# Comparative Evaluation of Search Algorithms for Crossword Puzzle Solving

ISMAEL CARRASCO MKHAZNI
*Innovative Information Systems (2IS)*
*Department of Information Technologies*
*University of Toulouse Capitole*
Toulouse, France
ismael.carrasco-mkhazni@ut-capitole.fr

ABDULLAH TARIQ
*Innovative Information Systems (2IS)*
*Department of Information Technologies*
*University of Toulouse Capitole*
Toulouse, France
abdullah.tariq@ut-capitole.fr

MINTESNOT YIMER
*Innovative Information Systems (2IS)*
*Department of Information Technologies*
*University of Toulouse Capitole*
Toulouse, France
mintesnot.yimer@ut-capitole.fr

*Abstract*— This paper evaluates three Artificial Intelligence techniques—Depth-First Search (DFS), Breadth-First Search (BFS), and Constraint Satisfaction Problem (CSP) for solving crossword puzzles. The task involves filling a grid with valid words from a dictionary under structural constraints such as word length, direction, and letter intersections. We implement all three methods and compare their performance using small, medium, and large grid configurations. The CSP approach clearly outperforms DFS and BFS in terms of scalability and efficiency. We present a detailed discussion on the strengths and limitations of each method and conclude with suggestions for future work.

## I. Introduction (*Heading 1*)

Crossword puzzle solving is a popular benchmark for evaluating algorithmic efficiency in Artificial Intelligence (AI). The problem consists of filling in a grid of horizontal and vertical word slots using a dictionary, such that all overlapping letters are consistent and every filled word is valid. While humans solve such puzzles using semantics and pattern recognition, automated techniques must rely on well-defined rules and search strategies.

In this project, we compare three fundamental AI techniques:

- Depth-First Search (DFS)
- Breadth-First Search (BFS)
- Constraint Satisfaction Problem (CSP) modeling

Our goal is to evaluate their effectiveness in solving crossword puzzles under increasing levels of complexity.

## II. Problem Modeling

### A. Search-Based Approach

When modeled as a search problem, each state represents a partially filled board, where some of the words have already been placed while others slots remain empty. The initial state is an entirely empty grid, and the goal state is achieved when all slots are filled and the intersections are correct.

Actions are inserting valid words from the dictionary into available slots on the board. For example, placing “HOMES” in the next available horizontal/vertical slot with five letters is a valid action. Applying these actions to a node leads to a new state, updated with the new world in place.

The goal test checks whether the board is completely filled with words that fit together without conflict, giving a solution.

The primary challenge is the combinatorial explosion of the state space. At worst, the number of states can be approximated as O(M^N), where M is the number of words in the dictionary and N is the number of slots to be filled. However, the branching factor (number of possible actions from a given state) can vary. Without constraints, it is roughly O(M). Although, when applying the constraint in the actions (word lengths), the branching factor is significantly reduced, helping to prune the search space effectively.

### B. CSP Approach

The state of the crossword puzzle is defined by the current configuration of the board and the list of positions that are still empty. In the initial state, the board is completely empty; intermediate states have some letters placed; and the goal state is one where every cell is filled with letters that form valid words in every row and column. Each action in the process is the placement of a single letter into one of these empty cells. The chosen letter must be consistent with the partial words already in the row and column, ensuring that when more letters are added, the final words will be valid. The transition function is a straightforward operation that takes the current state and an action, then outputs a new state where the action (i.e., the placement of the letter) has been applied.

A goal test is used to check whether the board is completely filled and all sequences of letters correspond to valid dictionary words. Only when there are no empty cells left and every row and column forms a proper word does the state qualify as a solution.

The overall complexity of the search process is quite high because each empty cell has a potential of up to 26 choices (assuming the English alphabet), so the number of states in the worst case can grow exponentially—approximately O(26^N) for N empty cells.

However, practical constraints, such as the requirement for intersecting words to form valid letter combinations, reduce the branching factor significantly. Even so, the effective search space remains extremely large when each step allows multiple letter choices at each empty position.

## III. Methodology

### A. DFS and BFS Implementation

To explore possible solutions for the problem, we have implemented two classic search strategies: Breadth-First Search (BFS) and Depth-First Search (DFS). Both approaches aim to find a valid way to fill the crossword grid by placing words in the correct slots, but they do so in different orders and with different trade-offs.

At the core of the implementation are two main components: The *Node* and the *Grid* classes.

- The *Node* represents a possible state of the puzzle. It keeps track of the words already placed and how deep it is in the search.
- The *Grid* contains all the information about the problem: the size of each slot, where words intersect and the list of words to choose from. It also includes methods to check if a state meets the goals, get the possible actions from a node and to generate the new states from those possibilities.

The functionality behind both BFS and DFS is pretty similar. The difference is in the way they explore new nodes, due to the data structure used to save the frontier:

- In BFS, the algorithm explores at the current level before moving on to deeper levels. The reason for this is that it uses a FIFO queue (First In, First Out) to keep track of which node to explore next. This approach guarantees that the solution found will use the fewest steps, but it can require a lot of memory.
- In DFS, the algorithm goes as deep as possible, trying to fill one slot after another before going back and trying different options. The reason for this is that it uses a LIFO queue (Last In, First Out) to keep track of which node to explore next. This approach uses less memory but may take longer to find a solution if the correct path is buried deep.

The steps taken in the algorithm start with the current node, which represents the current state. Using the information stored in the *Grid* object, it determines which actions are possible based on the length of the next available slot and the words that have already been used.

Once that is done, the transition function is applied to generate new nodes with the new world implemented. These new nodes are then added to the frontier to be explored afterwards.

This process continues until a goal state is reached: a fully completed board where all slots are filled correctly and the intersecting letters match

### B. CSP Implementation

The implementation centers on solving a crossword puzzle problem using two different constraint solvers: one based on the python-constraint package and another using OR-Tools' CP-SAT solver.

At the heart of the implementation there is a custom data structure: The Slot class

- The Slot class encapsulates each word's position, orientation, length, and an optional clue. The constructor for Slot assigns a unique identifier and additional attributes such as the starting row, column, orientation (either "across" or "down"), and the length of the slot.

The process begins by loading the grid and the dictionary from a text file.

- This file represents the crossword puzzle's structure using distinct tokens, where a "0" marks a black cell, a "." represents a white cell with no associated clue, and any other token (usually a number) indicates a white cell that has an assigned clue.
- The grid is read into a two-dimensional list where each cell is stored as a tuple, with the first element indicating its fillable status and the second the clue if it exists.

For the functionality of Slot Extraction and Intersection Computation we used these method extract_slots(grid, min_length=2)

- The implementation extracts "slots" by scanning row-wise for across words and column-wise for down words. Only sequences that exceed a minimum length threshold are recognized as valid word slots. The resulting list of slots is then sorted

by their starting positions to ensure a consistent order for further processing.

- Subsequently, the code computes the intersections between slots—in particular, it identifies the grid cells where an "across" slot and a "down" slot overlap. For every overlapping pair, the solver saves the corresponding positions (using one-indexed positions) in each word such that the letters in those positions must be identical.

For the first scenario (small grid–small dictionary) and second scenario (large grid–large dictionary), as shown respectively in Fig. 1 and Fig. 2, we found a solution using Python constraint, but for the harder scenario (large grid–small dictionary), Python constraint failed to provide a solution, so we used the CP-SAT model, as shown in Fig. 3.

- In the python-constraint model, the program assigns each slot a domain that consists of all words from a dictionary that match its required length. A global constraint is added to enforce that every slot receives a unique word. Then, for each pair of intersecting slots, a lambda constraint is used so that the letter in the specified position of one word must equal the letter in the corresponding position of the intersecting word.
- If any slot ends up with an empty domain because no dictionary word of the appropriate length is available, the solver aborts with an appropriate message.
- In the CP-SAT version, each slot is again associated with a domain of words of matching length. Instead of directly assigning words as domain elements, each slot is modeled as an integer variable representing the index of a word within its domain list.
- The intersection constraints are then enforced using allowed assignments; that is, for every pair of intersecting slots, the code collects all valid pairs of domain indices where the letters at the intersecting positions match. These allowed pairs are provided to the model using the AddAllowedAssignments constraint, ensuring that only consistent assignments appear in any solution.

The visualization functionality is provided by helper functions that print the grid in an intuitive manner. White cells are initially shown as blanks and black cells as "#" symbols. Letters from the solution are then overlaid on the grid in the corresponding positions as determined by the cell coordinates computed earlier by each slot. This offers the user a clear picture of the completed crossword puzzle.

## IV. Results

### A. Small Grid Baseline

A minimal test case was created to validate the correctness of the BFS and DFS implementations. The 3x3 grid had two horizontal 3-letter slots and one vertical 3-letter slot that matched both horizontal words in both the first and last letter, with a small dictionary of three words: ["GOD", "DAM", "DOG"].

| Algorithm | Time (s) | Nodes | Solution |
|---|---|---|---|
| BFS | $7.96\times10^{-5}$ | 15 | H=['DAM', 'GOD'] V=['DOG'] |
| DFS | $6.01\times10^{-5}$ | 11 | H=['DAM', 'GOD'] V=['DOG'] |

The correct solution was reached with minimal computation, showing both methods work for constrained inputs.

To really ensure that a solution was reached, we used the same grid with the small dictionary provided. Getting a solution from DFS but none from BFS.

| Algorithm | Time (s) | Nodes | Solution |
|---|---|---|---|
| BFS | none | none | none |
| DFS | 0.0173 s | 3873 | H=['ZZZ','ZUZ'], V=['ZIZ'] |

With this simple problem, we can see that BFS already struggles to keep up. For example, if there were 1000 three-letter words inside the dictionary for the three slots (taking in account that the words can not repeat), the number of possible combinations is:

$1000*999*998 \approx 997$ mill nodes

Meaning that most combinations will be invalid due to intersection rules, but BFS does not know that, it explores them all. Only at the end it checks if the words match at the intersections. So it explores millions of invalid states before even getting close to a valid one.

### B. Small, Medium and Large Grid Comparison

| Grid Size | Dictionary Size | Algorithm | Output | Time (s) |
|---|---|---|---|---|
| Small | Small | CSP(python constraint library) | Success | 0.1213 |

| | | | | |
|---|---|---|---|---|
| Medium | Medium | DFS | Failed | >120 |
| Medium | Medium | BFS | Failed | >120 |
| Large | Small | CSP(CP-SA mode) | Success | 60.4955 |
| Large | Large | DFS | Failed | >300 |
| Large | Large | CSP(CP-SA mode) | Success | 4.1968 |
| Large | Large | BFS | Failed | >300 |

In the medium grid scenario, DFS generated 3873 nodes and eventually produced a solution. BFS and DFS were unable to solve larger grids due to memory or time constraints.

## V. Discussion

The results highlight the scalability issues of uninformed search algorithms like DFS and BFS. While BFS explores all possibilities level-by-level, it quickly becomes intractable due to exponential growth. A theoretical analysis estimates that a 3-letter puzzle can already yield ~997 million permutations. For a 5x5 grid, the BFS state space can explode to $3.96 \times 10^{41}$ combinations, leading to estimated runtime on the order of $1.26 \times 10^{29}$ years in the worst case.

DFS, although faster and more memory-efficient than BFS, also suffers from inefficient search and can get trapped in deep but fruitless paths.

CSP handles this complexity by enforcing constraints at each step. With domain filtering and propagation (like AC-3), CSP avoids most invalid states entirely, reducing the need for exhaustive exploration.

## VI. Conclusion

Through our comparative analysis, we conclude that CSP-based modeling is the most suitable approach for solving crossword puzzles in terms of performance, scalability, and accuracy.

DFS and BFS are limited by their lack of pruning and constraint awareness.

To enhance the efficiency and scalability of the algorithm, several improvements can be made to the current DFS and BFS approaches.

First of all, one important area of development is the incorporation of early pruning. Checking partial intersections at the time of generating the actions for a node. This would prevent the algorithm from continuing down invalid paths, significantly reducing unnecessary expansions.

Another promising technique is forward checking. After placing a word in a slot, the algorithm should immediately verify that the remaining empty slots still have at least one possible word that could fit. If not, that path can be discarded early on. This prevents the construction of dead states and helps the search stay on viable paths.

Finally, the search process can also benefit from smart word ordering based on simple heuristics. For example, words that are more common, that fit more intersections or that appear earlier alphabetically could be prioritized. This increases the chances of finding valid solutions earlier in the search. These heuristics can be implemented with search methods like A*.

## References


[1] R. Dechter, Constraint Processing. Morgan Kaufmann, 2003.

[2] Gecode Team. Gecode: Generic Constraint Development Environment. http://www.gecode.org

[3] Google OR-Tools. https://developers.google.com/optimization

[4] Python Constraint Solver. https://labix.org/python-constraint

```
Solving with python-constraint...
Solution found:
Slot  1 across  at (1,1) len=3 (Clue: 1) -> MOY
Slot  7 down    at (1,1) len=7 (Clue: 1) -> MAZOUTS
Slot  8 down    at (1,3) len=3 (Clue: 2) -> YUM
Slot  2 across  at (1,5) len=3 (Clue: 3) -> YON
Slot 11 down    at (1,5) len=3 (Clue: 3) -> YAR
Slot 13 down    at (1,7) len=7 (Clue: 4) -> NOYESES
Slot  3 across  at (3,1) len=7 (Clue: 5) -> ZYMURGY
Slot 10 down    at (3,4) len=3 (Clue: 6) -> UNI
Slot  4 across  at (5,1) len=7 (Clue: 7) -> UVEITIS
Slot  9 down    at (5,3) len=3 (Clue: 8) -> EYE
Slot 12 down    at (5,5) len=3 (Clue: 9) -> TYG
Slot  5 across  at (7,1) len=3 (Clue: 10) -> SYE
Slot  6 across  at (7,5) len=3 (Clue: 11) -> GUS

Filled Grid:
M O Y # Y O N
A # U # A # O
Z Y M U R G Y
O # # N # # E
U V E I T I S
T # Y # Y # E
S Y E # G U S
Time: 0.1213 seconds
```

*Fig* 1 Result for small grid - small dictionary

```
Solution found:
Slot  1 across  at (1,1) len=5 (Clue: 1) -> COWRY
Slot 12 down    at (1,1) len=6 (Clue: 1) -> CHINTZ
Slot 14 down    at (1,3) len=11 (Clue: 2) -> WRECKMASTER
Slot 15 down    at (1,5) len=4 (Clue: 3) -> YUTZ
Slot 17 down    at (1,7) len=6 (Clue: 4) -> FANFIC
Slot  2 across  at (1,9) len=3 (Clue: 5) -> ZZZ
Slot 19 down    at (1,9) len=11 (Clue: 5) -> ZYGOPLEURAL
Slot 20 down    at (1,11) len=4 (Clue: 6) -> ZOBO
Slot  3 across  at (2,5) len=5 (Clue: 7) -> UNARY
Slot  4 across  at (3,1) len=5 (Clue: 8) -> INERT
Slot  5 across  at (4,5) len=7 (Clue: 9) -> ZUFFOLO
Slot  6 across  at (6,1) len=11 (Clue: 10) -> ZYMOTICALLY
Slot 16 down    at (6,5) len=6 (Clue: 11) -> TWENTY
Slot 21 down    at (6,11) len=6 (Clue: 12) -> YESSES
Slot  7 across  at (8,1) len=7 (Clue: 13) -> ZOSTERS
Slot 13 down    at (8,1) len=4 (Clue: 13) -> ZANY
Slot 18 down    at (8,7) len=4 (Clue: 14) -> SYPH
Slot  8 across  at (9,7) len=5 (Clue: 15) -> YURTS
Slot  9 across  at (10,3) len=5 (Clue: 16) -> ESTOP
Slot 10 across  at (11,1) len=3 (Clue: 17) -> YAR
Slot 11 across  at (11,7) len=5 (Clue: 18) -> HYLES

Filled Grid:
C O W R Y # F # Z Z Z
H # R # U N A R Y # O
I N E R T # N # G # B
N # C # Z U F F O L O
T # K # # # I # P # #
Z Y M O T I C A L L Y
# # A # W # # # E # E
Z O S T E R S # U # S
A # T # N # Y U R T S
N # E S T O P # A # E
Y A R # Y # H Y L E S
Time: 4.1968 seconds
```

*Fig 2 Result for* large grid - large dictionary

```
Solving with OR-Tools CP-SAT model...
Solution found:
Slot  1 across  at (1,1) len=5 (Clue: 1) -> UDONS
Slot 12 down    at (1,1) len=6 (Clue: 1) -> UPGUSH
Slot 14 down    at (1,3) len=11 (Clue: 2) -> OVERSTAYERS
Slot 15 down    at (1,5) len=4 (Clue: 3) -> SADI
Slot 17 down    at (1,7) len=6 (Clue: 4) -> YIRDED
Slot  2 across  at (1,9) len=3 (Clue: 5) -> HAD
Slot 19 down    at (1,9) len=11 (Clue: 5) -> HYPERDACTYL
Slot 20 down    at (1,11) len=4 (Clue: 6) -> DANT
Slot  3 across  at (2,5) len=5 (Clue: 7) -> ACIDY
Slot  4 across  at (3,1) len=5 (Clue: 8) -> GLEED
Slot  5 across  at (4,5) len=7 (Clue: 9) -> INDWELT
Slot  6 across  at (6,1) len=11 (Clue: 10) -> HOTHEADEDLY
Slot 16 down    at (6,5) len=6 (Clue: 11) -> EUOUAE
Slot 21 down    at (6,11) len=6 (Clue: 12) -> YEARLY
Slot  7 across  at (8,1) len=7 (Clue: 13) -> DAYLONG
Slot 13 down    at (8,1) len=4 (Clue: 13) -> DICH
Slot 18 down    at (8,7) len=4 (Clue: 14) -> GUST
Slot  8 across  at (9,7) len=5 (Clue: 15) -> UTTER
Slot  9 across  at (10,3) len=5 (Clue: 16) -> RYALS
Slot 10 across  at (11,1) len=3 (Clue: 17) -> HES
Slot 11 across  at (11,7) len=5 (Clue: 18) -> TALKY

Filled Grid:
U D O N S # Y # H A D
P # V # A C I D Y # A
G L E E D # R # P # N
U # R # I N D W E L T
S # S # # # E # R # #
H O T H E A D E D L Y
# # A # U # # # A # E
D A Y L O N G # C # A
I # E # U # U T T E R
C # R Y A L S # Y # L
H E S # E # T A L K Y
Time: 60.4955 seconds
```

*Fig 3  Result for* large grid - small dictionary