

OCR Word Search Solver

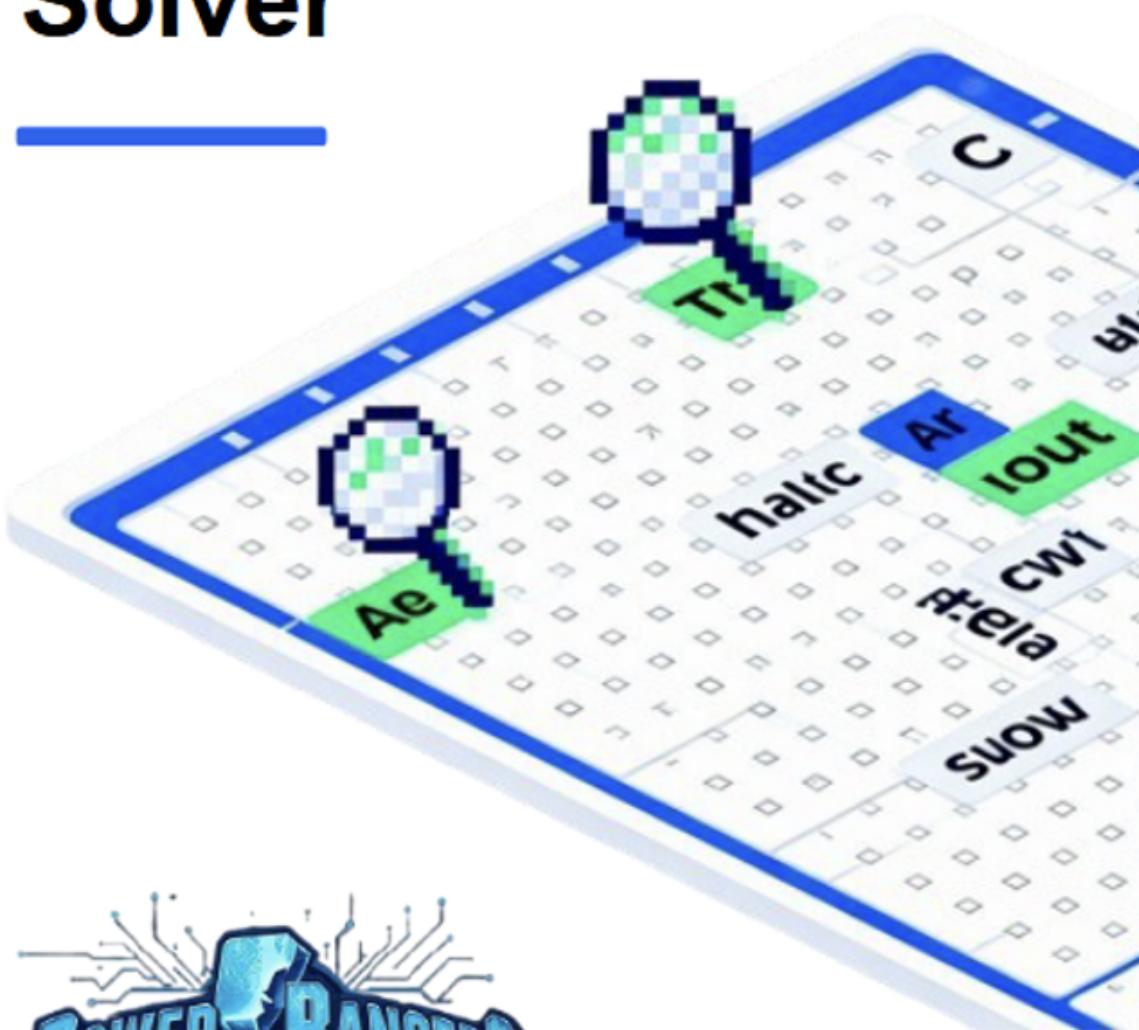


Table of Contents

Contents

1	Introduction	4
2	Task Distribution	5
3	Objectives and Project Progress	6
4	Development Methodology and Technical Justifications	8
4.1	Loading an image and interface design	8
4.2	Color Removal and Image Binarization	11
4.3	Finding and Slicing Grid, Words, and Letters	14
4.3.1	Grid Detection and Extraction	14
4.3.2	Word List Region Extraction	16
4.3.3	Slicing Words from the Word List	16
4.3.4	Letter Segmentation from Words	17
4.3.5	Challenges Encountered	18
4.4	Solver	22
4.4.1	Function Overview	22
4.4.2	Challenges Encountered	23
4.5	Solved grid	24
4.5.1	Data acquisition and preparation	24
4.5.2	Reconstructing Word Paths from Coordinates	25
4.5.3	Assigning Colours to Identified Words	25
4.5.4	Building the Highlight Map for Rendering	26
4.5.5	Console-Based Rendering Prototype	26
4.5.6	Integration into the GTK Graphical User Interface	26
4.5.7	Challenges Encountered	27
4.6	Automatic Rotation	30
4.6.1	Image Preprocessing	30
4.6.2	Edge Detection and Angle Calculation	30
4.6.3	Determining the Rotation Angle	31
4.6.4	Rotating the Image	31
4.6.5	Grid Reconstruction	31
4.6.6	Challenges and Solutions	32
5	Artificial Neural Networks	33
5.1	General Overview	33
5.2	Fundamental Principles of the Neural Network	33
5.2.1	The Formal Neuron	33
5.2.2	Non-Linearity: Activation Functions	34

5.2.3	Layer Architecture	34
5.3	Training Methodology	35
5.3.1	The Learning Cycle	35
5.4	Our Implementation	35
5.4.1	Understanding and Visualizing the Neural Network	35
5.4.2	Implementing Everything from Scratch and Managing Memory .	36
5.4.3	Building and Normalizing the Dataset	36
5.4.4	Saving and Reloading Network Weights	36
5.5	Lessons Learned from the Neural Network Implementation	37
6	Conclusion	38
7	Bibliography	39

1 Introduction

The OCR Word Search Solver project is part of the EPITA semester S3 curriculum and aims to develop a fully functional program capable of automatically solving a word search puzzle from an image. This project sits at the intersection of multiple fields of computer science, including image processing, optical character recognition (OCR), machine learning, algorithm design, and low-level programming in C. Its goal is to provide a comprehensive system capable of processing an image of a word search, identifying its structure, recognizing the characters it contains, reconstructing the puzzle logically, and finding all hidden words efficiently.

Since the initial phase, significant progress has been made, and the project has now reached a level of completion suitable for the final defense. All the major components required for a fully functional application have been implemented and integrated. The preprocessing module has been completed and now includes automatic rotation, allowing the system to correctly orient any input image without manual intervention.

The neural network component is now fully functional. It has been trained to recognize letters both in the grid and in the list of words, achieving reliable accuracy in letter classification. The network supports both training and inference, enabling not only recognition of letters in new images but also the potential to refine its performance with additional data. This neural network is central to the system, as it bridges the gap between raw image data and structured information that can be used for puzzle solving.

Following recognition, the grid and word list reconstruction modules allow the system to accurately map the puzzle layout and extract the complete list of words to search. The reconstruction is performed automatically, ensuring that the solver receives a correct representation of the puzzle structure regardless of image quality or orientation.

Beyond the technical implementation, this project emphasizes software engineering principles and teamwork. Each member of the team contributed to specific aspects of the project, including preprocessing, neural network training, puzzle reconstruction, algorithmic solving, and GUI integration. Effective collaboration, project planning, and version control through Git were essential to ensure that all modules were developed in parallel and integrated smoothly. The final version represents not only the culmination of technical work but also the successful coordination of a multidisciplinary team working under real-world constraints.

This final defense, therefore, highlights both the technical maturity of the OCR Word Search Solver and the methodological rigor of the development process. From initial image loading and preprocessing to letter recognition, puzzle reconstruction, automatic solving, and visual presentation, the project demonstrates a complete and fully integrated solution capable of addressing the challenges posed by word search puzzles from images. It serves as a comprehensive example of how multiple areas of computer science can be combined to create an efficient, user-friendly, and robust software application.

2 Task Distribution

The development of our OCR-based word search solver application was divided among the different members of the team to ensure a balanced workload and efficient collaboration. Each member focused on a specific part of the project while maintaining a coherent overall structure and design.

- Océane Delogé was responsible for the design and implementation of the solver module, which handles the automatic resolution of word search grids. She developed the algorithm that detects and matches words within the grid based on their position and orientation. Her work provided the core functionality of the project, ensuring that the program could effectively locate and highlight all words once the image was processed and recognized.
- Floralie Niort worked on the image loading system, the creation of the graphical window, and the manual rotation feature. Her role was essential in allowing users to import images in standard formats and adjust their orientation to prepare them for further OCR processing. She designed and implemented the necessary image manipulation tools and user interface elements to make this step smooth and accessible.
- Aliya Aparicio was in charge of the auto-rotation and saving of images, ensuring that any loaded image could be automatically aligned and stored for further processing. She also oversaw the visual design and user interface, carefully defining the color palette, button styles, layout, and graphical hierarchy. By combining these responsibilities, she made sure that the technical features, like image rotation and grid solving, were seamlessly integrated into a clean, intuitive, and visually appealing interface, allowing users to interact with the application easily and effectively.
- Arthur Brune handled the integration and technical coordination of the project. His responsibilities included connecting the different modules developed by the team such as the solver, image processing, and graphical interface to ensure they worked seamlessly together. He also managed the implementation of additional core features, such as the automatic detection of the grid and word list, the segmentation of the image into individual letter tiles, and the preparation of data for neural network training. Arthur played a crucial role in maintaining the coherence and stability of the entire program, debugging complex interactions between modules, and ensuring that the final application functioned reliably from image import to result display.

This distribution of tasks allowed each team member to contribute meaningfully according to their strengths while maintaining a collaborative approach. Regular discussions, shared testing sessions, and coordinated integration ensured that the project advanced steadily toward its final objective: delivering a fully functional, visually polished, and intelligent OCR-based word search solver.

3 Objectives and Project Progress

This section outlines the main objectives, task planning, and the comparison between expected and actual progress of the project. It provides an overview of how the work was structured, distributed, and executed within the team.

At the start of the project, the group defined clear technical and organizational goals to guide development and ensure steady progress toward the final deliverable. These objectives were divided into smaller, well-defined tasks covering the main components of the project such as image preprocessing, grid detection, character recognition, neural network training, and word search solving.

A timeline was established to plan milestones for each phase and track progress up to the first and final defenses. The following list presents all tasks planned for the entire project, along with their purpose and expected order of completion.

Complete Project Task List:

1. **Loading an image:** Import the image from a file and ensure it is loaded correctly and readable.
2. **Interface:** Create a visually engaging and easy-to-use interface for the user.
3. **Removing colors:** Convert the image to grayscale to simplify analysis and remove unnecessary colors.
4. **Pretreatment:** Reduce image noise with a filter, correct rotation if needed, and normalize the size.
5. **Locating the grid:** Identify the edges of the grid and extract the rectangular area containing the letters.
6. **Locating the word list:** Locate the area containing the word list and isolate it for text recognition.
7. **Locating the cells of the grid:** Divide the grid into individual cells and prepare each cell for letter analysis.
8. **Basic neural network:** Create and train a neural network to recognize letters from images and test it on a small dataset.
9. **Character recognition from images:** Recognize the letters in each grid cell and in the word list while handling recognition errors or ambiguities.
10. **Rebuilding the grid as an array of characters:** Transform the recognized grid into a 2D array of characters.
11. **Rebuilding the word list as an array of strings:** Create an array of strings for the word list and correct any errors from recognition.

12. **Solving the grid:** Search for all words in every direction within the grid, mark their positions, and handle shared or overlapping letters.
13. **Displaying the result:** Visually display which words from the list have been detected or remain.
14. **Saving the result:** Export the image with highlighted words and save the grid and word list.

Progress:

	First Defense (Week of November 3)	Last Defense (Week of December 15)
Loading an image	100%	100%
Interface	90%	100%
Removing colors	100%	100%
Pretreatment	100%	100%
Locating the grid	100%	100%
Locating the word list	100%	100%
Locating the cells of the grid	100%	100%
Basic neural network	100%	100%
Character recognition from images: the letters in the cells and the letters that make up the word	10%	100%
Rebuilding the grid as an array of characters	0%	100%
Rebuilding the word list as an array of strings	0%	100%
Solving the grid	100%	100%
Displaying the result	0%	100%
Saving the result	0%	100%

4 Development Methodology and Technical Justifications

4.1 Loading an image and interface design

For this last presentation, we developed the entire graphical user interface (GUI) module using the GTK3 library, with a strong emphasis on ergonomics, clarity, and visual consistency. The interface was designed as the central entry point of the project, orchestrating all preprocessing and analysis steps required to solve a word-search grid image. Particular attention was given to usability in a pedagogical context, as well as to the constraints of rapid development within the EPITA curriculum.

Upon launch, the user is presented with a fully styled main window that clearly separates controls from the image display area. The layout follows a vertical structure composed of a header section containing the main interaction buttons, a central drawing area dedicated to image visualization, and a status zone used to display feedback and processing messages. This organization guides the user smoothly from image loading to solver execution while minimizing cognitive load.

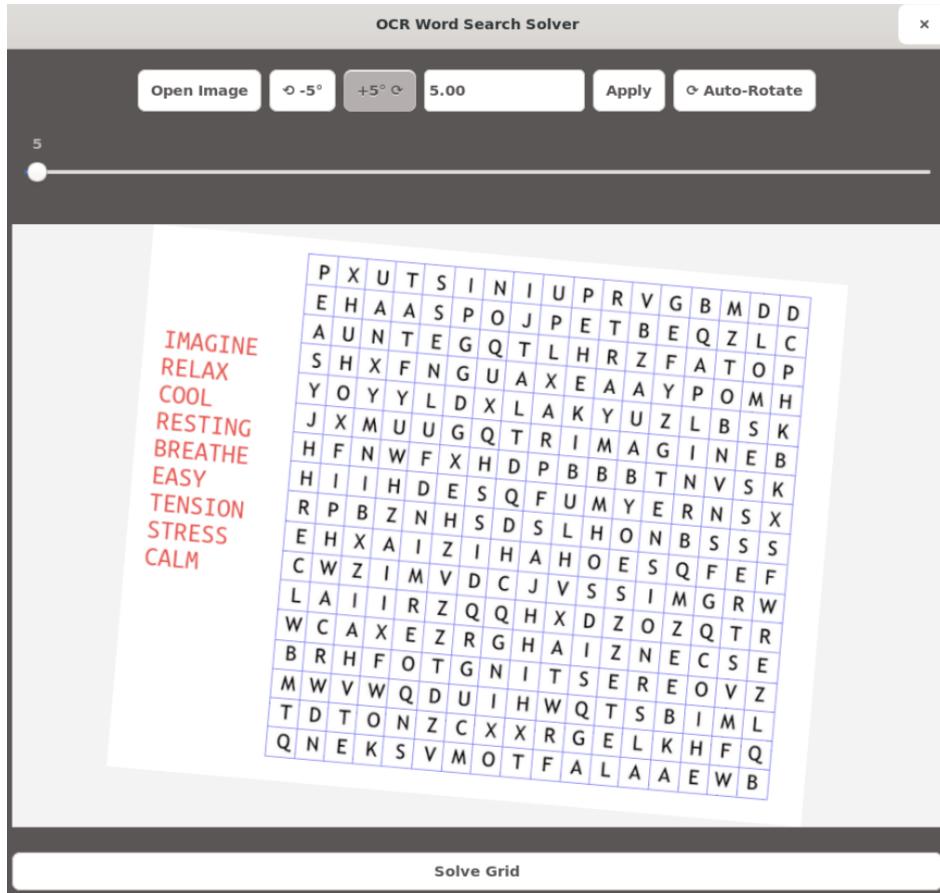


Image of the interface

Image Loading and Visualization

When the “Open Image” button is pressed, a file chooser dialog appears, filtering supported formats such as PNG, JPG, JPEG, and BMP to prevent incompatible inputs. Once selected, the image is loaded into a GdkPixbuf and displayed at the center of the interface.

The image is rendered inside a dedicated GtkDrawingArea using the Cairo graphics library. This approach allows precise control over graphical transformations while ensuring that the image is automatically scaled to fit the available space and that its original aspect ratio is preserved. Regardless of resolution or orientation, the rendering pipeline computes the bounding box of the rotated image and applies an appropriate scaling factor, preventing distortion or clipping.

Simultaneously, a working copy of the image is saved as `./output/image.bmp`, ensuring that all preprocessing and solver modules operate on the exact image currently displayed to the user.

Image Manipulation and Preprocessing Controls

To prepare the image for OCR and grid extraction, the interface provides multiple interactive manipulation tools. Manual rotation is available through three complementary mechanisms:

- Fixed-step rotation buttons
- A horizontal slider allowing fine-grained rotation between 0° and 360° ,
- A text entry field for manual angle input, applied explicitly by the user.

All these widgets are connected to GTK3 callback functions. Each interaction updates a shared AppData structure storing the application state and triggers an immediate redraw of the image. This design provides real-time visual feedback, allowing precise alignment of the grid without trial and error.

In addition to manual controls, an Auto-Rotate feature automatically detects and corrects the orientation of the grid. This process iteratively applies rotation correction and cleans rotation artifacts such as black borders, replacing them with a white background. Once completed, rotation controls are reset to reflect the internally corrected orientation.

A Denoise button is also integrated into the interface. This operation applies a median filter to reduce visual noise while preserving grid lines and letter strokes. After denoising, the processed image is reloaded into the viewer, ensuring visual consistency with the image used for subsequent processing.

Solver Execution and Result Interface

- **Save**, which stores the solved grid using a default or previously selected path,
- **Save As**, which opens a file chooser dialog allowing the user to specify a custom filename and destination.

Throughout the interaction, the interface provides continuous feedback through a dedicated message label, informing the user of ongoing operations, errors, or successful preprocessing steps. When the “Solve Grid” button is pressed, the application launches the complete extraction, OCR, and solving pipeline using the most recently processed image.

Once the solving process is complete, the workflow transitions to a second dedicated interface focused on result visualization. This new window displays the solved word-search grid, where detected letters and identified words are clearly presented. By separating preprocessing from result interpretation, the application maintains a clean and readable main interface while offering a focused view of the final output.

To support result persistence and reuse, this solved-grid interface includes two export options:

- Save, which stores the solved grid using a default or previously selected path
- Save As, which opens a file chooser dialog allowing the user to specify a custom filename and destination.

This two-stage interface design ensures a complete end-to-end workflow, from raw image loading and preprocessing to solution visualization and export. It also reinforces a clear conceptual separation between image preparation and analytical results, improving both usability and maintainability.

Challenges Encountered and Lessons Learned

Developing a responsive and robust graphical interface in GTK3 using C proved to be both challenging and instructive. The complexity of the GTK3 framework, combined with explicit memory management, required careful handling of widgets, signals, and graphical resources. Each feature, from real-time rotation to automatic preprocessing and solver integration, demanded precise coordination between the GUI and the underlying processing logic.

Debugging was particularly demanding due to segmentation faults caused by improper pointer usage or incorrect object lifetimes. These issues often occurred only under specific interaction sequences, making them difficult to reproduce. The lack of dedicated automated testing tools for GTK3 led us to rely heavily on manual testing, console logging, and systematic inspection of callback execution paths.

Despite these challenges, this development experience significantly strengthened our understanding of event-driven programming, memory management in C, and the integration of real-time graphical interfaces with complex backend pipelines. The resulting interface is modular, extensible, and well-suited for both educational and practical use within the scope of the project.

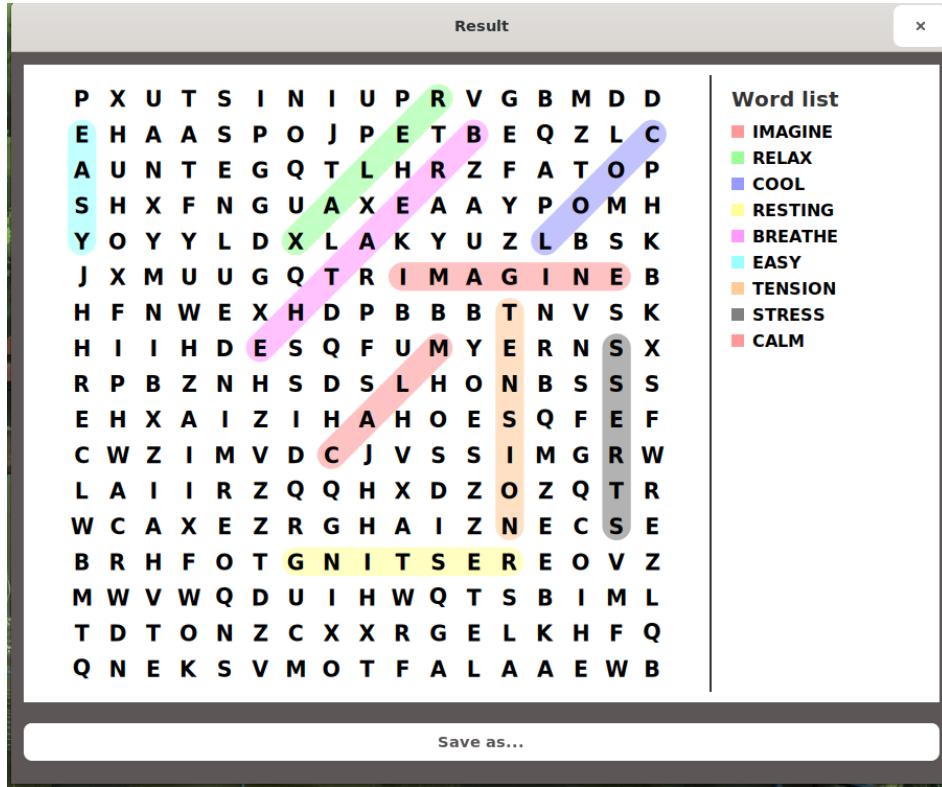


Image of the second interface

4.2 Color Removal and Image Binarization

The preprocessing step is a foundational stage in the computer vision pipeline of our word-search solver project, as it transforms the raw input into a format suited to robust grid, word, and letter extraction. This discussion describes the processing pipeline from image loading to optimized binary output, focusing on design choices meant to maximize robustness for diverse user photos or scans.

Image loading and conversion:

Raw user images are loaded using the SDL2 library. Since input image formats and pixel layouts vary, every image is immediately converted to a standardized pixel format (ARGB8888) for predictable access and performance.

Grayscale conversion:

All processing is performed in grayscale, a standard simplification which discards color data but preserves the text's structural information. Each RGB pixel value is mapped to a grayscale level using an integer approximation of the luma formula:

$$L = \frac{77 \times R + 150 \times G + 29 \times B}{256}$$

This is implemented as:

```
Y = (77 * R + 150 * G + 29 * B) >> 8
```

This fast operation avoids floating-point arithmetic and delivers accuracy suitable for thresholding.

Global Thresholding with the ISODATA/Riddler-Calvard Method:

The core of binarization is threshold selection: determining which pixels are considered “ink” and which are “background / paper”. After testing several strategies, we chose the Riddler-Calvard (ISODATA) global method over Otsu’s method due to implementation simplicity and robust performance on moderately noisy backgrounds.

- The algorithm starts with the mean gray value as an initial guess.
- Pixels are split into two groups (\leq threshold, $>$ threshold), and the threshold is iteratively updated as the mean of the two groups until convergence.

This algorithm requires only integer operations and a single pass over the image per iteration, making it efficient and practical for C/SDL2 environments.

Binarization and Output:

After thresholding, pixel values are replaced by pure black or white (0 or 255). The binarized image is written to a bitmap with the same dimensions as the input, preserving spatial relationships for later processing by extraction modules.

Noise Reduction via Median Filtering

Before binarization, a dedicated denoising step is applied to improve robustness against acquisition noise commonly found in user-provided photographs, such as salt-and-pepper noise, compression artifacts, or uneven lighting. This step is implemented in the denoise.c module using the SDL2 library.

To ensure consistent pixel access, the input surface is first converted to the ARGB8888 format if necessary. This guarantees a fixed 32-bit layout and simplifies channel extraction during processing.

The denoising algorithm is based on a 3×3 median filter, a non-linear filter well suited for text-heavy images. Unlike mean or Gaussian filters, the median filter preserves sharp edges while effectively removing isolated noisy pixels, which is critical for maintaining letter strokes and grid lines.

For each interior pixel, the algorithm:

- Collects the RGB values of the 3×3 neighborhood surrounding the pixel.
- Separates the red, green, and blue channels into three small arrays.
- Sorts each array and selects the median value.
- Reconstructs the pixel using the median of each channel independently.

Border pixels are copied unchanged to avoid out-of-bounds memory access and unnecessary distortion near image edges.

This implementation relies exclusively on integer operations and simple sorting logic, making it efficient and portable within a C/SDL2 environment. The output image retains the same dimensions as the input and serves as a cleaner foundation for subsequent grayscale conversion and thresholding.

Integration with Binarization

The denoised image is then passed to the grayscale conversion and global thresholding stages. By removing high-frequency noise prior to binarization, the median filter significantly reduces the risk of false foreground pixels and broken character strokes in the binary image. This improves the stability of later stages such as grid detection, letter segmentation, and OCR recognition.

Overall, this preprocessing pipeline—format normalization, denoising, grayscale conversion, and ISODATA thresholding—produces a high-contrast binary image that balances noise suppression with structural preservation, which is essential for reliable word-search puzzle solving.

4.3 Finding and Slicing Grid, Words, and Letters

This stage of our project transforms a binarized image into structured data objects: the puzzle grid, the word list, and ultimately the individual character or word images needed for OCR. Following the preprocessing phase—including denoising, grayscale conversion, and global thresholding—the image is now a high-contrast binary representation where ink appears black and background is white. At this stage, our goal is to extract semantic structure from the image.

4.3.1 Grid Detection and Extraction

Ink Projection Profiles: To localize the grid (the area containing puzzle characters), we compute *ink density* projections along both horizontal and vertical axes. These projections are histograms of black pixel counts:

- **Row profile:** For each image row, count the number of black pixels. Rows with high counts typically correspond to horizontal grid lines or rows with letters.
- **Column profile:** For each column, count the black pixels. Columns with high counts likely indicate vertical grid lines or columns containing letters.

These profiles serve as a robust representation of the image structure. Even in cases of uneven lighting, shadows, or minor scanning artifacts, the aggregation over rows and columns smooths out noise and emphasizes structured ink regions. **Pseudo-code:**

```
for y = 0 to image_height-1:  
    row_count = 0  
    for x = 0 to image_width-1:  
        if pixel(x, y) is black:  
            row_count += 1  
        row_profile[y] = row_count  
    for x = 0 to image_width-1:  
        col_count = 0  
        for y = 0 to image_height-1:  
            if pixel(x, y) is black:  
                col_count += 1  
            col_profile[x] = col_count
```

Thresholding and Line Detection: Once the projections are computed, thresholds are applied to distinguish grid lines from whitespace. Typically, thresholds are set to 50

- Any row or column with black pixel count exceeding the threshold is marked as a potential line.
- Consecutive marked rows/columns are grouped into bands, defining candidate horizontal or vertical lines.

This produces an initial set of grid line positions. The median gap between consecutive lines is computed to estimate the expected cell size. Lines that are outliers (too close or too far) are further analyzed:

- Narrow bands might indicate broken grid lines.
- Wide bands could indicate merged lines or two touching letters.

Pseudo-code for band detection:

```

bands = []
in_band = False
for i in range(profile_length):
    if profile[i] >= threshold:
        if not in_band:
            band_start = i
            in_band = True
        else:
            if in_band:
                band_end = i - 1
                bands.append((band_start, band_end))
            in_band = False
    if in_band:
        bands.append((band_start, profile_length-1))

```

Handling Skewed or Rotated Images: Real-world images often have slight rotations. To mitigate this, the algorithm computes the variance of row and column profiles for incremental rotations:

- Rotate image by small angles within a predefined range (e.g., -5° to $+5^\circ$).
- Compute horizontal and vertical projections at each angle.
- Select the rotation angle that maximizes the variance of the projections (which sharpens lines).

After rotation, a new set of projections is computed, ensuring accurate grid detection even for imperfect photographs. **Grid Cropping:** Using the refined grid line positions:

- Compute minimal bounding rectangle enclosing all intersecting lines.
- Apply a small padding to prevent cutting off parts of letters at the grid edges.
- Crop the rectangle as a new SDL surface for subsequent processing.

Memory consideration: each cropped surface is dynamically allocated and freed after use to avoid leaks, particularly when handling high-resolution images.

Grid Slicing – Extracting Cells: Each grid cell is defined by pairs of adjacent horizontal and vertical lines:

- Iterate through all horizontal line pairs r_i, r_{i+1} and vertical line pairs c_j, c_{j+1} .
- Define bounding box for cell as:

$$[(c_j + \text{trim}), (r_i + \text{trim}), (c_{j+1} - c_j - 2 \cdot \text{trim}), (r_{i+1} - r_i - 2 \cdot \text{trim})]$$

x_0 y_0 w h

- Extract each cell as a separate image surface.
- Save each cell as `c_XX_YY.bmp` for OCR.

Edge-case handling:

- If trimming produces zero-width or zero-height boxes, skip the cell.
- For uneven grids, adjust trimming based on median cell size.

4.3.2 Word List Region Extraction

After grid extraction, the word list must be localized. The algorithm examines each of the four margins:

- Define a padding strip along the left, right, top, and bottom of the grid.
- Count black pixels in each strip.
- Select the margin with the maximum black pixel count.

This method accommodates puzzles where word lists are printed in varying orientations or positions. **Memory and performance considerations:** Each margin's pixel count is computed in a single pass, avoiding unnecessary duplication of surfaces.

4.3.3 Slicing Words from the Word List

Text Row Detection: Within the selected word list region:

- Compute a horizontal projection profile.
- Identify consecutive low-ink rows as gaps separating words.
- Each high-ink block corresponds to a candidate word row.

To handle complex layouts:

- Large blocks are recursively subdivided using increasing gap sensitivity.
- Words spanning multiple lines are treated as a single entity until gaps are detected.

Word Slicing:

Within each detected row:

- Compute vertical projection profile.
- Detect consecutive zero-ink columns as word boundaries.
- Extract each word image, adding a small margin to preserve strokes.

Edge cases:

- Words with touching letters may be split based on median column width.
- Tiny isolated black pixels are ignored to prevent false word splits.

4.3.4 Letter Segmentation from Words

For advanced OCR or dataset creation, each word can be segmented into letters:

- Compute vertical projection profile of the word image.
- Identify columns containing ink and group consecutive ink columns as candidate letters.
- Apply a minimal horizontal margin for each letter.

Pseudo-code:

```
letters = []
start = -1
for x = 0 to word_width-1:
    if column_has_ink(x):
        if start == -1: start = x
        else:
            if start != -1:
                letters.append((start, x-1))
            start = -1
        if start != -1:
            letters.append((start, word_width-1))
```

Considerations:

- Very narrow letters (e.g., 'I') are preserved using minimum width thresholds.
- Merge adjacent letters if the gap is smaller than a fraction of median letter width.

Memory and Performance Optimization:

- Intermediate surfaces for each slice (grid, word, letter) are freed promptly after saving.
- Projection profiles are computed using integer arithmetic to maximize speed.
- Recursion in word/letter segmentation is bounded to prevent stack overflows on large images.

Summary: By the end of this stage:

1. The puzzle grid is cropped and cleaned.
2. Each grid cell is extracted and stored as an image.
3. The word list region is localized and segmented into words.
4. Optional per-letter segmentation is performed for advanced OCR applications.

This comprehensive slicing pipeline transforms the binary image into structured, analyzable data. By combining projection profiles, thresholding, recursive segmentation, and memory-safe surface handling, the method is robust to noise, skew, variable layouts, and font irregularities.

4.3.5 Challenges Encountered

The extraction and slicing of the word-search grid from a binarized image proved to be one of the most technically demanding stages of the project. Despite the apparent simplicity of dividing a grid into rows and columns, numerous challenges arose from the variability in image quality, grid layout, OCR accuracy, and preprocessing results. This section details the most significant difficulties encountered, the strategies applied to mitigate them, and the lessons learned in the process.

1. Variability in Input Images:

One of the first challenges was dealing with the diversity of input images. Users could provide high-resolution scans, smartphone photographs, or screenshots, each with different lighting conditions, rotation angles, and noise characteristics. Even after preprocessing—grayscale conversion, denoising, and binarization—some images contained residual artifacts, such as small ink smudges, irregular pixel clusters, or partial shadows along the edges of the puzzle. These artifacts often produced false positives in the row and column ink projection profiles, leading to misidentified grid boundaries or extra “phantom” lines.

To address this, the team implemented a combination of adaptive thresholding, median filtering, and post-processing heuristics. Projection profiles were smoothed by ignoring isolated peaks smaller than a few pixels and by enforcing a minimum gap between

consecutive detected segments. This significantly reduced the impact of noise while preserving legitimate grid lines.

2. Detecting Grid Lines in Non-Uniform Grids:

A second challenge involved the detection of grid lines when puzzles were not perfectly uniform. Some grids exhibited slightly uneven cell spacing due to scanning or printing errors, while others had variable line thickness. Initial algorithms using fixed thresholds for row and column projection counts often failed in these scenarios, either merging two adjacent cells into one or splitting a single cell into multiple segments.

To improve robustness, the approach was enhanced by computing the median cell size and allowing a tolerance range for each detected segment. Segments larger than 1.5 times the median were recursively split, while smaller segments were verified against their neighbors to avoid spurious cuts. This “smart cut” strategy required careful tuning: overly aggressive splitting could break legitimate cells, whereas conservative parameters left merged letters unresolved.

3. Handling Skewed or Rotated Grids:

Many real-world images were slightly rotated or skewed. Even minor rotations caused significant misalignment in the projection profiles: vertical lines no longer corresponded neatly to columns, and horizontal lines no longer aligned with rows. Early versions of the algorithm, which assumed perfectly axis-aligned grids, frequently produced jagged or unevenly sized cells.

The solution combined two strategies. First, a preprocessing auto-rotation step estimated the dominant grid orientation and corrected small rotation angles. Second, when minor residual skew persisted, the slicing algorithm incorporated a tolerance margin and allowed overlapping segments to ensure that no letter was lost. This combination greatly improved the accuracy of both cell boundaries and subsequent OCR recognition.

4. Determining Thresholds for Projection Profiles:

Choosing appropriate thresholds for identifying “inked” rows and columns was particularly challenging. Fixed thresholds often failed because the number of black pixels varied with cell size, font thickness, and image resolution. A threshold too low captured noise; a threshold too high missed faint lines or letters.

To resolve this, thresholds were calculated relative to the image dimensions (typically a percentage of width or height) and adjusted dynamically using the histogram of pixel densities. Additionally, a minimum and maximum segment size check was applied to prevent the algorithm from treating very thin noise clusters as cells or merging abnormally large blocks that likely contained multiple letters.

5. Memory Management and Performance:

Working with large images introduced both memory and performance challenges. Each projection profile, intermediate surface, and final cell image required careful allocation and deallocation to avoid leaks and excessive memory consumption. In early prototypes, repeated allocation of temporary structures for each row or column caused performance bottlenecks, especially when processing high-resolution images with hundreds of cells.

The solution involved preallocating arrays for raw ranges, final ranges, and projection profiles, along with reusing surfaces for temporary copies. By ensuring that each allocation had a corresponding deallocation and by limiting the number of temporary objects created per iteration, the system became both faster and more memory-efficient.

6. Handling Merged or Split Letters:

Another significant challenge arose when two letters were touching or when thin grid lines caused letters to appear disconnected. Simple thresholding often merged adjacent letters into a single cell or split a single letter across multiple cells. This was particularly problematic for cursive-style fonts or stylized puzzle layouts.

To mitigate this, the algorithm included a recursive splitting mechanism based on segment width relative to the median cell size. Each oversized segment was subdivided into smaller candidate cells, and the resulting bounding boxes were verified to ensure each contained significant ink. This approach significantly increased letter isolation accuracy while minimizing errors caused by merged or split characters.

7. Integration with Word List Extraction:

After grid detection, the system needed to locate the word list along one of the image margins. This required combining the results of the grid extraction with analysis of surrounding margins. Challenges included distinguishing the word list from other nearby text or graphical elements, and handling cases where the word list orientation differed from the grid (e.g., vertical lists on the side). The solution was to compute black-pixel counts along all four margins and select the one with the maximum density, providing a robust and flexible approach that worked across multiple puzzle layouts.

8. Debugging and Validation:

Debugging grid extraction proved difficult because errors were often subtle and dependent on specific images. Misaligned cells or incorrect boundaries were not always obvious from visual inspection. To overcome this, the team implemented detailed logging of projection profiles, detected ranges, median cell sizes, and split decisions. Overlay

visualizations were also generated in test images to show proposed cuts, enabling quick identification of problematic thresholds or segments.

9. Lessons Learned:

Through these challenges, the team gained significant insight into the complexity of reliable grid extraction. Key lessons included the importance of dynamic thresholds, robust handling of edge cases, careful memory management, and the utility of intermediate visual debugging. These insights informed not only the grid extraction module but also subsequent stages such as cell OCR, word slicing, and result visualization, ensuring the system's overall reliability and accuracy.

In conclusion, the grid extraction stage required careful consideration of image variability, noise, rotation, segmentation thresholds, memory, and integration with downstream processes. By implementing median-based smart cuts, adaptive thresholds, recursive splitting, and comprehensive debugging strategies, the system achieves robust and accurate grid detection even under challenging real-world conditions. This work lays the foundation for the precise extraction of individual cells and words, which is critical for reliable OCR and solver performance.

4.4 Solver

The solver was divided into several distinct functions, each handling a specific task. This separation makes the code easier to read, debug, and extend. Functions can be modified independently without affecting the rest of the program, redundant code is minimized, and the overall logic of the solver becomes clearer. It also allows for efficient testing and optimization of individual components.

4.4.1 Function Overview

1. `read_grid`

The solution begins with the `read_grid` function, which is responsible for loading the grid data from a text file. It reads each line, strips newline characters, and populates a two-dimensional array representing the grid. Additionally, it records the number of rows and columns, ensuring that grid dimensions are always known for subsequent operations. This setup is essential for maintaining consistency and reliability throughout the word search procedure.

2. `check_word_in_direction`

To perform the search, the solver uses the `check_word_in_direction` function. This function checks whether a given word exists starting at a specific grid location, moving in one of eight possible directions—horizontal, vertical, and both diagonals. It computes coordinates for consecutive letters based on a direction vector, verifies array bounds at each step, and compares characters in the grid with the target word. An early exit occurs if a boundary is exceeded or a character mismatch is detected. This approach improves runtime efficiency and ensures robustness for grids of varying sizes.

3. `find_word`

The central search logic is handled by the `find_word` function. It iterates through every cell of the grid, and upon finding a match for the first letter of the target word, it applies directional search across all orientations. If a search is successful, the start and end coordinates of the word are calculated and returned according to project specifications. Using direction vectors allows exhaustive search without duplicating code, ensuring accurate detection in all directions.

4. `main`

The `main` function manages program flow, including command line argument validation, grid initialization, and result reporting. It converts the input word to uppercase to ensure case-insensitive searches, and then invokes `find_word`, presenting results either as a coordinate pair for successful matches or as a “Not Found” message if the search is unsuccessful.

4.4.2 Challenges Encountered

During development, several challenges arose that required careful attention to ensure the solver was robust and efficient. Reliable boundary checking was critical to prevent out-of-bounds memory access and segmentation faults, particularly when the search traversed reverse directions and diagonals. Each potential move required verifying that the computed coordinates remained within the valid grid limits.

Implementing the eight-direction search presented a logical challenge. It was necessary to create a system that could exhaustively check all directions without duplicating code. Direction vectors were used to standardize the process, reducing redundancy and minimizing indexing errors.

Handling input words and grid characters also required careful attention. Input normalization, such as converting all letters to uppercase, was necessary to guarantee case-insensitive matching. Additionally, proper buffer management was important to avoid memory errors during comparisons.

Robust handling of file input and output was another essential consideration. The program had to detect and report failures immediately, such as missing or improperly formatted files. Ensuring that the grid was correctly loaded before initiating any search operations was a priority for reliable execution.

While dividing the solver into multiple functions improved maintainability, it also required disciplined design to ensure that each function had a clear responsibility. Carefully passing parameters and managing the flow of information between functions was essential to avoid introducing bugs.

Finally, although correctness was the primary goal, performance could not be ignored. Efficient iteration through the grid and early exit conditions in the directional search were implemented to reduce unnecessary computations, especially for larger grids or longer word lists.

4.5 Solved grid

Once the solver identifies the location and orientation of each word inside the puzzle grid, the next step of the system is to present a clear and readable visual representation of the final solution to the user. While the console-based prototype prints coloured characters using ANSI escape sequences, the graphical version integrates this functionality into the GTK-based GUI. The goal of this module is to open a dedicated “Result Window” that displays the solved grid with each discovered word highlighted in a distinct colour.

This part of the pipeline is essential because it provides intuitive feedback to the user. Instead of scanning solver logs or interpreting numerical coordinates, the user directly sees the extracted grid with visual cues indicating where each word appears. In addition, displaying the solved grid in a separate window makes it easy for the user to compare the original image with the solver’s interpretation. The following subsections detail the logic and implementation strategy behind loading the solver’s output, mapping word coordinates to grid cells, assigning colours, and finally rendering the completed grid in a GTK window.

4.5.1 Data acquisition and preparation

The solved-grid display module depends on several pieces of information produced earlier in the pipeline. The first is the textual representation of the puzzle grid generated by the OCR component and stored in the file `output/words.txt`. This file contains the letters exactly as they were recognized in the image, arranged in rows and columns that match the layout of the original puzzle. The program loads this file into memory using the same mechanism employed by the solver, populating a two-dimensional array of characters along with global variables that store the grid’s dimensions.

In parallel, the word list extracted by OCR is loaded from `output/words.txt`. This file contains each word that should be searched within the grid. During the solving phase, each of these words is passed to the function `find_word()`, which returns a pair of coordinates indicating the first and last letter of the word in the grid. These coordinates form the fundamental data source for the display module, because they provide the information needed to reconstruct the entire sequence of grid cells that must be highlighted. Once both the grid and the word list are loaded, the display module prepares internal structures to store the final rendering information. Each found word is represented by a structure containing the start and end coordinates as well as an associated colour index. These structures are stored temporarily and later used by the rendering code to determine which cells in the grid should be colourised.

4.5.2 Reconstructing Word Paths from Coordinates

The solver returns only the two endpoints of each word, but this is sufficient to reconstruct the full path of cells that belong to the word. Because all words in a word-search puzzle are aligned along straight lines, the traversal can be defined by computing directional increments for the x- and y-coordinates. Each increment takes a value of -1, 0 or 1 depending on whether the word travels left, right or not at all along the given axis. By starting at the initial coordinate and repeatedly adding these increments, the system visits every cell from the beginning of the word to its final letter. This method supports all eight possible orientations—horizontal, vertical and diagonal—without any additional complexity.

This reconstruction step is essential because it translates abstract solver output into concrete cell-level information that can be used for highlighting. As the program iterates through the coordinates of each word, it marks each visited cell and associates it with a specific colour index. The result is a complete mapping between the grid's spatial layout and the solver's logical representation of word positions.

4.5.3 Assigning Colours to Identified Words

Colour assignment is carried out using a predefined palette of ANSI background colours. These colours are chosen for their readability and distinctiveness; they include pastel shades of orange, green, yellow, blue, magenta, cyan and pink. The system assigns colours sequentially to words in the order they are found, and the palette cycles if the number of words exceeds the number of available colours. This ensures that the program remains robust and functional even when dealing with particularly large puzzles.

The colour index assigned to each word is stored along with its coordinate data. This makes the system flexible because the visual appearance is decoupled from the underlying coordinate logic. The rendering function simply checks which colour index belongs to the word occupying a given cell and prints or draws the appropriate background colour. Special attention is given to overlapping words, a common feature in word-search puzzles. In cases where two words share one or more letters, the program gives priority to the word that was processed last, ensuring determinism and visual clarity. This approach prevents ambiguous or inconsistent colouring, and it guarantees that the result remains stable across multiple executions of the program.

4.5.4 Building the Highlight Map for Rendering

Once all word paths have been reconstructed and colour indices assigned, the system produces a complete highlight map of the grid. For each cell in the grid, the program checks whether the cell belongs to any word by comparing its coordinates with the positions generated during the traversal step. If the cell is part of a word, it is associated with the corresponding colour index; if not, it remains uncoloured.

The mapping step does not directly depend on either the solver or the OCR process. Instead, it consolidates their results into a form that is convenient for display. The highlight map is constructed entirely in memory before any rendering begins, allowing the same data to be used both for console output and for the graphical interface. This design also allows the display module to operate independently from the rest of the system: the highlight map can be examined, modified or extended without altering the solver logic or OCR pipeline.

4.5.5 Console-Based Rendering Prototype

Before integrating the result into the GTK interface, a console-based prototype was developed to verify the correctness of the highlighting logic. This prototype prints the grid directly in the terminal, surrounding each letter belonging to a found word with the appropriate ANSI background colour escape codes. Although simple, this mechanism proved invaluable during development because it made it possible to check that diagonal words, reverse orientations and overlapping highlights were handled correctly.

The console version also revealed potential issues such as off-by-one errors, incorrect coordinate interpretation or misaligned OCR output. Because the console output is lightweight and does not depend on any graphical toolkit, it was frequently used as a debugging tool throughout the project. Only once the logic was validated thoroughly was the graphical version implemented on top of this foundation.

4.5.6 Integration into the GTK Graphical User Interface

The final user-facing version of the solved grid is displayed inside a dedicated GTK result window. This window is launched through the function `show_result_window()`, which is triggered when the solving process finishes.

In the graphical interface, the grid is rendered either using GTK label widgets arranged in a grid layout or by drawing onto a Cairo surface. In either case, the rendering logic makes use of the highlight map prepared earlier. Each letter is drawn with or without a coloured background depending on whether it belongs to a found word, and the colour chosen matches the palette defined during the console prototype. The graphical rendering therefore reproduces exactly the same highlight pattern as the console version but presents it in a more appealing and modern form suited for end users.

4.5.7 Challenges Encountered

Developing the solved grid display module revealed a number of challenges, both conceptual and technical, that required careful consideration and iterative solutions. Although the underlying logic of highlighting words in a word-search puzzle appears straightforward, translating the solver's abstract coordinates into a robust, readable graphical output proved to be considerably more complex than anticipated. Several categories of challenges emerged during the implementation process.

1. Data Consistency and Synchronization:

One of the earliest difficulties arose from ensuring consistency between the solver's output, the OCR-derived grid, and the word list. Since the display module relies on precise mapping between the textual grid and the visual grid, any discrepancy in indexing or dimensions could result in misaligned highlights. For example, a mismatch between the number of rows and columns in the OCR output and the solver's internal representation would cause a shift in highlighted words, potentially marking incorrect letters. To address this, additional verification routines were implemented to cross-check grid dimensions and validate that all coordinates returned by `find_word()` were within the bounds of the loaded grid. These validation steps included both runtime assertions and conditional logging to catch anomalies during development.

2. Reconstructing Word Paths:

Although the solver only returns the start and end coordinates of each word, reconstructing the complete path required careful handling of directional increments. Initially, off-by-one errors frequently occurred, particularly for diagonal words and reverse orientations. Each cell along a word's path must be visited exactly once, with no duplication and no omission. This led to the implementation of a generalized algorithm that computes increments along both axes as -1, 0, or 1, depending on the relative positions of start and end coordinates. Extensive testing was required to ensure that all eight possible word orientations were supported, including corner cases where words spanned from the bottom-right to top-left or diagonally across the grid. Debugging was further complicated by subtle inconsistencies in zero-based indexing between the OCR output array and solver calculations, which required meticulous inspection of array bounds and coordinate translation.

3. Colour Assignment and Overlaps:

Assigning distinct colours to each word introduced both aesthetic and technical challenges. A fixed palette of visually distinct colours was selected to enhance readability, but the number of words in some puzzles exceeded the palette size, necessitating a cycling mechanism. More critically, overlapping words required a deterministic strategy

to prevent visual ambiguity. Initially, overlapping cells would sometimes display the colour of the first word processed, resulting in confusing highlights for the user. To solve this, the algorithm was modified to assign the colour of the last processed word to overlapping cells, ensuring a consistent and predictable display. Additionally, careful attention was paid to colour contrast and legibility; background and foreground colours were tested against various grid sizes and letter densities to guarantee that each word remained distinguishable.

4. Highlight Map Construction:

Building a highlight map for the entire grid presented a memory management challenge. Each cell required storage of a colour index or an uncoloured state, resulting in a two-dimensional array that mirrored the grid layout. Initially, attempts to store additional metadata per cell, such as word IDs or multiple colour layers, led to increased memory consumption and slower rendering. The final design simplifies each cell to a single colour index, leveraging the sequential processing of words to ensure correct representation of overlaps. This approach required careful iteration over all word paths to update the highlight map efficiently while avoiding redundant writes and excessive computational overhead.

5. Debugging and Prototype Verification:

A console-based prototype proved essential in verifying the correctness of the highlight logic. Before integrating GTK rendering, the system printed the solved grid with ANSI background colours directly in the terminal. This lightweight testing method allowed rapid identification of issues such as incorrect coordinates, off-by-one errors, or misaligned OCR data. Without this intermediate step, graphical debugging would have been significantly more time-consuming, as errors would only manifest visually. Moreover, the console prototype served as a reliable reference to ensure that the final GTK implementation reproduced the expected colour patterns exactly.

6. Integration with GTK and Cairo:

Rendering the solved grid in a GTK window introduced several additional challenges. Unlike the console, graphical output requires precise positioning of letters and backgrounds, careful handling of widget layouts, and efficient redraw strategies. Initial implementations using individual label widgets for each cell suffered from slow performance and excessive memory use, particularly for large grids. Transitioning to a Cairo-based drawing approach allowed batch rendering of letters and backgrounds directly onto a surface, significantly improving responsiveness. However, this required careful mapping of grid coordinates to pixel positions, scaling considerations for different window sizes, and handling of padding to maintain alignment. Additionally, the program had to ensure that redraws were synchronized with GTK's main loop to prevent flickering or inconsistent rendering during window resizing or updates.

7. Error Handling and Robustness:

Finally, the system needed to gracefully handle edge cases such as missing OCR data, unrecognized letters, or words that could not be found by the solver. In these scenarios, the display module defaults to leaving cells uncoloured or marking them with a placeholder colour, ensuring that the interface remains usable and informative. Extensive logging and exception handling routines were added to notify developers of inconsistencies without disrupting the user experience. This robustness ensures that even partially successful solver runs can be visualized effectively, a critical feature for testing and pedagogical purposes.

In summary, developing the solved grid display module required careful attention to data integrity, coordinate reconstruction, colour management, memory efficiency, graphical rendering, and error handling. Overcoming these challenges not only produced a clear and reliable visual representation of the solved puzzle but also strengthened the team's expertise in integrating computational logic with user interface design, particularly in the context of C and GTK-based graphical applications. The final implementation provides an intuitive, visually consistent, and responsive display that accurately reflects the solver's output and enhances the overall usability of the system.

4.6 Automatic Rotation

Automatic rotation of the word search grid is a key component of the OCR Word Search Solver project. For the solver to function correctly and automatically, the input image must be properly aligned. Without proper orientation, subsequent processing—such as extracting the grid cells or analyzing the letters—would be unreliable or even fail. To address this, we developed a rotation module that determines the dominant orientation of the grid and rotates the image accordingly, while also enabling reconstruction of the grid in its new orientation.

4.6.1 Image Preprocessing

The first step in the rotation process is preprocessing the input image to simplify further analysis. The image is loaded into memory and converted to a format suitable for numerical operations. To reduce the complexity and ensure robustness, the image is initially treated as grayscale. Color information is discarded because it is unnecessary for detecting the structural features of the grid.

The preprocessing stage also prepares the image for edge detection. Each pixel is analyzed in relation to its neighbors to compute gradients in both horizontal and vertical directions. This operation highlights changes in intensity, which correspond to the edges of the grid lines and letters. Only edges with a magnitude above a defined threshold are considered significant. This ensures that minor noise or small artifacts do not interfere with the rotation process.

4.6.2 Edge Detection and Angle Calculation

Once the image is preprocessed, the algorithm detects the edges and calculates their orientation. For each pixel with a significant gradient, the algorithm computes the edge orientation angle using the arctangent of the vertical and horizontal gradient components. These angles are then adjusted to fall within a 0° to 180° range.

After computing the orientation of all significant edges, an angle histogram is constructed. Each edge contributes a vote to the corresponding bin in the histogram based on its angle. The angle with the highest number of votes is identified as the dominant angle, which represents the overall tilt of the grid in the image. This approach is conceptually similar to the Hough Transform in that it accumulates evidence for a particular orientation, but it operates at the pixel level rather than detecting explicit lines.

4.6.3 Determining the Rotation Angle

The dominant angle extracted from the histogram is then processed to determine the rotation angle that will align the grid. Minor corrections are applied to ensure the rotation is minimal and does not overcompensate. For example, angles larger than 90° are adjusted by subtracting 180° , and small deviations below a threshold are ignored.

This step ensures that the grid is rotated to an upright orientation while maintaining the integrity of the cells and minimizing distortions. By carefully adjusting the angle, the algorithm avoids introducing unnecessary artifacts that could affect later stages of processing.

4.6.4 Rotating the Image

The actual rotation of the image is carried out using a 2D rotation matrix. Each pixel in the output image is mapped back to its original coordinates using the inverse of the rotation matrix, and bilinear interpolation is applied to compute pixel values. This interpolation preserves the shapes of grid lines and letters, avoiding the blocky or distorted appearance that would result from nearest-neighbor interpolation.

To accommodate the change in dimensions caused by rotation, the algorithm calculates the new width and height of the rotated image. The center of rotation is adjusted accordingly to ensure that the grid remains centered in the new image frame. After the rotation, the resulting image is a fully aligned version of the original, ready for grid reconstruction.

4.6.5 Grid Reconstruction

Rotating the image visually is not sufficient; the internal representation of the grid must also be reconstructed to reflect the new orientation. Each cell's coordinates are recalculated based on the rotation transformation to ensure that the spatial relationships between cells are preserved.

This reconstruction is critical for the solver to function properly. If cells are misaligned or coordinates are incorrect, words could be missed or incorrectly interpreted. The algorithm maps every original cell position to its new location in the rotated grid, ensuring consistency between the image and the data structures used in the program.

4.6.6 Challenges and Solutions

During the development of the automatic rotation module, several challenges were encountered:

- Noisy images and artifacts: Shadows, uneven lighting, and image imperfections sometimes produced false edges. Thresholding in edge detection and careful selection of gradient magnitudes helped reduce these errors.
- Partial or skewed grids: When only part of the grid is visible or the tilt is extreme, the dominant angle estimation can be inaccurate. To address this, outlier angles were ignored in the histogram, and angle smoothing was applied.
- Preserving grid integrity: Rotating the image could distort cell shapes or letters. Bilinear interpolation and careful center alignment minimized these distortions.
- Updating the internal grid structure: Recalculating all cell positions after rotation required precise coordinate transformations to maintain the relationships between cells. Mistakes here could break the solver.
- Performance considerations: Iterating over every pixel and performing interpolation could be computationally intensive. Memory management and efficient loops were crucial for maintaining acceptable execution speed.

5 Artificial Neural Networks

5.1 General Overview

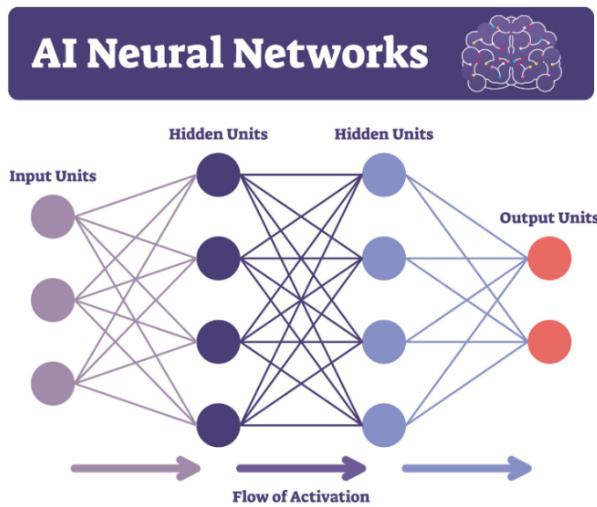
Optical Character Recognition (OCR) is a pivotal branch of artificial intelligence that allows the translation of text images (printed or handwritten) into machine-editable text files. In the context of our project, the objective was not limited to using an existing solution but aimed at the complete design of a recognition system capable of identifying isolated letters to reconstruct larger semantic entities, such as words or complete crossword grids. The heart of this system rests on an **Artificial Neural Network**(ANN). Unlike classical algorithmic approaches where every rule is manually coded (ex : if a vertical bar is followed by a circle, it's a 'b'), the neural network approach allows the system to learn these rules by itself from examples. Our major technical constraint was the "from scratch" development in the C language. We prohibited ourselves from using high-level Machine Learning libraries (like TensorFlow or Keras) to directly manipulate the underlying mathematical concepts: matrix calculation, partial derivation, and stochastic optimization. This report details our journey, from fundamental theory to the concrete challenges of implementing a network capable of processing a database of nearly 18,000 images.

5.2 Fundamental Principles of the Neural Network

To understand our implementation, it is essential to revisit the theory governing "Multilayer Perceptron" (MLP) type neural networks.

5.2.1 The Formal Neuron

The basic unit of our system is the artificial neuron. It is a mathematical function designed to roughly simulate the behavior of a biological neuron, which activates when it receives sufficient electrical stimulation.



In our model, each neuron receives a series of numerical input values (x_i). Each input is associated with a weight (w_i), which represents the importance of this connection (analogous to synaptic strength). The neuron calculates a weighted sum of these inputs, to which it adds a bias (b_i), an internal value that allows shifting the activation threshold³.

$$z = \sum (x_i \cdot w_i) + b$$

5.2.2 Non-Linearity: Activation Functions

The weighted sum alone is a linear operation. For the network to resolve complex problems (like distinguishing an 'O' from a 'Q'), we must introduce non-linearity. This is the role of the activation function. We chose to use two specific functions in our architecture :

- **Sigmoid (for the hidden layer):** It transforms any real value into a number between 0 and 1. It is a historical function in neural networks, ideal for learning because it is differentiable everywhere.
- **Softmax (for the output layer):** This function is crucial for classification. It transforms a vector of raw numbers into a probability distribution. The sum of all outputs equals 1, which allows interpreting the result as a "confidence rate" for each letter of the alphabet.

5.2.3 Layer Architecture

A single neuron can only solve very simple problems. We therefore organized them into a multilayer network (MLP):

Input Layer: It performs no calculation. It simply receives the image pixels. Each neuron corresponds to a pixel.

Hidden Layer: This is where the magic happens. The neurons in this layer combine information from the pixels to detect abstract "features" (a curve in the top left, a vertical bar in the center, etc.). We used a size of 32 neurons.

Output Layer: It aggregates the detected features to decide the final class of the image (is it an A, a B, ... a Z?). We set this to 26 neurons for the alphabet.

5.3 Training Methodology

A neural network is born "ignorant." Its weights are initialized with random values, which means that initially, its predictions are due to pure chance. Training is the process of adjusting these weights to minimize prediction error.

5.3.1 The Learning Cycle

Learning takes place in repeated cycles, called epochs. For each image in our database, we perform the following steps:

- Forward Propagation: The image traverses the network layer by layer. We calculate the output of the hidden layer, then that of the final layer.
- Loss Calculation: We compare the network's prediction with the "true" answer (the image label). For example, if the image is an 'A', the output corresponding to 'A' should be 1, and all others 0. The difference constitutes the error.
- Backpropagation: This is the most mathematical step. We use the gradient descent algorithm. It involves calculating the "responsibility" of each weight in the final error. Mathematically, we calculate the gradient of the error with respect to each weight. We start from the end of the network and move back towards the beginning (hence the name "backpropagation").
- Weight Update: We slightly modify each weight in the opposite direction of the error, multiplied by a factor called the Learning Rate. We used a rate of 0.01.

5.4 Our Implementation

The development of our letter-recognition system was far from a straightforward application of neural-network theory. As we progressed, we encountered obstacles that were not only technical but often conceptual. Each challenge forced us to rethink, reorganize, or revisit the foundations of what makes a neural network actually learn. Below are the main issues we faced and how we addressed them.

5.4.1 Understanding and Visualizing the Neural Network

One of the first barriers was conceptual. A neural network cannot be debugged like a traditional algorithm. Nothing crashes. No error message pops up to guide you. A single wrong sign in a derivative or a misplaced matrix index will not stop the program, but it will silently prevent the network from learning. The loss remains flat, and the system appears "alive" yet completely ineffective.

This gave the impression of working with a sealed box. We were manipulating matrix multiplications and activation functions, but their real impact was difficult to see. When the loss stagnated, we had no immediate clue whether the problem came from the forward pass, the gradient formulas, or the preprocessing of the dataset.

To regain control, we adopted a step-by-step approach. Before writing code, we tested our formulas on tiny “paper networks” with just a few neurons to manually check the computations. Then, during development, we monitored the loss curve after every epoch. A consistent decrease was the only reliable indication that backpropagation was behaving correctly. Without this method, most debugging would have been guesswork.

5.4.2 Implementing Everything from Scratch and Managing Memory

Translating mathematical equations directly into C means dealing with memory at a very low level. No automatic garbage collector comes to the rescue, and every SDL image loaded into memory must eventually be freed. In practice, handling thousands of training images quickly exposed weaknesses in our initial implementation.

In early tests, the program consumed more and more memory as it ran. Surfaces loaded through SDL2 were not released properly, and training for long sessions would eventually saturate the RAM.

We solved this by enforcing strict memory hygiene. Every loaded surface had to be freed immediately after it was processed. We reorganized the training pipeline so that only the essential data remained in memory: the weight matrices and the current image being processed. With this change, the system became stable, efficient, and capable of handling large datasets without collapsing.

5.4.3 Building and Normalizing the Dataset

A neural network cannot learn without clean and consistent data. Our initial corpus, however, came from mixed sources with different sizes, ratios, brightness conditions, and levels of contrast. A small “A” tucked into a corner and a large “A” centered in the frame are mathematically two very different objects. Without normalization, the network was effectively seeing multiple unrelated versions of the same letter.

To fix this, preprocessing became essential. Every image was converted to grayscale and resized to a strict 32×32 pixel grid. This step positioned the character consistently and ensured that the network always received inputs with the same geometry. This normalization phase was one of the key factors that stabilized training and improved the network’s ability to generalize.

5.4.4 Saving and Reloading Network Weights

In the beginning, the network lost all of its progress each time the program stopped. Retraining from scratch on tens of thousands of images was not sustainable, especially as the system grew more complex.

We implemented a serialization mechanism that exports all weight matrices and bias vectors into lightweight text files. When the application starts, it checks if these files exist. If they do, the network instantly reloads its learned parameters and becomes ready for inference. If not, a full training session is launched and the newly learned weights are saved automatically.

This feature transformed our system from a temporary prototype into a practical tool suitable for real OCR tasks involving entire words or crossword grids.

5.5 Lessons Learned from the Neural Network Implementation

The neural network component of our system became a project within the project, and working on it taught us far more than expected. Implementing a complete learning pipeline entirely from scratch forced us to confront the internal mechanics that are usually hidden behind high-level libraries. Instead of calling prepackaged functions, we had to understand every operation, from the structure of the matrices to the subtle interplay between activation functions, gradients, and weight updates. This hands-on experience gave us a much deeper, more intuitive grasp of how a neural network actually learns.

The final implementation now behaves as a self-contained recognition engine. It preprocesses raw images, converts them into standardized inputs, and runs them through a multilayer perceptron trained on thousands of samples. The network is able to distinguish between the 26 letters of the alphabet with reliable accuracy, despite the inherent limitations of a simple MLP architecture. More importantly, it does so using tools that we built ourselves, which gives us full visibility into every stage of the computation. This transparency is valuable for both debugging and future improvements.

Throughout the development, we also discovered how sensitive neural networks are to data structure and preprocessing choices. Normalizing every image to a 32×32 grid proved essential, as even small inconsistencies in scale or alignment could disrupt learning. Likewise, storing and reloading trained weights gave the system a sense of continuity, turning it into a practical tool instead of an academic exercise that must relearn everything at each launch.

Several paths for improvement emerged naturally from our observations. Adding convolutional layers would allow the system to extract spatial features more intelligently and become more resistant to shifts, rotations, or distortions in the input. Adjusting the input layer to use all 1024 pixels of the normalized images could help capture finer structural differences between letters such as “O” and “Q” or “C” and “G.” We could also explore deeper architectures, introduce regularization techniques, or expand the dataset using synthetic variations to increase robustness.

All these possibilities highlight what this implementation really achieved: not just a functioning neural network, but a solid technical foundation that can support more ambitious experiments. Through this work, we developed a clearer understanding of how learning emerges from numerical operations, how small design decisions influence performance, and how a network evolves as it absorbs examples. The module now plays a central role in the OCR pipeline, and it opens the door to richer and more sophisticated recognition systems in the future.

6 Conclusion

The OCR Word Search Solver project has proven to be a challenging yet highly instructive experience, combining multiple areas of computer science, including image processing, algorithm design, low-level programming, and software integration. The development of the automatic rotation and grid reconstruction modules has been particularly significant, as these components ensure that input images, regardless of their initial orientation, can be processed reliably and accurately.

By the time of this final defense, the team has successfully implemented a complete preprocessing pipeline capable of automatically detecting the dominant orientation of a word search grid, rotating the image accordingly, and reconstructing the internal representation of the grid to match the rotated image. These capabilities are essential for ensuring that the solver can function correctly, as any misalignment could compromise the detection of words and the overall accuracy of the application.

The progress achieved demonstrates not only technical proficiency in image processing and algorithm implementation but also effective problem-solving in addressing challenges such as noisy images, partial or skewed grids, and precise cell alignment after rotation. The graphical user interface now allows users to interact with the solver and visualize the results in a coherent and intuitive manner, confirming the usability of the system.

Future objectives will focus on further refining the preprocessing phase, improving the robustness of rotation and reconstruction for more complex or degraded images, and integrating these components seamlessly with the solver algorithm. These steps will bring the project closer to a fully functional, user-friendly application capable of accurately solving any word search puzzle from an image.

In conclusion, the work completed in this phase highlights the importance of careful preprocessing in OCR applications and demonstrates the team's ability to implement a reliable, automated system from image input to a fully aligned grid. The foundation is now in place for the final stages of development, which will focus on enhancing robustness, efficiency, and overall system integration.

7 Bibliography

- GTK Documentation. (2025). *GTK 3 Reference Manual*. Retrieved from docs.gtk.org
→ Used for designing the graphical user interface, managing widgets and signals, and rendering images using `GtkDrawingArea` and Cairo in the C application.
- SDL Documentation. (2025). *Simple DirectMedia Layer 2.0 API Reference*. Retrieved from wiki.libsdl.org
→ Used for low-level image loading, pixel format conversion (ARGB8888), surface manipulation, and saving bitmap images during preprocessing and denoising.
- Wikipedia contributors. (2025). *Median filter*. In *Wikipedia, The Free Encyclopedia*. Retrieved from en.wikipedia.org
→ Used to justify the application of a 3×3 median filter for noise reduction while preserving grid lines and character edges.
- Cairo Graphics Documentation. (2025). *Cairo 2D Graphics Library*. Retrieved from cairographics.org
→ Used for implementing real-time image rotation, scaling, and rendering within the GTK drawing area.
- Wikipedia contributors. (2025). *Word search*. In *Wikipedia, The Free Encyclopedia*. Retrieved from en.wikipedia.org
→ General description and structure of word-search puzzles, used to understand grid formatting and expected solver behavior.