GARCIA DIAZ, Antonio
Academic Year 2016-2017

PROJ-H402 Computing Project

Karnaugh Map Generator

The aim of this project was to create a JavaScript web application that allows students to train themselves in logic circuit synthesis, specifically in the solving of Karnaugh Maps, and facilitates the creation of new exercise sheets by teachers. The application allows the user to generate their own exercises by building a custom Karnaugh Map of up to 5 logic variables (with an option for "don't care" symbols), and dynamically displays the exercise's solution in real-time, by means of an interactive logic function, and an exportable string of LaTeX (askmaps) code.

1. Introduction: What is a Karnaugh Map?

The use of a Karnaugh Map[1] (or K-Map, named after Maurice Karnaugh) is a frequently used method to simplify Boolean algebra expressions, especially when working with logic circuits. It consists in rewriting a truth table, one of the most classical representations of a logic expression or circuit, into a 2D grid (or a 3D grid in some cases[2]) where each cell space's position represents a combination of input options (the variables), and its content represents the system's output value for such combination. The cell spaces are ordered in what is called Gray Code (00, 01, 11, 10), which allows to recognize patterns in the K-Map that correspond to terms in the (optimal) canonical form of the table's logic function.

The patterns to be recognized in the K-Map are called n-cubes. They are based on the concept of n-dimensional cubes, where each dimension corresponds to one of the logic variables. These n-cubes are thus groups of cell spaces that are linked together through "edges": a single space in the K-Map is a cube of dimension 0, and a cube of dimension n is formed from two cubes of dimension n-1 that are the same except for one input variable, set to 0 in one cube and to 1 in the other (this is where the new edges appear).

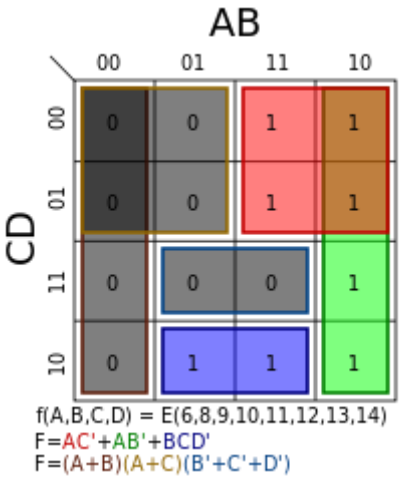

Fig. 1: A K-Map diagram with n-cubes highlighted. Those cubes containing only "1" values (here coloured) correspond to the minterms of the logic function, those containing only "0" values (here in grey) correspond to the maxterms.

When the n-cubes are highlighted in the K-Map (fig. 1), they appear as actual rectangles containing the linked spaces. Any possible rectangle with 2^n spaces along each side (1, 2, or 4

[1] Source: https://en.wikipedia.org/wiki/Karnaugh_map
[2] In a case with 5 logic variables, the fifth variable may be seen as the "depth". This is represented by putting two 4-variable Karnaugh Maps side by side, one for each state of the fifth variable.

spaces) is in fact an n-cube. The K-Map is also toroidally connected, so an n-cube may also come across the limits of the K-Map and appear on the opposite side.

The web application that was developed in this project was conceived for creating and solving instances of Karnaugh Maps, often a time-consuming task for both students and teachers. Solving a K-Map requires to discover a minimal cover of the K-Map using n-cubes. **That is, to find a way, by using as little n-cubes as possible, to cover all of the "1" values in the K-Map[3], without covering any value of the opposite sign**. When "don't care" values are present together with "1" and "0" values, an extra rule is required: the minimal cover may not contain an n-cube with only "don't care" values.

2. Choice of the programming language: Why JavaScript?

Before coding the application, a first choice to be made was the programming language to be used. Four major programming languages were considered: Python, C++, Java, and JavaScript.

- Python is a simple, open source language with many applications in web development. But Python is also a rather high-level language: the compiler manages a lot of tasks instead of the programmer, and builds executable programs that could be too heavy. This problem can nevertheless be avoided if instructions are interpreted one by one, but the language still needs various third party libraries to provide most of the services that this project requires (graphic interfaces, server-side web development, etc.). **Despite these constraints, Python remained one of the best options for this project.**
- C++ is a very powerful and popular language which provides a good basis for compiling fast, light-weighted programs, but which lacks an interpreted option. Its main constraint is the opposite to Python's: the language is too complex and low-level, leaving too many responsibilities on the programmer's hands. In the case of a web application, it would also need many third party libraries. Unless the application's speed were the top priority, **C++ is the least suited for this project, and for web development in general.**
- Java is another popular language that focuses heavily on object-oriented applications. It's relatively professional, yet relatively high-level, lacking C++'s complexity and demanding less responsibility for the programmer (for instance, Java provides a garbage collector). As for Java libraries, most of them are first party and included with the development kit. **Java stands as a good option, although it forces the object-oriented paradigm too much**, making the coding of some simple functionalities too complicated (for instance, just writing "Hello World" in the console already asks for relatively complex coding).
- JavaScript is an interpreted language that, along with HTML and CSS, is one of the core technologies for web development. On the surface it resembles Java very much, but it allows programmers to use Python-like code for more simple functions, without the need to make every aspect of the program fit the object-oriented paradigm. It also doesn't need any packages for simple graphical content (it can directly manipulate HTML and CSS content already on the page), although they can be useful for implementing more complex functionalities. **JavaScript is one of the best alternatives (if not the best one) for simple web applications like this project**, as code can simply be written inside an HTML web page and interpreted in real-time.

[3] That is, if it is the canonical form of the minterms that is sought. For the canonical form of the maxterms the objective is to cover the 0 values.

In conclusion, **JavaScript** was chosen as the preferred language for coding the application. This was because of its relative simplicity, its focus on web programming as the main application field, and its capacity to blend and interact with HTML and CSS content to provide useful results.

3. Choice of the algorithm: ESPRESSO and "pseudo-ESPRESSO"

Another important programming choice was to select a pertinent algorithm for solving K-Maps. Indeed, while the K-Map exercises themselves are created by the user, it is important that their solutions are calculated in an efficient way, using a quick and efficient algorithm.

- The most trivial algorithm would be to iterate over every possible n-cube in the map, from the largest to the smallest, retaining only those that contain just "1" values, and that are not already contained in a previously visited n-cube. This algorithm is in fact too slow, as its complexity grows exponentially with the number of variables of the expression (the dimensions of the n-cube). It is therefore too impractical for solving large K-maps.
- **The Quine–McCluskey algorithm**, also known as the method of prime implicants, ensures the minimization of the Boolean function, meaning that the result is the simplest possible. It operates on the truth table, combining the minterms for which the function is active or its value is irrelevant ("1" or "don't care" values), and then finds the smallest set of prime implicants that cover the function. The method is much quicker than the trivial method, but processing time still doubles with every new variable.
- **The ESPRESSO algorithm**[4] is a heuristic method based on direct observation of the n-cubes rather than their corresponding minterms, and it is generally much faster and simpler than Quine-McCluskey. As most heuristic methods, it is iterative: cubes are first selected by expanding them, until any further expansion covers at least one "0" value. Then redundant cubes are removed leaving only a subgroup of essential cubes. The solution obtained on this kind of run, while close to it, is not always the global minimal solution (hence the heuristic aspect), but by following different expansion paths and comparing the number of cubes, eventually one should find the minimal cover.

Because of its superior efficiency both in time and in memory usage, and its simplicity in terms of code, **the ESPRESSO algorithm** was chosen as the most appropriate basis for implementing the application's K-Map solving functionality. However, the actual ESPRESSO method was not used as such, but instead a similar approach (which will be called "**pseudo-ESPRESSO**" from now on) was taken.

The main reason for developing pseudo-ESPRESSO was to remove the iterative aspect of ESPRESSO (and, in some way, its heuristic aspects), and develop an equivalent deterministic method, so as to provide more stable and optimal results. The main difference between both methods is in the execution phases of both algorithms: ESPRESSO repeats three phases iteratively (Expand, Irredundant Cover, and Reduce), while the pseudo-ESPRESSO method only needs to execute the two first phases at once (Expand and Irredundant Cover):

[4] Source: http://www.physics.dcu.ie/~bl/digi/unitd17.pdf

ESPRESSO begins its first Expand and Irredundant Cover phases by generating a "reference" cover: it makes a set of n-cubes grow as much as possible (Expand), following a random growth path, and then keeps only the n-cubes that are essential for covering all the 1 values (Irredundant Cover). ESPRESSO then improves on said "reference" cover iteratively (fig. 2), by reducing again the size of the remaining cubes (Reduce) and exploring a slightly different expansion path (Expand and Irredundant Cover). If any newly found cover has less n-cubes than the "reference", that cover becomes the next reference. This continues until all of the expansion paths have been explored, or until a maximum execution time expires (for a heuristic result).

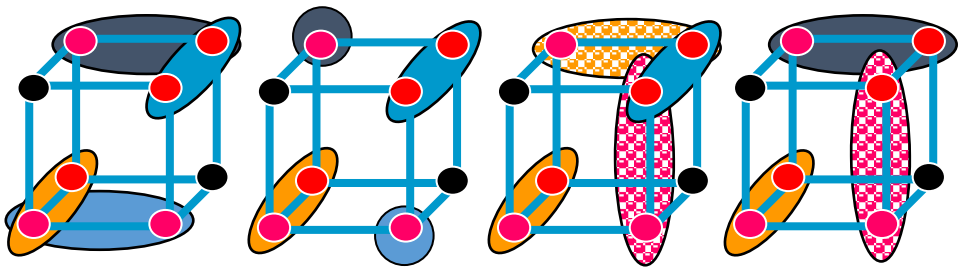

Fig. 2: Iterative steps of the ESPRESSO algorithm, and their effect on the n-cubes. From left to right: A Reduce step makes some of the n-cubes smaller. The next Expand step makes them grow again through a different path. A cube then becomes irredundant.

The pseudo-ESPRESSO approach replaces ESPRESSO's randomized iterative process with an equivalent deterministic approach. All possible expansion paths (and all n-cube sizes) are explored at once in a single Expand phase, after which any n-cubes contained within other cubes are eliminated from the cover. This leaves a cover with only the biggest possible n-cubes, on which a single Irredundant Cover step is applied to select only the cubes that are relevant. Since all possible sizes are explored at once, and only the biggest are kept, the result is always optimal. No further iterations (or a Reduce phase) are needed.

4. Implementation of the program

The JavaScript web application is built upon an HTML hypertext document (a website), which serves as a frame for displaying the K-Map and the solution in real-time, as well as a user interface for modifying the K-Map's content and some parameters. Indeed, the user may utilize the HTML document to modify global variables such as the number of logic variables (`numVar`), the appearance of "don't care" values (`allowDC`), and the content of the K-Map itself (`KMap`). Other global variables, such as the lists of the covered spaces (`coverList`) and of the n-cubes (`nCubeList`), are also indirectly influenced by the user's actions.

Concerning constant properties, the program keeps the names for each logic variable (`varNames`), the dimensions of the K-Map in depth (`KLvl`), width (`KWid`), and height (`KHei`) corresponding to each number of variables, as well as how many of these variables will appear horizontally (`KVarX`) and vertically (`KVarY`), and the Gray code order for the cell spaces (`bitOrd`). The program begins by using these constants for initialising the K-Map, using the function **`initKMap`**. This function is very important, as it is also called any time that the K-Map is reset (for instance when the number of variables is changed). The other most important function in the code is **`redraw`**, as it activates the real-time generation of the K-Map's new layout, the new solution, and the way it is reflected in the HTML document.

The `redraw` function is in fact mostly composed of three calls to different functions, one for each element of the interface:

- **`generateKMapHTML`** produces the HTML code for displaying the K-Map as a matrix of clickable buttons, each one labelled with the value of its corresponding cell space;
- **`generateSolutionHTML`** initiates the "pseudo-ESPRESSO" algorithm (`EspressoSolve`), and then uses the solution to generate an interactive HTML canonical function (`getFunctionHTML`), which is linked to the K-Map's representation, more specifically to the n-cubes;
- **`generateLaTeXCode`** writes the exportable LaTeX representations of both the K-Map and the solution into a text box, and has its own methods for building the canonical function (`writeLogicFunction`), and displaying the n-cubes (`writeNCubes`).

The functions that are related to the "pseudo-ESPRESSO" algorithm, such as `EspressoSolve`, `EspressoExpand`, and `EspressoIrredundantCover`, behave mostly as explained in the previous section. Two remarks must nevertheless be made about specific cases, for what the original ESPRESSO was not foreseen: K-Maps with more than 4 variables, and K-Maps allowing "don't care" values.

For cases with 5 variables, as mentioned before, we can consider a 3-dimensional K-Map, with the fifth variable represented as a third dimension (the depth) in addition to the map's width and height. This can be seen in the usual representation of the 5-variable K-Map, which is two 4-variable maps side by side, representing two states of the fifth variable, or two layers of depth. The actual ESPRESSO algorithm could exploit this interpretation by using the third dimension as an additional growth direction for its Expand phase paths, but in "pseudo-ESPRESSO" we need to explore all the paths at once with an efficient method.

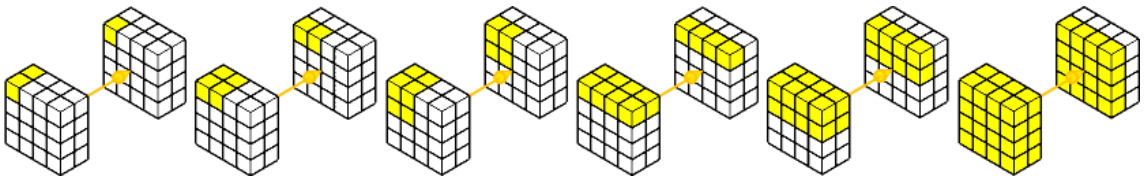

Fig. 3: When considering the fifth variable as "depth", there is a 1-to-1 correspondence between n-cubes covering two layers of depth, and n-cubes covering a single layer of depth.

A simple solution (fig. 3) is to consider a 1-to-1 correspondence between n-cubes that cover a single state for the fifth variable (a single layer of depth) and n-cubes that cover both states (both layers). This means that the algorithm can be normally run on each layer separately, and then consider n-cubes on both layers as if it were exploring a third layer (which corresponds to the logic AND of the two other layers).

For cases with "don't care" values, to keep the cover optimal, additional mechanisms had to be added to the two main phases of the algorithm.

- During the Expand phase: Every stage of growth from a specific n-cube is evaluated: an n-cube may not be a candidate, but its next expansion may be one. An n-cube is considered a candidate to be part of the cover only if it includes both no "0" values, and at least one "1" value (not only made of "don't care"). When all candidates are found, the algorithm, as usual, eliminates from the cover those n-cubes found inside other n-cubes.

- During the Irredundant Cover phase: Usually, to save computation time, an n-cube is only considered irredundant if the covers with and without the n-cube are identical (the same spaces are covered). With "don't care" values, if the covers are not identical, the algorithm explicitly runs through the entire K-Map, checking if any "1" values have been left outside the cover.

The Espresso algorithm fills up the list of covered spaces (`coverList`) as an array of coordinates, and the list of n-cubes (`nCubeList`) as an *array of arrays* of coordinates, where each array of coordinates represents an n-cube. Functions such as **`getFunctionHTML`** and **`getFunctionText`** use each of these n-cubes to provide the terms of the canonical logic function that constitutes the solution (respectively as an interactive HTML element or as a LaTeX-exportable string of text). These function terms are each built from the logic input leading to the first space of an n-cube (which input variables must be active and which must not). Then the cube's other spaces are visited, and if one of the input variables has to change for that space, then the variable is eliminated from the term (its state doesn't matter for that term's activation).

One final comment for function **`writeNCubes`**, which generates the part of the exportable LaTeX code that draws the n-cubes onto the K-Map: even though the code uses a package (askmaps) specifically designed for presenting K-Maps, the representation of the **toroidal aspect** of the map is quite difficult. Any n-cubes that cross out of the K-Map and then reappear on the other side must be coded as two different boxes drawn into the diagram. A coding trick is used to detect this kind of n-cube: the first space that the program stores in an n-cube's array of spaces is always the top-left space, and the origin point of the map's coordinates is on it's top-left corner. This means that if there is an n-cube that contains a space with coordinates (height of width) lower than those of the cube's "first" (top-left) space, it must be because it exploits the toroidal geometry in some way, and it must appear accordingly in the LaTeX code.

5. Use of the application

The web application is used through the interface presented in the HTML document. Said interface (fig. 4) contains an interactive Karnaugh Map, and an interactive logic function that is generated in real time. Some extra buttons below the main interface present options to reset the K-Map (that is, to set all values to "0"), to change the number of logic variables (allows for K-Maps from 2 to 5 variables), and to allow the K-Map to include "don't care" symbols among the possible values.

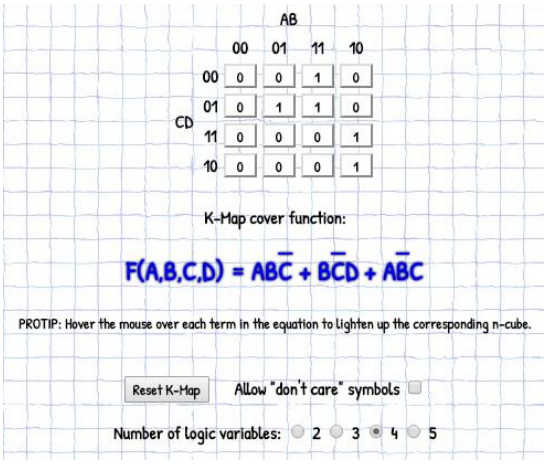

Fig. 4: The web application's interface contains an interactive Karnaugh Map and function, together with some extra buttons to reset the K-Map or change some of the problem's parameters.

To use the interactive K-Map, one may click on the buttons representing the map's spaces, and toggle their content between "0", "1", and (optionally) "don't care" (represented by a "-"). As the K-Map is being built, the interactive logic function underneath it changes in real time. At any moment, if the mouse is hovered over one of the terms of the function, the spaces covered by that term's corresponding n-cube will be highlighted in yellow.

Under the interface, the user may find two text boxes. The first one contains an exportable LaTeX (askmaps) code that represents the K-Map, with its n-cubes marked with dash boxes of different colours. The second box contains a header to be used in the LaTeX document, including commands to implement required packages and colours. It must be noted that the LaTeX package used to display the K-Map, askmaps, was created by a third party, and must be downloaded as a resource for the LaTeX document.

6. Conclusion

In conclusion, it has been developed a lightweight web application that allows users to create Karnaugh Map exercises, and provides their solutions in real time. This application was implemented using an efficient, newly developed algorithm (pseudo-ESPRESSO) that provides stable and optimal results. This could be very useful for both students wishing to train themselves, and teachers in search for an easy and quick way to create new exercise sheets. Both the exercises and their solutions are exportable to LaTeX, using the askmaps package. This application runs within an HTML document, which could be uploaded online as a web page, for the benefit of electronics and engineering students and teachers worldwide.