AlphaLogos: Logic Synthesis Optimization using Quine–McCluskey Algorithm

Ahmed Allam
Computer Science Dept.
American University in Cairo
Cairo, Egypt
ahmedeallam@aucegypt.edu

Mohamed Abdelmagid
Computer Science Dept.
American University in Cairo
Cairo, Egypt
mabdelmagid@aucegypt.edu

Mohamed Mansour
Computer Science Dept.
American University in Cairo
Cairo, Egypt
Hamdy47@aucegypt.edu

***Abstract*—In this project, we present AlphaLogos, a web-based application that optimizes logic synthesis using the Quine–McCluskey algorithm. The application allows users to enter a logical expression and generates the corresponding truth table. It then applies the Quine–McCluskey algorithm to obtain the canonical sum-of-products (SoP) and product-of-sums (PoS) forms. The application also identifies the prime implicants and essential prime implicants. Finally, it generates the minimized expression using three-step heuristic and Petrick's method and displays the corresponding Karnaugh map and interactive logic circuit. The application is deployed on a web server and can be accessed using a web browser. The application is available at AlphaLogos. All the source code is available at Github.**

***Index Terms*—Quine–McCluskey algorithm, Logic Synthesis, Optimization, Karnaugh Map**

I. Project Workflow - *Ahmed Allam*

A. Github Repository

In the early stages of the AlphaLogos project, a clear and robust workflow was established to maintain a well-organized Git repository. This workflow was carefully planned and initiated by Ahmed Allam, and was followed by all team members, ensuring a coordinated and efficient development process.

1) **Issue Creation:** Before making any modifications, team members were required to create an issue detailing the intended changes, whether it was a feature addition or a bug fix. Detailed requirements were provided for feature enhancements, while steps to reproduce were provided for bug fixes. Team members then assigned themselves to the respective issue.
2) **Branch Creation:** Once an issue was assigned, a new branch was created from the latest updates in the main branch. This practice ensured that the main branch remained stable and free from unreviewed changes.
3) **Test-Driven Development (TDD):** Following the Test-Driven Development approach, the first step after branch creation involved creating automated unit tests. These tests were designed to cover all the requirements specified in the issue, ensuring that all features or fixes were thoroughly validated against predefined benchmarks before implementation.
4) **Feature Implementation:** With a solid set of tests in place, team members proceeded to implement the necessary functions to address the feature or fix the bug as detailed in the issue.
5) **Pull Requests and Issue Closure:** Upon completion of the feature or bug fix, a pull request (PR) was created, highlighting the primary changes made. The PR also mentioned that it resolved the original issue upon merging.
6) **Branch Protection and Code Reviews:** To ensure the integrity of the main branch, protection rules were set up. Every PR required at least one code review and approval from a fellow team member (excluding the author). Additionally, all automated checks configured on GitHub had to pass successfully with each push.
7) **Automated Formatting and Static Analysis:** To maintain a consistent coding style and to identify potential issues early, Git pre-commit hooks were utilized. Before each commit, these hooks triggered 'clang-format' to ensure consistent formatting across all edited files, and 'clang-tidy' to perform static analysis on the C++ code, checking for any hidden issues or warnings.

Overall, with more than 25 pull requests and 15 issues on Github, the disciplined Git workflow facilitated smooth collaboration, ensuring a clean, organized, and error-free codebase. It significantly contributed to the successful deployment and functionality of AlphaLogos, showcasing a well-coordinated team effort in software development.

B. Test-driven development (TDD)

Test-Driven Development (TDD) is a software development approach where tests are written before the code that needs to be tested. The process is primarily divided into the following iterative cycle: write a test, write the minimum amount of code necessary to pass the test, and then refactor the code.

The TDD cycle promotes a robust, error-free codebase, and ensures that all code functionality is verified against defined test cases.

In the AlphaLogos project, every function implemented within the `src` directory was required to have corresponding unit tests in the `tests` directory. These tests were designed to cover all code paths and possible test cases, ensuring that every function performed as expected under a variety of conditions.

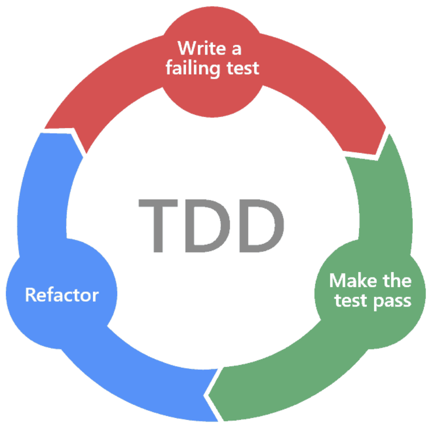
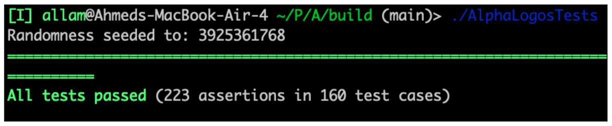


Fig. 1. Cycle of TDD

This practice significantly contributed to the reliability and robustness of the software.

For the implementation of automated unit tests, we utilized Catch2, a C++ library known for its simplicity and effectiveness in testing. Catch2 facilitated the writing and management of test cases, making the TDD process streamlined and efficient.

Our project comprised a total of 223 test cases, each meticulously crafted to verify the functionality of individual functions. The comprehensive suite of tests provided a solid foundation for continuous integration and ongoing development, ensuring that every change in the codebase was validated rigorously.



```
[I] allam@Ahmeds-MacBook-Air-4 ~/P/A/build (main)> ./AlphaLogosTests
Randomness seeded to: 3925361768
===============================================================================
==========
All tests passed (223 assertions in 160 test cases)
```

Fig. 2. Terminal showing all tests passing

The diligent application of the TDD methodology, aided by Catch2, significantly bolstered the reliability and maintainability of the AlphaLogos application. Through rigorous testing, we were able to ensure that the logic synthesis optimizations performed by AlphaLogos were both accurate and efficient, marking a crucial step towards achieving the project's objectives.

II. Input Validation - *Mohamed Mansour*

In the realm of Boolean algebra, the evaluation of expressions often involves intricate structures such as Product of Sums (PoS) or Sum of Products (SoP). These structures, while foundational to understanding and implementing logical circuits, can also be error-prone when input by users. It becomes imperative, therefore, to have a robust mechanism to validate these inputs. This section delves into the meticulous process of how user-provided terms, be it PoS, SoP, or other invalid expressions, are validated. Proper input validation serves a dual purpose.

A. `isValidSoP` Function

The ‘isValidSoP‘ function plays a crucial role in ensuring that Sum of Products (SoP) Boolean expressions are syntactically correct and adhere to expected conventions. Specifically, this function examines an expression string and scrutinizes the following:

- **Parentheses Matching**: The function ensures that every opening parenthesis ‘(‘ has a corresponding closing parenthesis ‘)‘. It also guarantees that parentheses do not nest within one another, which is a common stipulation for Boolean algebraic representations.
- **Operator Usage**: The operators ‘+‘ and ‘*‘ have distinct roles in SoP expressions. The function verifies that each operator is used correctly. For instance, it checks that a ‘+‘ operator does not immediately follow another operator, ensuring that expressions like ‘AB+*C‘ are flagged as invalid.
- **Apostrophe Negation**: In Boolean algebra, an apostrophe (‘’‘) after a variable denotes its negation. The function ascertains that apostrophes are used correctly - specifically, that they appear only after alphabetic characters and not consecutively, making expressions like ‘A”B‘ invalid.
- **Spacing and Character Sequencing**: White spaces are often ignored, but the function ensures that sequences like ‘A ’‘ (with a space between the variable and its negation) are considered invalid. The sequence of characters is also examined to prevent constructs like ‘++‘ or ‘**‘ from being deemed valid.
- **DeMorgan’s Law and Double Negation**: The function verifies that the expression adheres to DeMorgan’s law, ensuring that double negation and the distributive properties of Boolean algebra are preserved. This check ensures that common simplifications like ‘A”‘ (double negation) result in ‘A‘, and expressions conforming to DeMorgan’s transformations are validated correctly.

B. `isValidPoS` Function

The ‘isValidPoS‘ function, while analogous in purpose to its SoP counterpart, has its unique set of validation criteria tailored for Product of Sums (PoS) Boolean expressions:

- **Parentheses Handling**: Unlike the SoP validator, this function is especially stringent about what can reside inside parentheses. It ensures that each parenthesis pair encapsulates valid constructs, without the intrusion of nested parentheses.
- **Operator Placement**: The ‘+‘ operator has a specialized role within parentheses in PoS expressions. The function confirms that the ‘+‘ operator is used only between single characters (and their optional negations). This prevents erroneous constructs like ‘A+B+C‘ inside a single set of parentheses.
- **Apostrophe Negation**: As with the SoP function, the PoS validator ensures the correct usage of apostrophes.

It guarantees that they appear only after single alphabetic characters and not in immediate succession.

- **Character Validation**: Beyond ensuring the correct placement and sequence of operators and variables, the function checks for any invalid characters, both inside and outside parentheses. This comprehensive check guarantees that only syntactically valid PoS expressions pass the validation.
- **DeMorgan's Law and Double Negation**: For PoS expressions, the function validates that they conform to the principles of DeMorgan's law and that any instances of double negation are correctly interpreted. This ensures that expressions like '(A'+B')'' are recognized and simplified appropriately, and constructs following DeMorgan's transformations are validated accurately.

III. Truth Table Generation - *Mohamed Abdelmagid*

The main function of the Algorithm generateTruthTable receives the expression represented as a vector of implicants and returns a vector of vectors in bool. There are many functionalities related to generating a truth table for a given logical expression. Let's go through each function and understand its purpose:

A. `addParenthesesForPrecedence`

This function takes a vector of `Token` objects representing a logical expression and adds parentheses to the expression to enforce operator precedence rules. It scans the tokens and identifies sections that need parentheses based on the presence of the OR operator (`TokenType::OR`). If a section is already enclosed in parentheses, it preserves it; otherwise, it adds parentheses around the section. The function returns a new vector of tokens with the added parentheses.

B. `generatePermutations`

This function generates all possible permutations of variable assignments for a given set of unique variables in the logical expression. It takes a vector of `Token` objects representing the logical expression and returns a vector of vectors, where each inner vector represents a permutation. For instance, if there are two unique variables, the function will generate four permutations corresponding to all possible assignments of true/false to each variable.

C. `evaluateExpression`

This function evaluates a logical expression for a given permutation of variable assignments. It takes a vector of `Token` objects representing the logical expression, an index value (passed by reference) to keep track of the current token being evaluated, and a vector of pairs representing the current permutation of variable assignments. The function recursively evaluates the expression and returns the resulting boolean value.

D. `generateTruthTable`

This function generates a truth table for a given logical expression. It takes a vector of `Token` objects representing the logical expression. First, it calls `addParenthesesForPrecedence` to ensure operator precedence. Then, it calls `generatePermutations` to obtain all possible variable assignments. Finally, it iterates over each permutation, evaluates the logical expression using `evaluateExpression`, and constructs the truth table row by appending the resulting boolean value. The function returns a vector of vectors, where each inner vector represents a row of the truth table.

IV. Canonical Forms - *Mohamed Abdelmagid*

The provided code includes functions related to generating canonical sum-of-products (SoP) and product-of-sums (PoS) forms based on a truth table. The function receives the truth table and the unique variables of the expression to return a vector of minterms that can be translated to canonical SoP or a vector of maxterms to represent PoS. Let's examine each function and understand its purpose:

A. `generateMinTerms`

This function takes a vector of `Token` objects representing unique variables and a truth table represented by a vector of vectors of booleans. It generates the canonical sum-of-products (SoP) form by iterating over the truth table. For each row in the truth table, if the last value in the row is `true`, it creates a `Minterm` object. The `Minterm` object stores the index of the row, the binary representation of the variables, the count of ones in the binary representation, and a flag indicating if the minterm is covered. The function returns a vector of `Minterm` objects representing the canonical SoP form.

B. `generateMaxTerms`

This function takes a vector of `Token` objects representing unique variables and a truth table represented by a vector of vectors of booleans. It generates the canonical product-of-sums (PoS) form by iterating over the truth table. For each row in the truth table, if the last value in the row is `false`, it creates a `Maxterm` object. The `Maxterm` object stores the index of the row, the binary representation of the variables, and a flag indicating if the maxterm is covered. The function returns a vector of `Maxterm` objects representing the canonical PoS form.

V. Prime Implicants - *Ahmed Allam*

Prime implicants play a crucial role in the simplification of Boolean expressions, a core aspect of logic synthesis in the AlphaLogos application. The process of identifying prime implicants involves analyzing a collection of minterms from the given logical expression. A minterm is a product of literals (variables or their complements) representing a row in the truth table where the function evaluates to 1.

The algorithm for generating prime implicants follows these steps:

1) Initialize an empty list to hold the prime implicants and another list (referred to as a column) to hold implicants generated at each stage of the process.
2) Populate the initial column with implicants created from the given minterms. Each implicant holds information about the minterms it covers, its binary representation, and flags indicating whether it has been checked or is part of a larger implicant.
3) Enter a loop that continues until the column is empty, indicating that all possible implicants have been explored.
4) Within the loop, initiate a new column for the next level of implicants. Iterate through the current column, comparing each implicant with every other implicant to find pairs that differ in exactly one bit position.
5) When such a pair is found, create a new implicant that combines the minterms of the two original implicants and has a binary representation with a -1 in the position where they differed. This represents a don't-care condition, indicating that the bit in this position can be either 0 or 1. Add this new implicant to the next column and mark the original implicants as checked.
6) After all pairs have been explored, check for any unchecked implicants in the current column that are not already in the list of prime implicants, and add them to the list of prime implicants. These are implicants that cannot be combined with others to form larger implicants.
7) Replace the current column with the next column and repeat the process until the column is empty.
8) Return the list of prime implicants.

In the implemented function generatePrimeImplicants, a vector of minterms is accepted as input, and a vector of prime implicants is returned as output. Through nested loops, the function efficiently compares and combines implicants, gradually building up to the prime implicants which are crucial for the Quine–McCluskey algorithm used in logic minimization. This structured and systematic approach ensures that all prime implicants are identified accurately, laying a solid foundation for the subsequent steps in the logic synthesis process.

VI. Essential Prime Implicants - *Mohamed Mansour*

Essential Prime Implicants (EPIs) are a fundamental concept in the minimization of Boolean functions. Their role is pivotal as they represent minterms that can only be covered by only one prime implicant. Consequently, the identification and extraction of these EPIs is of paramount importance when aiming for a minimized Boolean expression while using Quine-McCluskey Algorithm.

A. Conceptual Overview

An implicant is termed as "essential" if there isn't any other prime implicant that covers any of its minterms. In simpler terms, a minterm covered by an essential prime implicant doesn't appear in any other prime implicant. Thus, it becomes imperative to include the EPI in the minimized expression.

B. Algorithm Design

The core of the EPI identification process revolves around two primary functions:

- `compareImplicants(const Implicant& a, const Implicant& b)`: A utility function, it facilitates the comparison between two implicants. It is primarily employed during the sorting phase to arrange implicants in a specific order.
- `generateEssentialPrimeImplicants(const vector<Implicant>& primeImplicants)`: This function's task is to iterate through the given prime implicants and systematically identify and extract EPIs, returning them as a vector.

C. Operational Mechanism

The algorithm exploits the associative mapping capabilities provided by the `map` data structure in C++. For each prime implicant, it maps its minterms to the implicants they belong to. As the mapping is constructed, the algorithm assesses each minterm. If a minterm is found to be linked with only a single implicant, this implicant is categorized as essential. This method ensures that every essential prime implicant is captured, avoiding redundancy.

D. Code Insights

The algorithm begins by defining a comparison function for implicants, ensuring they can be sorted effectively. Subsequently, the main function `generateEssentialPrimeImplicants` creates an associative map between minterms and their corresponding implicants. As the mapping unfolds, any minterm associated with just one prime implicant highlights that implicant as essential. The identified EPIs are then collected in a vector, sorted, and returned.

VII. Uncovered Minterms - *Mohamed Abdelmagid*

The algorithm mainly represents in a function that takes a vector of `Minterm` objects representing the minterms and a constant reference to a vector of `Implicant` objects representing the essential implicants. It aims to identify the minterms that are not covered by any of the essential implicants.

1. It starts by creating an empty vector called `uncoveredMinterms` to store the uncovered minterms.

2. Next, it iterates through each `Minterm` object in the `minterms` vector and sets the `is_covered` flag of each minterm to `false`. This step ensures that initially, no minterm is marked as covered.

3. Then, it iterates through each `Implicant` object in the `essentialImplicants` vector. For each implicant, it further iterates through the minterm indices stored in the `minterms` member variable of the implicant. In each iteration, it searches for the corresponding minterm in the `minterms` vector and sets its `is_covered` flag to `true`. This indicates that the minterm is covered by at least one essential implicant.

4. Finally, it iterates through each `Minterm` object in the `minterms` vector and checks if its `is_covered` flag is `false`. If so, it means the minterm is not covered by any essential implicant and therefore adds it to the `uncoveredMinterms` vector.

5. The function returns the vector `uncoveredMinterms`, which contains all the minterms that are not covered by any essential implicant.

VIII. Minimization by 3-Step Heuristic - *Mohamed Abdelmagid*

The function aims to minimize Boolean expressions using the three-step heuristics approach. The function mainly recieves a vector of prime implicants and return a vector of Implicants representing Here's a breakdown of the code and its functionality:

A. `eliminateDominatingColumns`

This function eliminates the dominating columns from the uncovered minterms. It iterates over each pair of uncovered minterms and checks if one minterm dominates the other. If a dominating relationship is found, the dominated minterm is removed from the list of uncovered minterms.

B. `eliminateDominatedRows`

This function eliminates the dominated rows from the prime implicants (PIs). It iterates over each pair of prime implicants and checks if one implicant dominates the other. If a dominated relationship is found, the dominated implicant is removed from the list of PIs.

C. `Minimization`

This function performs the 3-Step Heuristics method for expression minimization. It takes a set of prime implicants (PIs), a set of essential prime implicants (EPIs), and a set of uncovered minterms as inputs. The function first adds the EPIs to the minimized expression and removes the corresponding minterms from the uncovered minterms list. Then, it iteratively applies the three-step heuristics to further minimize the expression:

1) **Step 1**: Eliminate dominating columns and remove the corresponding minterms from the uncovered minterms list.
2) **Step 2**: Eliminate dominated rows from the PIs.
3) **Step 3**: Select the PI that covers the maximum number of remaining uncovered minterms and add it to the minimized expression. Remove the corresponding minterms from the uncovered minterms list and remove those minterms from all other PIs.

The function continues this process until all the minterms are covered, and the minimized expression is obtained.

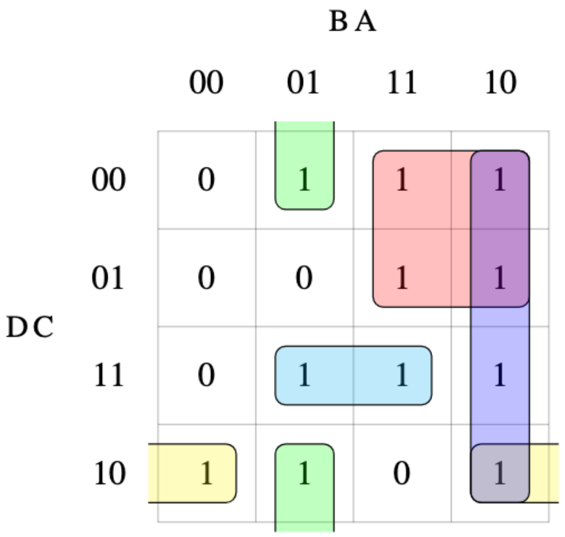


Fig. 3. An Example K-Map generated by AlphaLogos

IX. Karnaugh Map - *Ahmed Allam*

A pivotal aspect of representing logical expressions in a simplified form is through the use of Karnaugh Maps (K-maps). In AlphaLogos, the process of drawing K-maps is automated, translating prime implicants into a visual representation. This is achieved through a dedicated function in C++ that converts the list of prime implicants, obtained post minimization, into LaTeX code. The LaTeX code is structured to utilize the `karnaugh-map` package, which facilitates drawing K-maps in a colorful, visually appealing manner.

The C++ function `makeKMapLaTeX` is orchestrated to handle the translation of prime implicants into LaTeX code. It evaluates the number of variables involved to determine the dimensions of the K-map, subsequently creating the LaTeX environment for the K-map. The function then iterates through the minterms and prime implicants, calling another function `generateLatexForImplicant` to obtain the LaTeX commands for drawing each implicant on the K-map.

Within `generateLatexForImplicant`, the implicants are analyzed and translated into respective LaTeX commands based on their size and positions within the K-map. The function discerns whether an implicant covers a corner, an edge, or other areas of the K-map, and constructs the appropriate LaTeX command to represent the implicant covering on the K-map.

Post the generation of LaTeX code, a temporary `.tex` file is created on the server which houses this code. The command `pdflatex` is then executed against this `.tex` file, engendering a temporary `.pdf` file that visualizes the K-map. To ensure compatibility and seamless display on the web application, a subsequent command, `inkscape`, is run to convert the `.pdf` file into an `.svg` file. The `.svg` file, being a vector graphic, is apt for rendering on the web browser, ensuring that users have a clear, interactive view of the Karnaugh Map corresponding to their logical expression.

This automated process of generating, converting, and rendering K-maps not only simplifies the visualization of logical simplifications but also enriches the user experience by providing interactive and colorful representations of the logical expressions. Through the utilization of LaTeX for drawing, and `pdflatex` and `inkscape` for conversion, the application manages to bridge the computational logic with a user-friendly visual output, making logic synthesis more intuitive and engaging.

X. Logic Circuit - *Ahmed Allam*

Upon acquiring the minimized expression, the journey towards visualizing the logic circuit commences. The first stride in this is the generation of Verilog code, a hardware description language used for describing the behavior of electronic systems. This Verilog code serves as a blueprint for the logic circuit corresponding to the minimized expression.

The function `SoPToVerilog` has the responsibility of transmuting a Sum of Products (SoP) expression into Verilog code. This function iterates through the tokens representing the SoP expression, identifying variables, and constructing the necessary Verilog syntax to define the inputs and output of the circuit.

A Verilog module named `circuit` is generated, with declarations for input and output. The function meticulously constructs AND groups from the tokens, concatenating variables and NOT operations as required. Whenever an OR token is encountered, the accumulated AND group is finalized and stored, ready to be part of the OR operation in Verilog syntax. This process continues until all tokens have been processed, ensuring that all AND and OR operations are accurately represented in the Verilog code.

The assembled Verilog code embodies the logic of the minimized expression and is written to a temporary file with a `.v` extension on the server.

Subsequent to the Verilog code generation, the server invokes the command parser in `Yosys2DigitalJS`, which in turn beckons `Yosys` to synthesize the circuit from the Verilog code. Yosys, an open-source framework for Verilog synthesis, performs the necessary analysis and synthesis to convert the Verilog code into a netlist, a detailed description of the circuit including gates and connections.

Following the synthesis, the netlist is transmuted into DigitalJS format, a JSON representation delineating all the gates, connectors, and the overall structure of the circuit. This JSON file, now holding the essence of the logic circuit, is transmitted to the front-end.

On the front-end, the `render` function from the DigitalJS library is called to render the JSON representation, culminating in a vibrant, fully interactive logic circuit simulation right within the browser. This interactive simulation grants users the ability to visually comprehend the logic circuit, explore its operation, and interact with it in real-time, bridging the abstract logical expressions with tangible, visual logic circuits. Through this automated, well-orchestrated flow from logical expressions to interactive circuit simulations, AlphaLogos furnishes a compelling, educational, and engaging platform for users delving into the world of logic synthesis and circuit design.

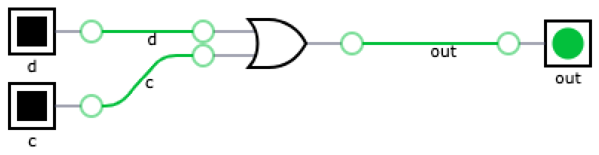


Fig. 4. An Example Interactive Circuit Simulation generated by AlphaLogos

XI. Front-end Webpage - *Ahmed Allam*

A. Development

The website for AlphaLogos was developed by Ahmed Allam. A blend of different technologies and frameworks was utilized to ensure a responsive, interactive, and user-friendly interface.

- **Backend Framework:** The backbone of the server-side operations was structured using a C++ library named Crow. Crow is a lightweight and intuitive backend framework that facilitates the handling of web requests and responses. It provided the necessary scaffolding to manage the interactions between the users and the server efficiently.
- **HTML Templates:** For creating dynamic and organized web pages, HTML templates were employed in conjunction with Jinja. Jinja is a fast, expressive, and succinct templating engine that allows for the injection of data into HTML templates. This made it possible to present data dynamically on the web pages, offering a tailored user experience.
- **JavaScript and CSS:** To enhance the interactivity and aesthetic appeal of the webpage, JavaScript and CSS (Cascading Style Sheets) were utilized. JavaScript enabled the creation of interactive elements on the webpage, enhancing the user engagement. On the other hand, CSS was employed to style the webpage, ensuring a visually pleasing and intuitive design.

The harmonious integration of Crow for backend operations, Jinja for HTML templating, and JavaScript and CSS for front-end interactivity and styling, empowered the creation of a compelling webpage. This well-architected webpage serves as the gateway for users to interact with the AlphaLogos application, making logic synthesis both accessible and enjoyable. Through these proficient development practices, the webpage stands as a user-centric platform, bridging the intricate logic synthesis operations with a simple, intuitive, and engaging user interface.

B. Deployment

The deployment of AlphaLogos was carried out with an emphasis on ensuring a seamless and robust operational environment. Utilizing a combination of modern tools and cloud services, the application was orchestrated to be highly accessible and reliable. Below are the key components and steps involved in the deployment process:

- **Build System:** The build system for the project was managed using CMake, a cross-platform build-system generator. CMake streamlined the process of compiling the source code, managing dependencies, and setting up the build configurations, ensuring a consistent build process across different environments.
- **Containerization:** Docker was employed to containerize the AlphaLogos application. Containerization encapsulates the application along with its dependencies into a 'container'. This ensures that the application runs reliably across different computing environments. Docker simplified the process of packaging the application and its dependencies into a portable container.
- **Cloud Hosting:** The application was hosted on the Amazon Web Services (AWS) platform. Specifically, the Docker containers were stored on AWS Elastic Container Registry (ECR), and deployed using AWS Elastic Container Service (ECS). This setup facilitated a scalable and manageable runtime environment for AlphaLogos.
- **Load Balancing:** To ensure efficient distribution of network traffic across several servers, a load balancer was employed. The AWS Load Balancer automatically routes the traffic to maintain a smooth user experience, even during times of high traffic.
- **Domain Acquisition:** An effort was made to acquire a custom domain to provide a more professional appearance, replacing the default AWS load balancer link. However, due to recent international payment restrictions in Egypt, the purchase of a custom domain could not be completed. There's a potential to revisit this in the future, aiming to provide a more polished and professional URL for accessing AlphaLogos.

The deployment strategy was meticulously crafted to ensure the robustness and accessibility of AlphaLogos. Through the prudent use of CMake for build management, Docker for containerization, AWS for cloud hosting and load balancing, the deployment successfully provides a stable and efficient platform for users to interact with AlphaLogos.

XII. Conclusion

The AlphaLogos project exemplifies a meticulous blend of logic synthesis, web development, and deployment strategies to provide a user-centric platform. Through intuitive web interfaces, users can explore and understand logic minimization with ease. The seamless deployment ensures a reliable and accessible service. AlphaLogos stands as a noteworthy stride towards making logic synthesis both engaging and educative for a broader audience.

A. Future Work

The journey of AlphaLogos has unveiled several avenues for enhancement and expansion. Here are some of the potential areas of focus for future work:

- **Optimization for Large Minterm Cases:** During the testing phase, it was observed that the Quine-McCluskey algorithm struggled with a high number of minterms, taking a significant amount of time to process. A potential improvement could be to use maxterms instead of minterms in scenarios where they are fewer, like in the expression A+B+C+D+E+F+G. The sole maxterm in this case would be considerably easier for the algorithm to handle compared to the myriad of minterms.
- **Heuristic Algorithms:** Transitioning to heuristic algorithms like Espresso, which operates in polynomial time as opposed to the exponential time complexity of the Quine-McCluskey algorithm, could be a substantial improvement. This change can drastically reduce the time required to perform logic minimization, especially for complex expressions.
- **Deep Reinforcement Learning:** Modern approaches utilizing deep reinforcement learning for logic minimization can also be explored. As demonstrated by A. Hosny, S. Hashemi, M. Shalan, and S. Reda in "DRiLLS: Deep Reinforcement Learning for Logic Synthesis," deep learning techniques, particularly the actor-critic method, have shown promise in this domain. This cutting-edge technique could potentially optimize the logic minimization process further.
- **Academic Collaboration:** There's an aspiration to collaborate with the Computer Science and Engineering department at the American University in Cairo (AUC) to integrate AlphaLogos within the academic curriculum. This collaboration could enrich the learning experience by providing a practical, interactive tool for students to delve deeper into logic synthesis and digital design.

These future improvements aim to refine the efficiency, speed, and educational value of AlphaLogos, making it an even more robust and valuable tool for both academic and self-driven learning endeavors in logic synthesis.

References

[1] A. Hosny, S. Hashemi, M. Shalan and S. Reda, "DRiLLS: Deep Reinforcement Learning for Logic Synthesis," 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), Beijing, China, 2020, pp. 581-586, doi: 10.1109/ASP-DAC47756.2020.9045559.