in a matter of *seconds*, since DLX is highly optimized for spatial constraints. However, without integrated semantic scoring, it often selects words that ignore the puzzle's theme.

5.3 Genetic Algorithm

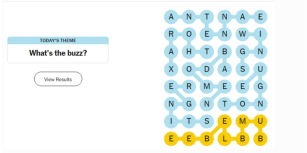
We did not record precise timings, but GA runs generally completed in a few minutes, much faster than CSP, yet frequently yielded either incomplete grid coverage or semantically mismatched words, due to its heuristic rather than exact approach.

5.4 Example Run April 10th

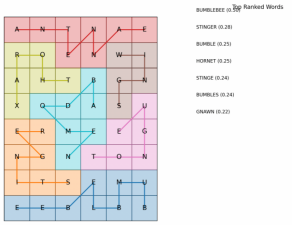
The 4/10 puzzle's theme clue was *"What's the buzz?"*. Our solver correctly identified the spangram *BUMBLEBEE* (touching the left and right edges) along with all six theme words: *antennae*, *thorax*, *wings*, *abdomen*, *tongue*, and *stinger*. Together they cover every grid cell exactly once, yielding a perfect solution without hints.



Candidates generated by the DFS + Trie



April 10th Strands puzzle



Visualized Results

6 Conclusions and Future Work

Throughout the development of the AI Strands Solver, we gained significant insight into the challenges of designing systems that mimic human-level reasoning for language-based puzzles. A major learning outcome was developing the ability to select appropriate models based on the nature and constraints of a given problem statement. In particular, we learned how to fine-tune and apply models according to the linguistic patterns and contextual clues required to solve the NYC Strands game. This project also helped strengthen our skills in model evaluation and iterative debugging.

Moving forward, we aim to enhance the solver's performance by addressing existing limitations such as semantic ambiguity and the computational cost associated with processing large sets of candidate words. Optimizing prompt design and experimenting with more efficient model configurations will be crucial to this effort.

Another key direction is automating the solver to function independently in a web browser. This would involve building a robust pipeline that fetches the daily puzzle, interprets the grid layout, and submits accurate solutions, fully hands-free. Such automation will not only showcase the solver's real-world applicability but also push us toward building more general-purpose AI agents.

7 Additional Deliverables

7.1 GitHub Repository

7.2 Contributions

Alexander focused on the DFS and Trie algorithm, wrote a scraping script for data and built a visualization tool. Saran worked with the transformer and came up with the ranking approach. Ananya worked on the different approaches, comparing the different algorithms and implementing those. Jaden focused on the CSP and putting everything together by coding the algorithms and pushing the initial solution.

References

[1] Matthew L. Ginsberg. Dr. Fill: Crosswords and an implemented solver for singly weighted CSPs. *CoRR*, abs/1401.4597, 2014. arXiv:1401.4597.

[2] Ali Jahanshahi, MohammadKazem Taram, and Nariman Eskandari. Blokus duo game on fpga. In *Proceedings of the 17th CSI International Symposium on Computer Architecture & Digital Systems (CADS)*, pages 145–149, Tehran, Iran, 2013. IEEE.

[3] Craig S. Kaplan. A computer analysis of boggle™. *University of Washington Technical Report*, 2001.

[4] Donald E. Knuth. Dancing links. In *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*. Palgrave, 2000.

[5] The New York Times. Strands, 2024.

[6] Rajagopal Venkat. Artificial intelligence resources — chapter 4: Local search and optimization, 2024.