Quantum Polynomial Certification of Non-Hamiltonian Graphs

Edward Haeusler Lorenzo Saraiva Robinson Callou

January 2025

Abstract

This article presents a novel approach to verifying the non-Hamiltonicity of graphs by combining standard proof theory and quantum computing. Starting from a non-Hamiltonian graph G, we construct a formula α_G provable in minimal implicational logic. From this, a natural deduction proof Π is generated, which is then transformed into a directed acyclic graph (DAG) proof ∇. This DAG proof is converted into a logical circuit C_∇, which outputs 1 (true) for every input boolean path vector, verifying the non-Hamiltonian property. By applying the Deutsch-Jozsa quantum algorithm to C_∇, we efficiently obtain a quantum polynomial certificate that verifies with 100% probability that G is non-Hamiltonian. This method bridges proof systems and quantum algorithms, offering a quantum advantage in graph non-Hamiltonicity verification.

1 Introduction

Proof verification is a fundamental challenge in formal logic, computational complexity, and automated reasoning. As proofs become more complex, verifying their correctness efficiently becomes increasingly difficult. Traditional proof systems rely on logical frameworks and combinatorial structures, but for large-scale proofs such as those proving the non-Hamiltonicity of a graph—direct, verification can be computationally expensive. This has motivated the search for alternative verification methods that make use of structured computational models and quantum algorithms.

This paper focuses on the problem of certifying non-Hamiltonian graphs using formal proofs and quantum computing techniques. A graph is Hamiltonian if it contains a cycle that visits every vertex exactly once. Proving that a graph is not Hamiltonian is significantly more difficult than proving

its Hamiltonicity, as it requires demonstrating that no such cycle exists for any possible traversal. Verifying these proofs efficiently remains a major challenge in computational complexity.

Our approach combines formal proof verification, circuit-based representations, and quantum computing to develop a more efficient method for non-Hamiltonicity certification. First, we analyze how DAG-based proofs encode non-Hamiltonicity and explore how these proofs can be compressed and transformed into structured Boolean representations. We then introduce a computational model that maps the proof into a structured framework. Finally, we show how the Deutsch-Jozsa algorithm can accelerate certification by leveraging quantum superposition and interference to verify the validity of a proof certificate in polynomial time.

This paper is organized as follows: We begin by discussing the theoretical foundations of Hamiltonicity and proof verification, reviewing existing methods for proving non-Hamiltonicity. Next, we present the computational formulation of the problem, explaining how classical proofs can be represented using circuits. Finally, we explore quantum polynomial certification, demonstrating how quantum computing can enhance efficiency in verifying non-Hamiltonian graphs. This work contributes to the broader discussion on proof compression, computational complexity (NP vs PSPACE), and the intersection of quantum computing with formal logic.

2 Previous works on Compressing Tree-like Natural Deduction Proofs

In a series of articles, [GH19, GH20, GH22b] L.Gordeev and the first author provided a proof that $NP = PSPACE$ based on the transformation of tree-like Natural Deduction proofs/derivations for purely intuitionistic minimal logic ($M_{\supset}$) into Direct Acyclic Graphs (Dags). This transformation identifies equal occurrences of formulas at the same level of the tree. Identification is given by collapsing the nodes with the same label formula, and, at the same level. The collapse changes the tree-like derivation to preserve information about the logical consequence after the collapse. It turns out, that the tree is transformed into a Dag. Since the publication of [GH19], many readers of it demanded a computer-assisted proof of the main result due to its underlying combinatorial structure being hard to follow. The size of the entire collapsed (Dag-like) proof is upper-bounded by a polynomial on the conclusion of the proof whenever the set of formula occurrences in the original tree-like proof and the height of the tree-like proof also is. Using

[Hud93], the article published in Studia Logica, [GH19], closes the requirements on a polynomial bound on both the height and the number of formula occurrences in any ND tree-like proof of a tautology in $M_{\supset}$. Demands of some readers about the need to read another proof-theoretical paper to understand a proof of a result on computational complexity moved the research to a further improvement. In [GH22b], we drop out the need of [Hud93] for the special case $NP = CoNP$. We show that the horizontal compression can be applied directly to a relevant and specific class of height and formula occurrences poly-bounded Natural Deduction proofs in $M_{\supset}$, namely the class of proofs of non-hamiltonicity for (non-Hamiltonian) graphs. Thus, [GH22b] shows a proof o $CoNP = NP$ that does not need Hudelmaier linear bound. Finally, in [GH22a] we can find an overview of these proofs that proposes a non-deterministic approach that takes into account the Dag-like proofs without the need for a deterministic way of reading it. It should be observed that many colleagues and readers have asked to prove most of, maybe all, the results in this report in an Interactive Theorem Prover. The first and the third author of this report are working in the formalization of part of results in [GH22a], as for example, the fact that part of the compression rules can comprehend an algorithm that compress any Natural Deduction proof of an implicational tautology with n types of labeling formulas and height polynomially bound by n into a dag-like derivation with a polynomially sized kernel, i.e., the internal nodes of the dag-like proof, or certificate. This is proved in [dMBF25]. using the $L\exists\forall N$ proof assistant.

We start the explanation of our compression algorithm by showing an example of the **HC** compression on a very simple[1] and, we hope, illustrative example. This is the content of the following section 3. Our logical language is restricted to the implication. The logic is the purely implicational propositional logic. We use the symbol $\supset$ for the implication to avoid confusion with the $\rightarrow$ symbol, largely used in graph pictures and representations. As a matter of fact, many readers and colleagues ask for a **Computer Assisted** proof of this result, since it is based on the algorithmic application of a set of rules. We reinforce that such assisted proof is under development as already mentioned in the previous section. The reader should have a basic knowledge of Natural Deduction and proof-theory to appreciate better what we convey here. It is interesting mentioning that in [Hae22] there is a proof-theoretical argument that provides a good explanation of reasons for the excellent compression rate of HC when compared to traditional benchmarks in terms of string compression.

[1] The simplest, in fact

3 A Small Example

We say that a graph $\langle V, E\rangle$ is Hamiltonian, if and only if there are $v_i, v_o \in V$ such that there is a path from V_i to V_o visiting every node in V, exactly once. The graph in figure 1 is not Hamiltonian. In figure 2, we have a Natural Deduction (ND) proof of the non-Hamiltonicity of this graph.

Figure 1: The smallest non-Hamiltonian graph

Using the classical mapping that takes a graph $\langle V, E\rangle$, with $n = card(V)$, to a formula α_G, written in the propositional logic language, $v^i, i = 0, n-1$, such that α_G is satisfiable, if and only if, G is Hamiltonian. α_G is the conjunction of the following formulas for the graph G in figure 1:

$$(a^0 \vee b^0) \wedge (a^1 \vee b^1) \quad \text{(a)}$$

$$(a^0 \supset \neg a^1) \wedge (a^1 \supset \neg a^0) \wedge (b^0 \supset \neg b^1) \wedge (b^1 \supset \neg b^0) \quad \text{(b)}$$

$$(a^0 \supset \neg b^1) \wedge (b^0 \supset \neg a^1) \quad \text{(c)}$$

By the definition of Hamiltonian path, the number of nodes is equal to the number of steps. Each node v is mapped to the propositional letter v^j, where, $j = 0, n-1$, where n is the number of nodes in V. The letter v^j means that v is visited in the j-th step of the path. According to this meaning, in the conjunction above, the formula in line **a** specifies that one of the nodes must be visited at each step. The formula in line **b** specifies that no node can be visited more than once. Finally, the formula in line **c** specifies which nodes cannot be visited in the next step for each node and step. For example, if at step 0 we visited the node a and there is no edge going out of a into b, then b cannot be visited at step 1. This is specified by the formula $a^0 \supset \neg b^1$. Note that the other part of the formula in line **c**, i.e.,$b^0 \supset \neg a^1$ takes care of the similar specification regarded to node b, instead of a.

α_G is the formula that is the conjunction of the three lines above. It is classically satisfiable whenever G is Hamiltonian. Thus, if G is not Hamiltonian, then α_G is unsatisfiable, and hence, its negation, i.e., $\neg\alpha_G$ is a tautology. Thus, if G is not Hamiltonian, then $\neg\alpha_G$ is provable. Figure 2 , below, shows a proof of $\neg\alpha_G$ in the minimal propositional logic. The double inference line presents a more concise version of the proof. It abbreviates

the deduction from $\neg\alpha$ to each of the different conclusions, which otherwise consists of sequences of $\wedge$-Elim steps to derive each one on this naive proof of non-Hamiltonicity of a graph derived from the decision tree analysis.

We observe that $\neg\alpha_G$ is a minimal logic tautology for any non-Hamiltonian graph G. This is a consequence of Glyvenko theorem, i.e., if $\vdash_{Cla} \neg\alpha$, then $\vdash_{Int} \neg\alpha$, and the pattern used in the proof, derived from the decision tree for G that represents all possible visiting paths, showing that all of them are not Hamiltonian. See [GH22b] for a more detailed reading.

Report [**?**] presents an effective proof compression method based on a set of rewriting rules. It transforms any Natural Deduction proof Π in minimal implicational[2] logic into a DAG-like proof of the same conclusion, preserving the original set of hypotheses. This DAG-like proof is represented using a mathematical structure called a DLDS (Dag-Like Derivation Structure). Figure 4 illustrates the DLDS corresponding to the proof of $\neg\alpha_G$ shown in Figure 1. We have to note that the proof Π should be translated to the implicational logic. This is explained in detail in [Hae14]. The propositional letters p_0 and p_1 abbreviate, respectively, $p_{a^0 \vee b^0}$ and $p_{a^1 \vee b^1}$ from the original translation in [Hae14][3]. Again, the details can be found in [Hae14]. In figure 3, we have the implicational translation of the proof in 2. The formula α_G^* is the implicational translation of α_G.

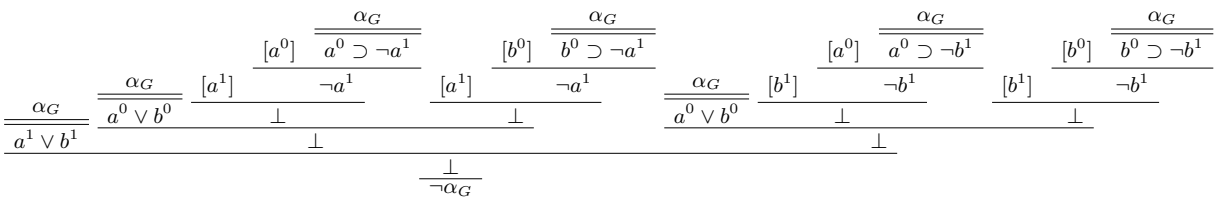


Figure 2: A naive proof that graph 1 is not Hamiltonian

[2]Implicational logic uses only the connective $\supset$ to form formulas.

[3]The translation in [Hae14] is inspired in [Sta79]

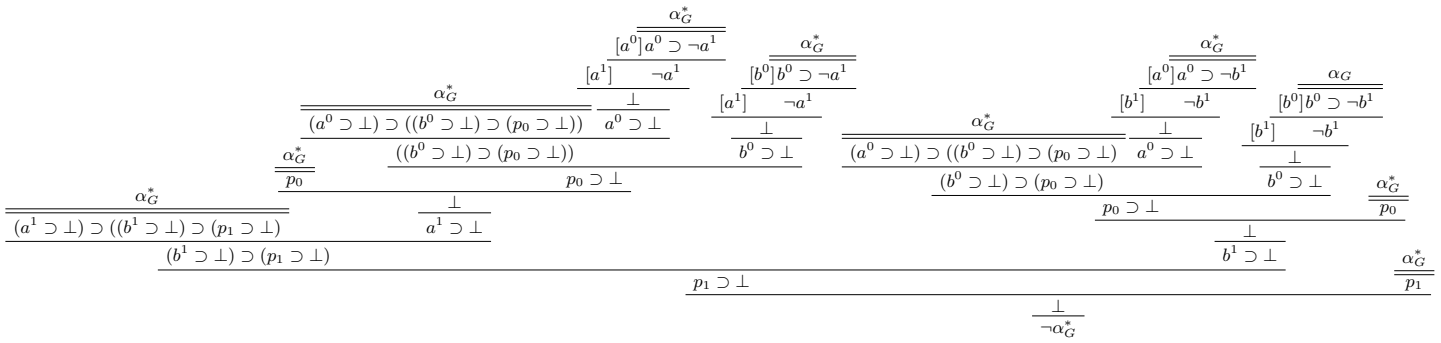


Figure 3: The implicational version of figure 2

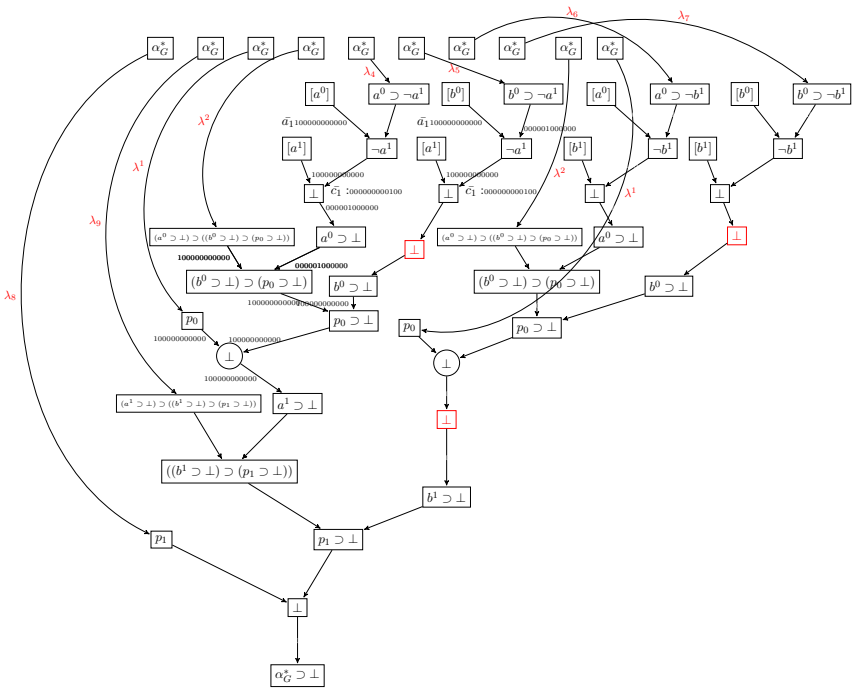


Figure 4: Tree-like DLDS proving $\alpha^*_G \supset \bot$

In figure 4 we show the tree-like *DLDS* representation of the Natural Deduction proof of $\neg\alpha^*_G$ that appears in figure 3. To have a clearer drawing we omitted the elimination rules applications associated to the doubled-bars inference in 3. We use the labels λ_i, $i = 1, \ldots, 9$. We repeat labels in the case they represent the same sequence of rule and formulas in the proof. We do not show always the dependence sets, for reasons of better visualization

of the drawings too.

In figure 12, we have applied all the MUE **HC** compression rules, but the last ones related to the elimination-part having α_G^* as top-formula, to the tree-like $DLDS$ depicted in figure 4. The displayed Dag-Like proof in figure 12 is the (almost final) result of the MUE compression, the collapse of the node labeled with α_G^* is not there yet. It is worth noticing that **HC** is an algorithm, not a set of MUE rules applied whenever possible in any order. The **HC** compression works from bottom-up and left-to-right, collapsing the nodes with the same labelling formula in each tree-like and dag-like proof level. Figure 13 represents the final compression $DLDS$ where only one occurrence of the node labeled by α_G^* happens.

Due to the scope of this article, we chose a small and graphically manageable example to show the result of the **HC** compression algorithm. The degree of compression for such small proof is not very high. To make the example more illustrative, we used a repetition rule, a technical device used in the original article [GH20], but that is not needed for **HC**, in general. Its use here is only to have a more compressible proof example. The conclusion of this rule is coloured in red in the DLDS in figure 4. In Natural Deduction this rule would be represented as:

$$\frac{B}{B}$$

Here B is any formula, and the inference is "from B we can derive B". The only purpose in our case is to put the premiss of this repetition rule application, the $\bot$, at the same level as the $\bot$ at the left branch that concludes the major premiss $(a^1 \supset \bot) \supset ((b^1 \supset \bot) \supset (p_1 \supset \bot))$. A similar pattern is found concerning the eliminations with $(a^0 \supset \bot) \supset ((b^0 \supset \bot) \supset (p_0 \supset \bot))$ as major premiss.

Figures 4, 5, 6,7 and subsequently until 12 illustrate the steps of applying the **HC** compression algorithm to the DLDS depicted in Figure 4. In each figure, the ellipsed nodes appear collapsed and filled (shaded) in gray in the subsequent figure. The ancestor edges are shown in blue, with their labels in red.

Except for the first figure, each subsequent figure highlights the nodes collapsed at the current level in grey, while the nodes scheduled for the collapse in the next step are enclosed in ellipses at the upper level. Notably, since **HC** collapses nodes in pairs, each figure represents a greedy step in which all formulas with the same label (or formula) at a given level collapse simultaneously. A simultaneous collapse is effectively achieved through a sequence of pairwise collapses. Moreover, we can prove that the collapsing

process is associative concerning pairwise collapsing, allowing us to abstract this property here.

From figure 12 on, we jump directly to the result of the collapse of all nodes, due to the Moving Up Edges applications that is shown in figure 13. We omitted the details of collapsing the subderivations represented by each of the edges labeled by λ_i, since they would only turn the drawing quite hard to read.

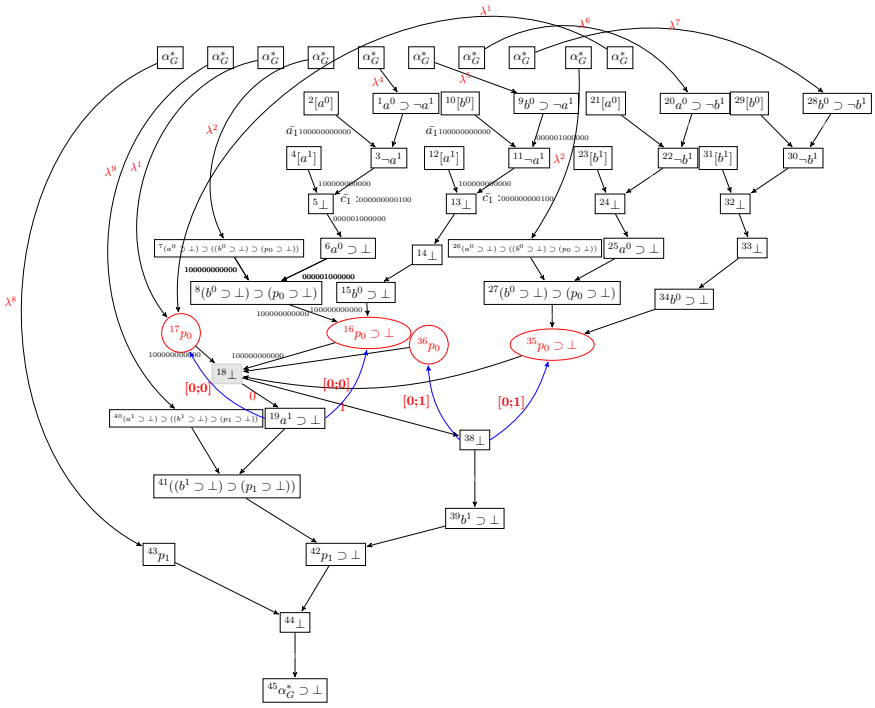


Figure 5: DLDS proving $\alpha^*_G \supset \bot$, with the 5th-level $\bot$-labeled nodes collapsed, i.e., those circled , in figure 4. The collapsed node is the gray one

The common rules and formulas that would be possible to collapse among the sub-derivations that are not named with the same λ will be not taken to be collapsed. The last and final compressed derivation has only one node labeled with α^*_G.

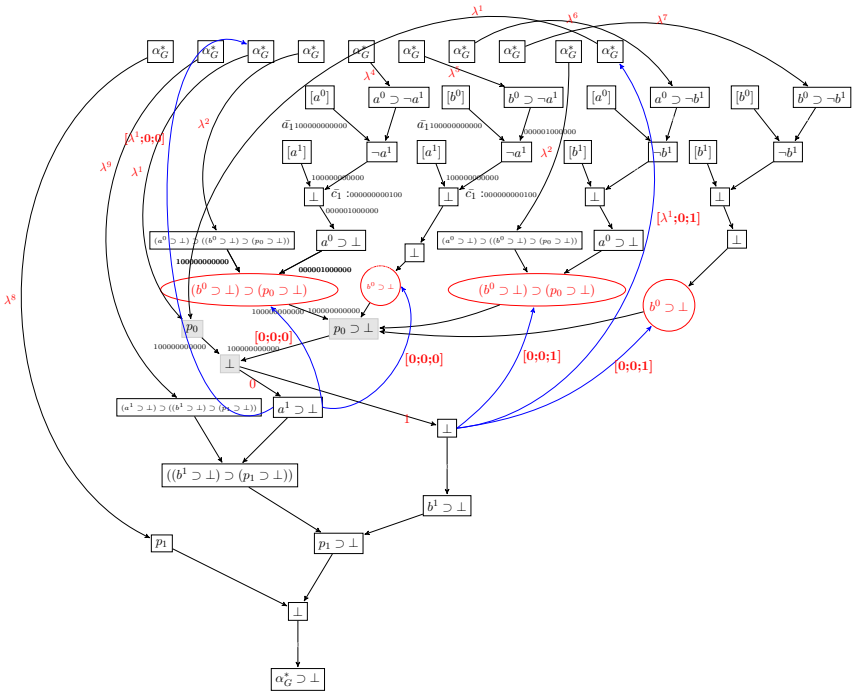


Figure 6: Proving $\alpha_G^* \supset \bot$, with 6th-level $p_0 \supset \bot$ and p_0 labeled nodes collapsed, i.e., the circled and ellipsed nodes, in figure 5 resp.

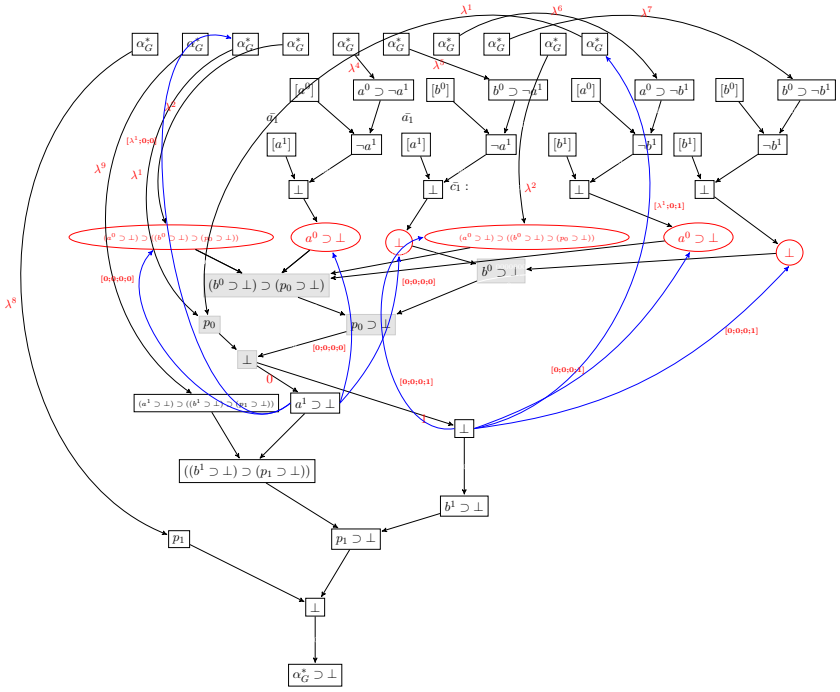


Figure 7: Proving $\alpha^*_G \supset \bot$, with 7th-level $(b^0 \supset \bot) \supset (p_0 \supset \bot)$ and $b^0 \supset \bot$ labeled nodes collapsed, i.e., the circled and ellipsed nodes, in figure 6 resp.

4 Boolean Circuits and the Smaller Example

After applying the HC algorithm, the compressed DLDS is encoded in a Boolean circuit. This circuit must accurately represent the DLDS and output 1 if the encoded DLDS is a valid proof; otherwise, it should output 0. To do so, the DLDS is expanded into a more generalised graph. This graph will have N^2 nodes arranged in a square $N \times N$ grid, where N is the number of unique clauses in the DLDS. Figure 19 shows an example of such a graph. The input of the Boolean circuit will be a *path*, which will represent which edges of the graph will be active, going from left to right. The path comprises at least N sub-paths, one for each of the leaves on the left, and possibly more, in the case that a unique clause is used more than once on the top level of the DLDS. Each path has N steps, each step having $logN$ bits that represent the indexes 1 to N of the edge that will be used in a specific step. The 0 is utilized when none of the outgoing edges of a node is

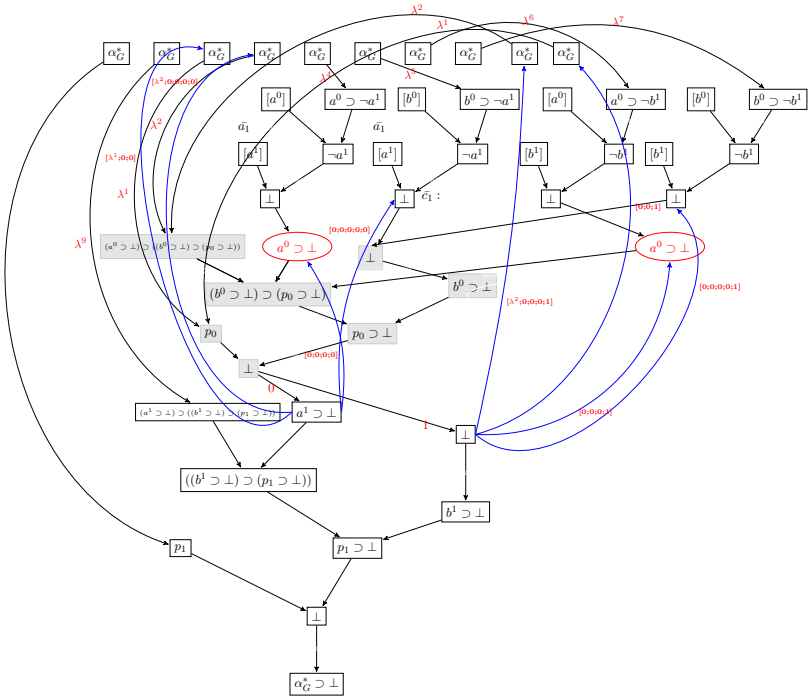


Figure 8: Proving $\alpha^*_G \supset \bot$, with part of the 7th-level, i.e, $(a^0 \supset \bot) \supset (b^0 \supset \bot) \supset (p_0 \supset \bot)$ and $\bot$ labeled nodes collapsed, i.e., the red circled and one of the ellipsed nodes, in figure 7 resp.

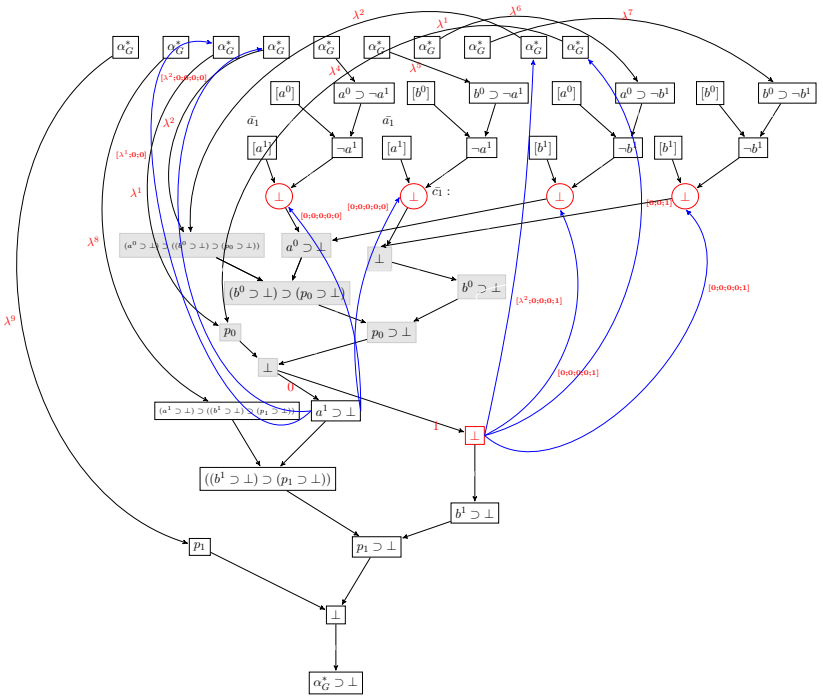


Figure 9: Proving $\alpha^*_G \supset \bot$, with 7th-level collapse, the $a^0 \supset \bot$ ellipsed labeled nodes in figure 8 already collapsed

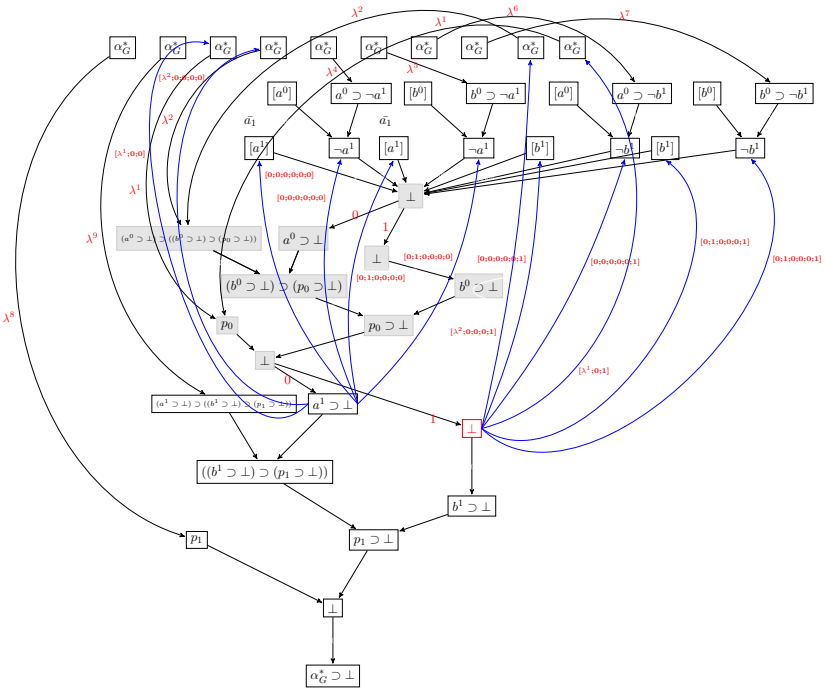


Figure 10: Proving $\alpha^*_G \supset \bot$, with 8th-level collapse, the $\bot$ red ellipsed labeled nodes in figure 9 already collapsed

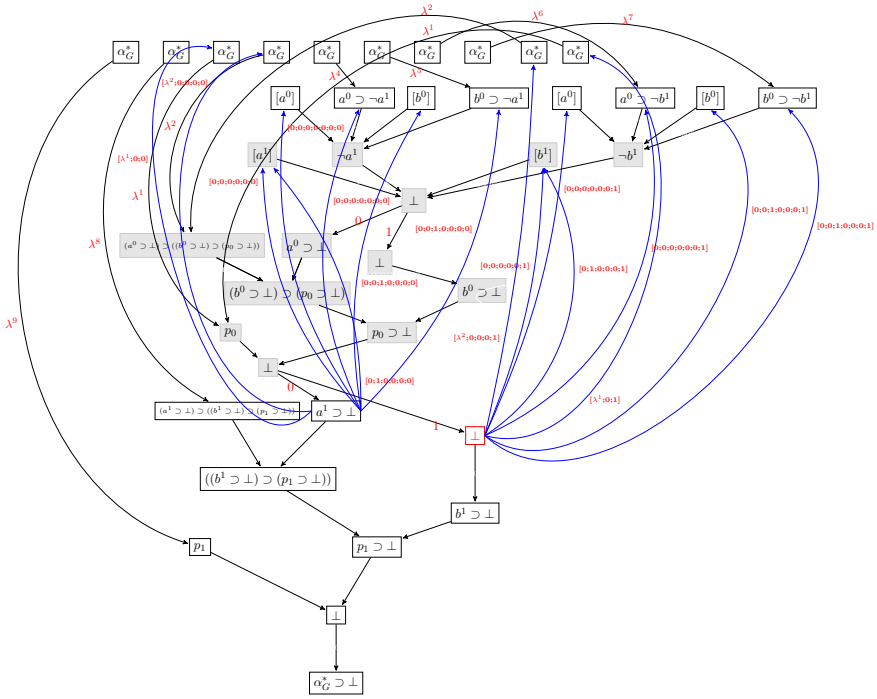


Figure 11: Proving $\alpha^*_G \supset \bot$, with 9th-level collapse, the nodes labeled $[a^1], \neg a^1, [b^1], \neg b^1$ in figure 10 already collapsed

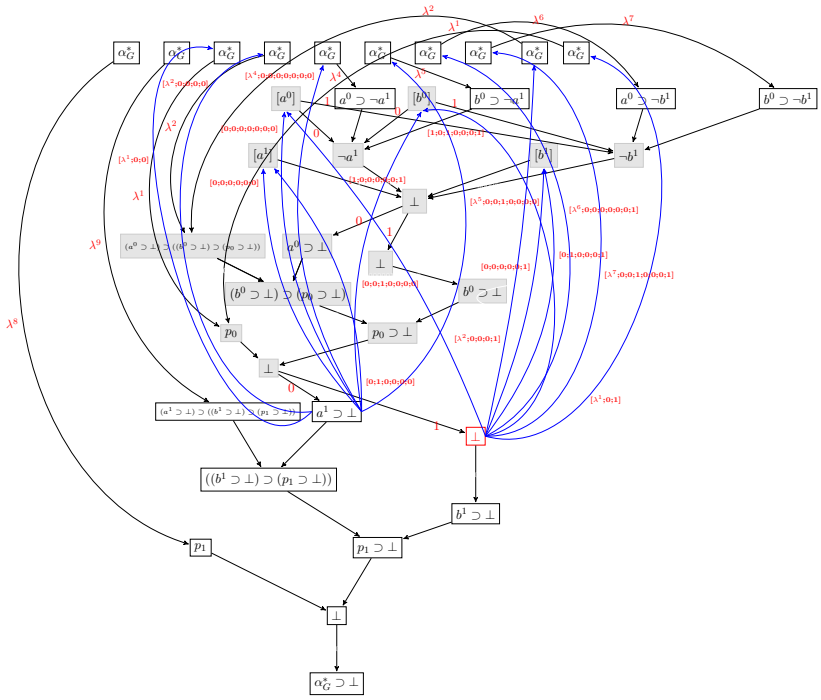


Figure 12: Proving $\alpha_G^* \supset \bot$, with almost all-levels collapsed

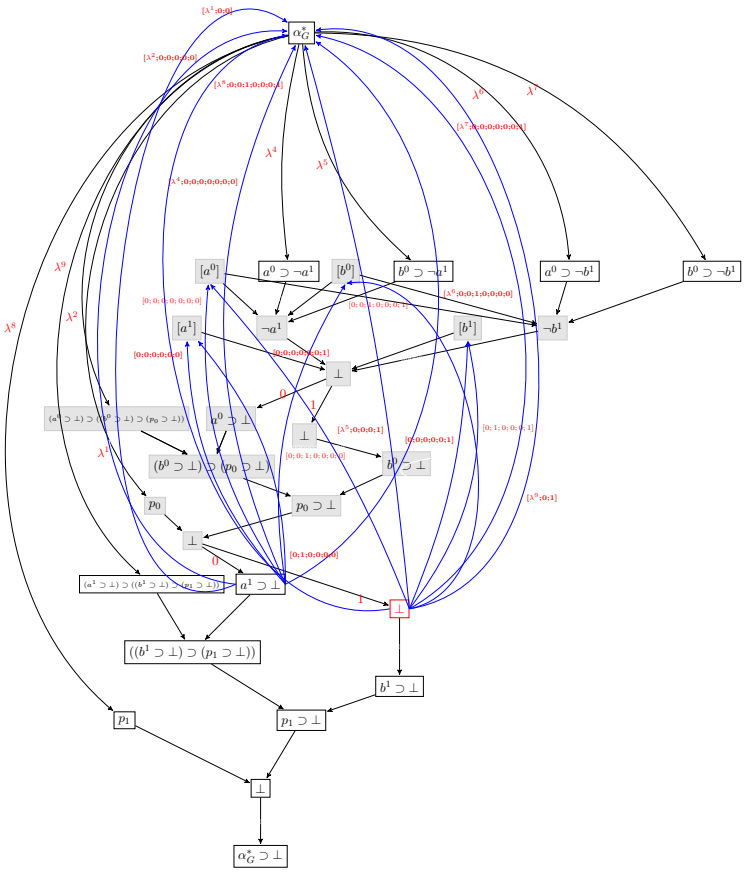


Figure 13: Proving $\alpha_G^* \supset \bot$, with all-levels collapsed

active. These active edges are interpreted as applications of Natural Deduction rules. Each combination of paths generates a subgraph of the original $N \times N$ fully connected graph, which is then executed as a Boolean circuit. If a DLDS is valid, then, for every possible path, the circuit will output 1. There are two possible ways for the circuit to output 1:

- Every node is generated by an application of a valid imp-intro or imp-elimination rule, using as premises the previous nodes that have active edges to the target node, and the final dependency set is all zeroes.
- The generated subgraph is invalid, that is, not all nodes are accurately generated by an application of the rules.

Thus, the circuit will output 0 in the cases that the generated subgraph is valid, that is, the ND rules are respected at every step, but in the end the dependency set is not zero, meaning that this graph is not a complete proof.

For our motivating example, depicted in figure 13, the boolean graph shows up in figure 14

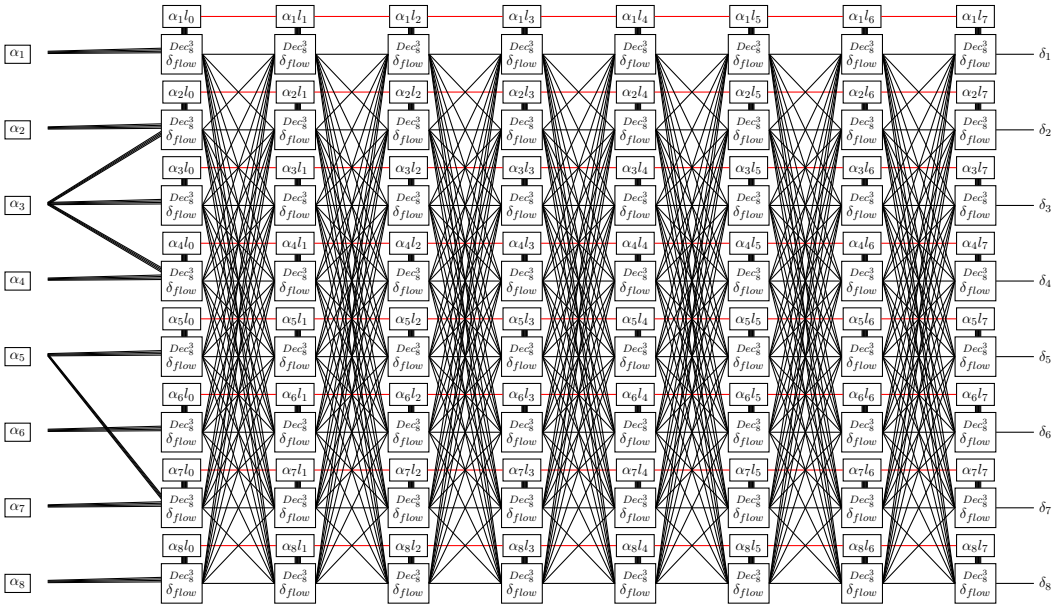


Figure 14: The higher-level circuit corresponding to the *DLDS* proof in fig 13

Each node in the DLDS is associated with a circuit similar to the circuit in figure 18. For each node, there are atmost N wires as input and N wires as output. Each wire represents either an *Introduction* or one of the premises of an *Elimination* rule, and the rule application must produce the clause of the node itself. For each node, we have the following properties:

- Each rule has N dependency bits.
- An *Introduction* rule has a single activation bit.
- An *Elimination* rule has two activation bits.
- Each rule has an associated dependency vector of size N
- The output is the associated dependency vector if exactly one rule is active.
- If no rule or multiple rules are active, the output is an all-zero vector of size N.

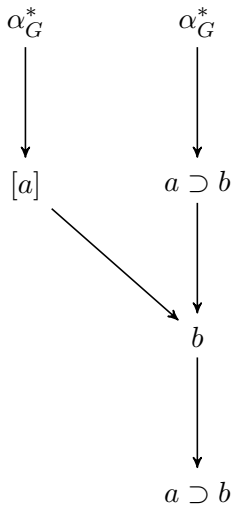
In order to achieve this, we first compute the activation bits of each rule. That is, we check, for the input wires, which are included in the *path* inputs, meaning that the wire is active. In figure X, we can see a detailed representation of a single node in the circuit. The possible rules are represented in the top; those rules come from all the possible combinations of the input wires that, when applied, generate the clause contained in the node.

In order to achieve this behavior, we have the following steps.

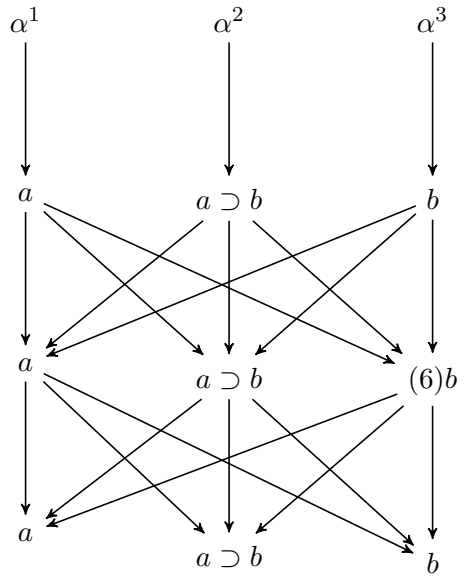
1. Compute the activation bits of all the rules. In the case of the Introduction rules, this is the bit itself; in the case of the Elimination rules, it performed an AND operation. This results in N bits $a^1, a^2, ..., a^n$ associated with each rule.
2. Perform a multiple XOR operation between all the activation bits to check if exactly one rule is active, resulting in the bit x^1.
3. Compute an AND operation between the multiple XOR result x^1 and each activation bit, resulting in the N bits $b^1, b^2, ..., b^n$.
4. Compute an AND operation between $b^1, b^2, ..., b^n$ and its corresponding dependency vector.

5. Compute an OR operation between the results of the previous step. This is the final output value.

The implementation and verification of this circuit have been carried out using the Lean 4 proof assistant. The circuit operates on a set of N possible input groups, where each group represents the rules with associated dependencies and activation bits. We will cover an even smaller example to better explain, explicitly describing the circuit's construction. We will start with the following proof of Natural Deduction:



From this proof, we first build the NxN graph, where N is the number of unique clauses in the original proof. In this case, $N = 3$, so we have the following graph:



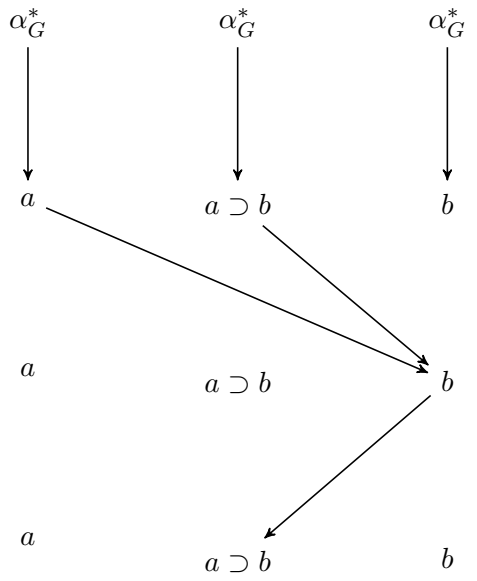
Each *path* will mark which of these edges are active (that is, carrying the true value), thus creating a subgraph. The original graph will always be one of these path options. For the algorithm to return true, every path should either be a complete proof or a invalid graph. In the paths, the 0 represents that no edge from the node is on, so the indexes start at 1. When to paths converge (O QUE FAZER NESSE CASO?). In this case, the paths that represent the proof are:

$$Path^1 = 3, 2$$

$$Path^2 = 0, 0$$

$$Path^3 = 3, 2$$

This generates the following subgraph, which is a valid proof;



For each node of this graph, a circuit similar to figure (STABLE) will be constructed. Let us detail this circuit for each of the nodes in the example. For each node, the rules that are present in the circuit are the rules that will actually generate the value. Let us use the node(6)b as an example. In order to generate b utilizing the available clauses, we can apply the $\supset$-Elimination rule with a as the Minor Premise and and $a \supset b$ as the Major Premise. Then we have the remaining wire of b itself, which will represent the repetition rule. In the case that a specific wire has no possible application in a node, the wire will lead directly to the error.

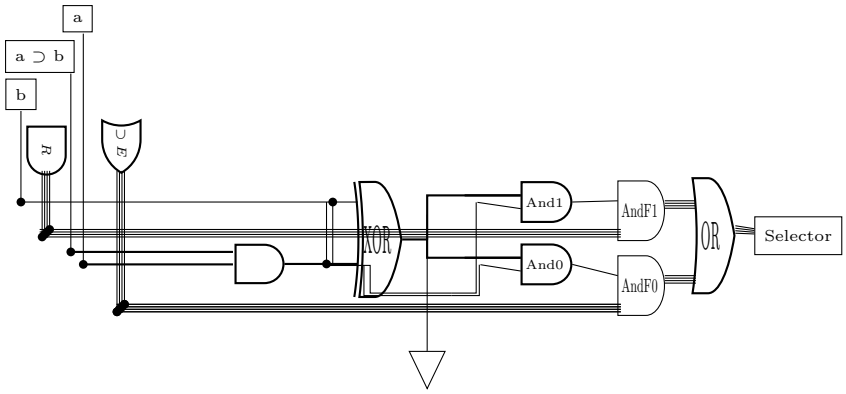


Figure 15: Circuit for the node b

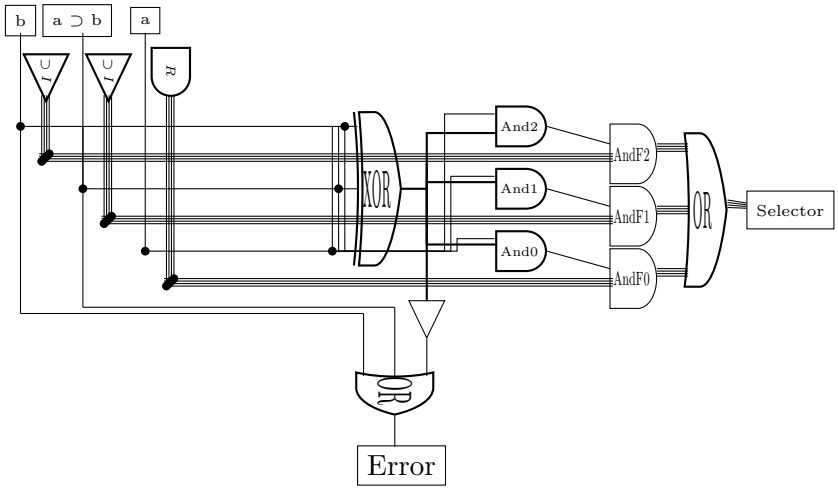


Figure 16: Circuit for the node a

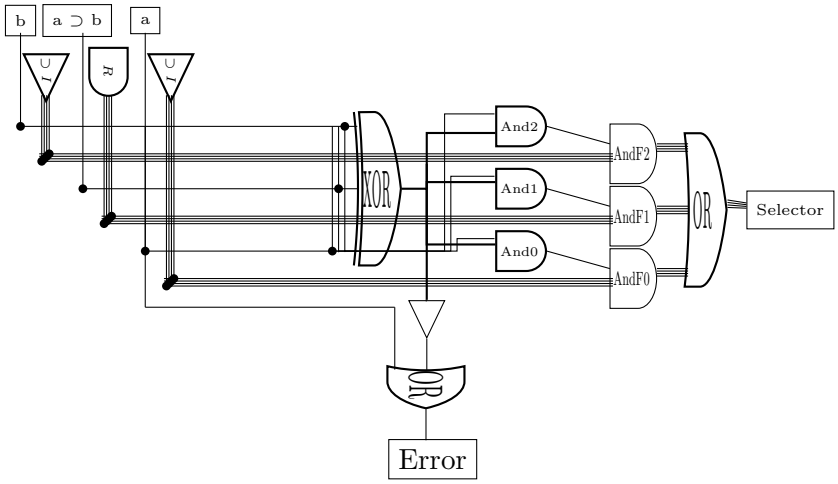


Figure 17: Circuit for the node $a \supset b$

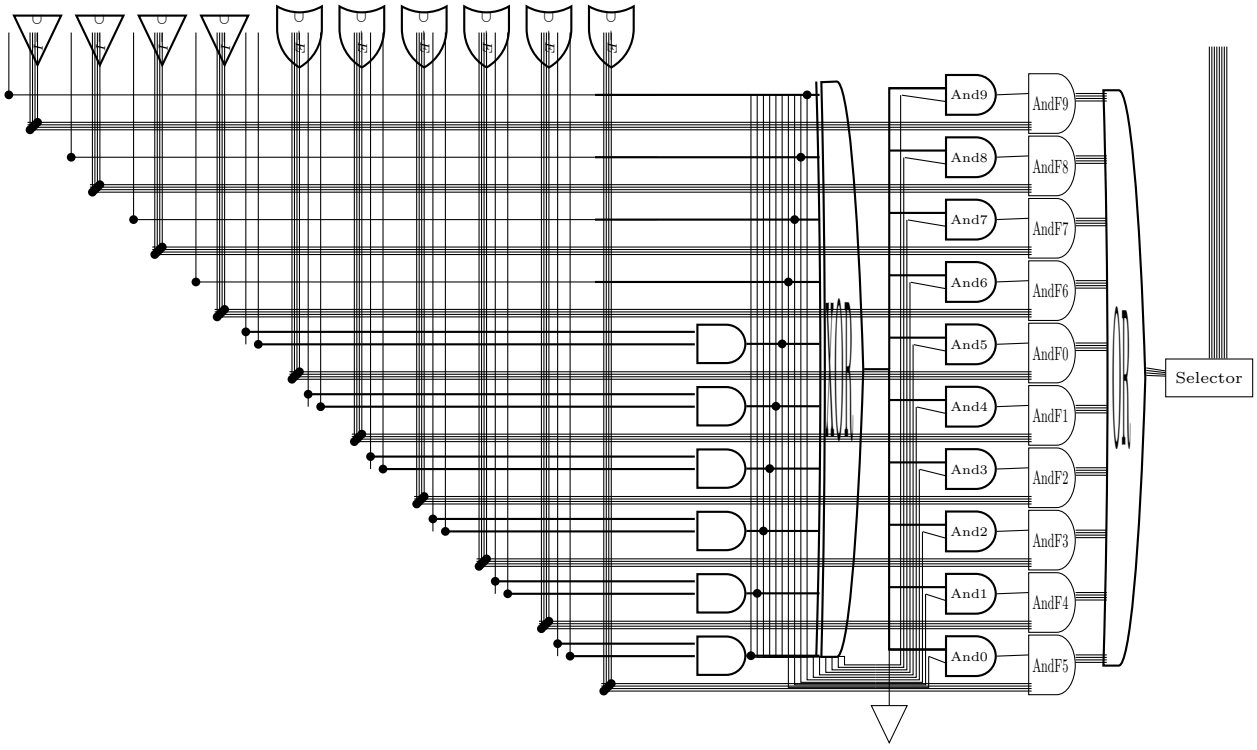


Figure 18: The Boolean circuit associated with the DLDS

The circuit in figure 19 represents the boolean function corresponding to the original dag-like proof, when each dag node is replaced by its associate node circuit.

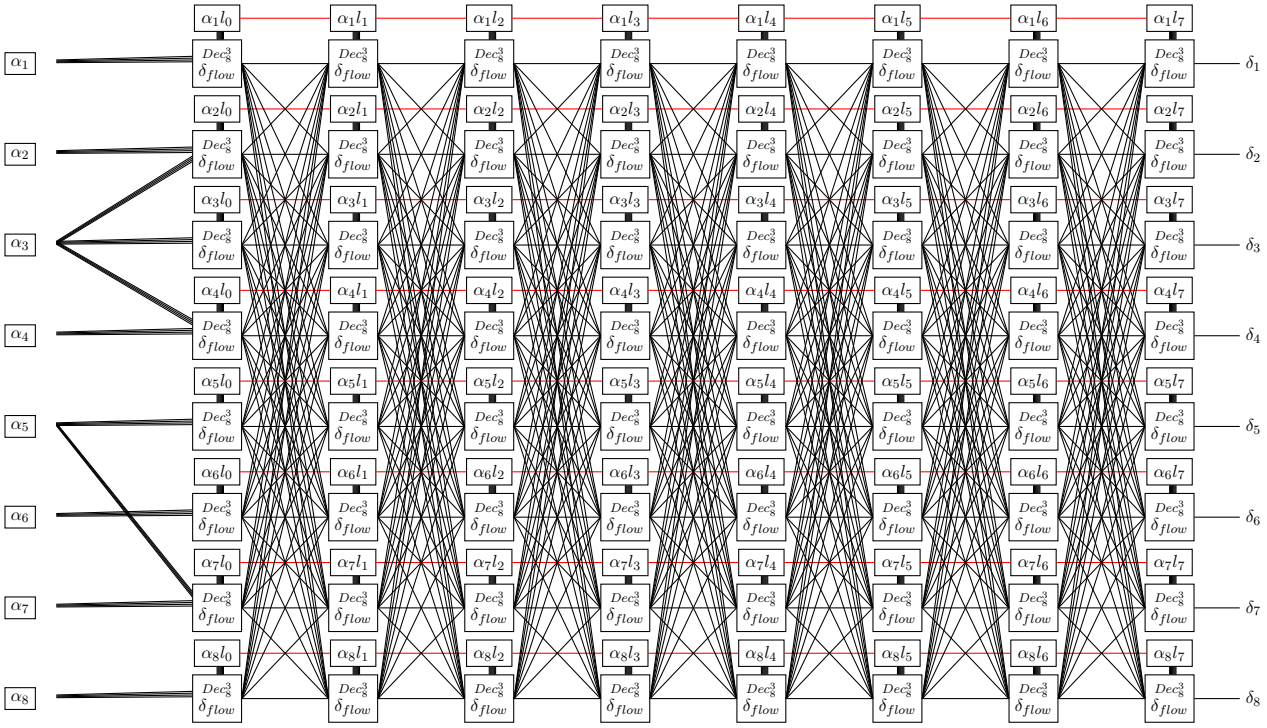


Figure 19: The circuit corresponding to the original dag-like proof

The circuit in the following, figure 20 makes $out_i = 1$, $i = 0, \ldots, 3$, whenever the value of i, in binary form, is $in_1 in_0$.

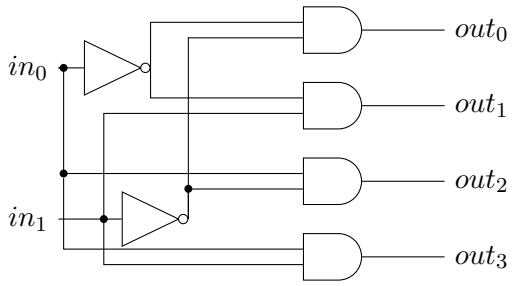


Figure 20: The two-bits selector

5 Formalization in Lean

We formalized the correctness of the Boolean circuit generated from a compressed DLDS using the Lean 4 theorem prover. The core theorem guarantees that if the DLDS encodes a valid minimal implicational proof, then the resulting circuit will return 1 on every input path. Our formalization is structured into two levels:

- **Local (node-level) correctness:** Each node in the grid correctly implements the logical behavior of its inference rules and outputs the correct dependency vector whenever exactly one rule is active.
- **Global (grid-level) correctness:** The composition of node computations along any valid proof path produces the correct final output, mirroring the structure and correctness of the DLDS proof.

5.1 Node Semantics

Each node is associated with a finite set of candidate inference rules, each of which may be either active or inactive. A node is modeled as a structure containing a pairwise-distinct list of such rules. The function that computes the node's output first checks which rules are active, ensures exactly one is active, and applies its `combine` function:

```
def node_logic {n : Nat} (rules : List (Rule n))
               (inputs : List (List.Vector Bool n)) : List.Vector Bool n :=
  let acts  := extract_activations rules
  let xor   := multiple_xor acts
  let masks := and_bool_list xor acts
  let outs  := apply_activations rules masks inputs
  list_or outs
```

The main node-correctness theorem packages this up for any `CircuitNode`:

```
theorem node_correct {n} (c : CircuitNode n)
    (inputs : List (List.Vector Bool n))
    (h_one : exactlyOneActive c.rules) :
   r  c.rules, c.run inputs = r.combine inputs
```

5.2 Grid Semantics

We arrange all nodes in a square $n \times n$ grid. Each node represents a formula at a particular level of the DLDS. The evaluation along a path is recursively defined: at each step, a node is selected and receives the current state as input, producing the next state. The evaluation is implemented as:

```
def evalGrid {n L : }
  (initial : List.Vector Bool n)
  (grid    : Grid n (CircuitNode n))
  (p       : GraphPath n L)
: List.Vector Bool n :=
  p.idxs.foldl (fun st sel => evalStepOne grid st sel) initial
```

The central grid-correctness theorem is:

> If every node in the grid has exactly one active rule, then for any valid path p of length L, there exists a sequence of intermediate states vs of length $L + 1$, where:
>
> - $vs[0]$ is the initial input;
> - $vs[L]$ is the output of `evalGrid initial grid p`;
> - for each i, $vs[i + 1]$ is obtained by applying the unique rule at the node selected by $p.idxs[i]$ to $vs[i]$.

This is formalized as:

```
theorem grid_correct {n L : }
  (grid : Grid n (CircuitNode n))
  (h_act :  i hi, exactlyOneActive (grid.nodes.nthLe i hi).rules)
  (initial : List.Vector Bool n)
  (p : GraphPath n L) :
   (vs : List (List.Vector Bool n)) (vs_len : vs.length = L + 1),
    vs.nthLe 0 _ = initial
    vs.nthLe L _ = evalGrid initial grid p
     (i : Fin L),
      let inputs := vs.nthLe i.val _
      let sel := p.idxs.nthLe i.val _
      vs.nthLe (i.val + 1) _ =
        (grid.nodes.nthLe sel.val _).run [inputs]
```

This modular theorem guarantees that the Lean formalization faithfully simulates the DLDS proof, step by step. The full development is available at: `https://github.com/lorenzosaraiva/DLDSBooleanCircuit`.

6 Analysis of the Deutsch-Jozsa Algorithm for Almost Constant Functions

In this section, we consider the usual quantum circuit to compute the Deutsch-Jozsa algorithm, as shown in figure 21. The Deutsch-Jozsa algorithm determines whether a Boolean function $f : \{0, 1\}^n \to \{0, 1\}$ is constant

(the same output for all inputs) or balanced (equal number of 0 and 1 s), see [NC00] for more details. Using Hadamard gates, an oracle, and interference, the quantum circuit solves this problem in one query, achieving an exponential speedup over classical methods. The previous section explored its application to a circuit representing a DAG proof for a non-Hamiltonian graph. However, we have to ensure that if f is a constant function, then the probability of the Deustch-Jozsa to provide a constant 1, i.e., *true* result is 100% and that is also a high probability that it recognizes a non-constant function f whenever f is not constant. In the sequel, we briefly show that Deutsch-Jozsa quantum circuit, depicted in figure 21, exhibits the above-mentioned behaviour on an arbitrary n-ary boolean function f, where $U_f(|x\rangle|y\rangle) = |x\rangle|y \oplus f(x)\rangle$, for any $x \in \{0,1\}^n$, and $y \in \{0,1\}$

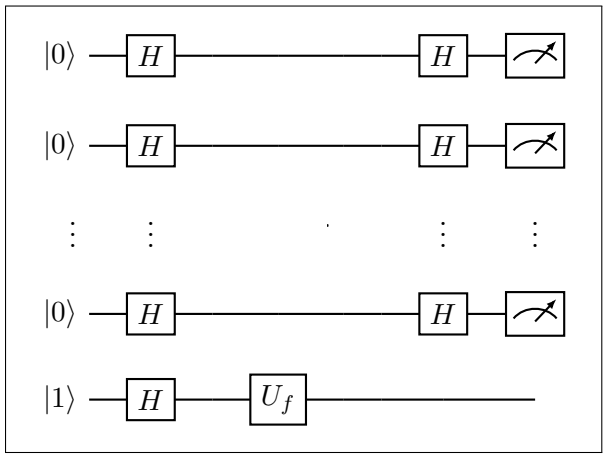


Figure 21: Quantum circuit for the Deutsch-Jozsa algorithm.

6.1 Deutsch-Jozsa probabilities for constant and non-constant functions

Taking a function $f : \{0,1\}^n \to \{0,1\}$, we have the following definition and consideration:

- The function is *almost constant*, if and only if, $f(x) = 0$ for all x, except for a single point x_0, where $f(x_0) = 1$.
- The initial state of the quantum system is:

$$|\psi_0\rangle = |0\rangle^{\otimes n} \otimes |1\rangle$$

After applying Hadamard gates to all input qubits and the auxiliary qubit, the state becomes:

$$|\psi_1\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

6.2 Application of the Function f

The operator U_f acts on the state by applying a phase shift based on $f(x)$:

$$U_f : |x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$$

Equivalently, the phase $(-1)^{f(x)}$ is applied to the basis state $|x\rangle$:

$$|\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{f(x)} |x\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

For an *almost constant* function, we have:

- For $x \neq x_0$, $f(x) = 0$, so the phase is $(-1)^0 = 1$.
- For $x = x_0$, $f(x) = 1$, so the phase is $(-1)^1 = -1$.

The state becomes:

$$|\psi_2\rangle = \frac{1}{\sqrt{2^n}} \left(\sum_{x \neq x_0} |x\rangle + (-1)|x_0\rangle \right) \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

6.3 Application of Hadamard Gates

After applying Hadamard gates to all input qubits, the state becomes:

$$|\psi_3\rangle = \frac{1}{\sqrt{2^n}} \sum_{z \in \{0,1\}^n} \left(\frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{x \cdot z + f(x)} \right) |z\rangle$$

The term of interest is the amplitude associated with $|z = 0^n\rangle$ (the state indicating that the function is *constant*):

$$A_{z=0^n} = \frac{1}{2^n} \sum_{x \in \{0,1\}^n} (-1)^{f(x)}$$

For an *almost constant* function:

- $(-1)^{f(x)} = 1$ for $x \neq x_0$,
- $(-1)^{f(x_0)} = -1$.

Thus:

$$A_{z=0^n} = \frac{1}{2^n}\left((2^n - 1) \cdot 1 + (-1) \cdot 1\right) = \frac{1}{2^n}\left(2^n - 2\right)$$

The probability of measuring $|z = 0^n\rangle$ is:

$$P_{z=0^n} = |A_{z=0^n}|^2 = \left(\frac{2^n - 2}{2^n}\right)^2$$

6.4 Probability of Error

The probability of the algorithm concluding that the function is constant, given that it is *almost constant*, is $P_{z=0^n}$:

$$P_{\text{constant}} = \left(\frac{2^n - 2}{2^n}\right)^2$$

For large n, this probability approximates:

$$P_{\text{constant}} \approx 1 - \frac{4}{4^n}$$

Thus, the probability of error (concluding that the function is constant when it is not) is:

$$P_{\text{error}} = 1 - P_{\text{constant}} = \frac{4}{4^n}$$

6.5 Adapting to the circuit

In our work, the proof is encoded by a series of paths, where each path represents the decision steps from each of the wires going from the leaf nodes to the end nodes of the circuit generated to represent the DLDS. In a classical setting, each possible path combination must be checked separately. In a quantum setting, on the other hand, we can leverage the advantage provided by the Deutsch-Jozsa algorithm to improve on this. We know that any Boolean circuit can be represented by a function $f : \{0,1\}^n \to \{0,1\}$. By representing our circuit as such, we can discover if it is balanced or constant in one query. Since the case of a valid proof will output all zeroes for every possible input, we can verify the validity of the DLDS in this way. In our encoding, we apply the Hadamard gates on all the bits referring to the paths. By doing this, we can superposition all possible paths in the DLDS.

References

[dMBF25] Robinson Callou de Moura Brasil Filho. Horizontal-compression: Lean-assisted proofs about the hc algorithm. `https://github.com/RCMBF/Horizontal-Compression/tree/main`, 2025. Last accessed on 2025-01-02.

[GH19] L. Gordeev and E. H. Haeusler. Proof compression and np versus pspace. *Studia Logica*, 107(1):55–83, 2019.

[GH20] L. Gordeev and E. H. Haeusler. Proof compression and np versus pspace ii. *Bulletin of the Section of Logic*, 49(3):213–230, 2020.

[GH22a] L. Gordeev and E. H. Haeusler. On proof theory in computational complexity: overview, 2022.

[GH22b] Lew Gordeev and Edward Hermann Haeusler. Proof Compression and NP Versus PSPACE II: Addendum. *Bulletin of the Section of Logic*, 51(2):197–205, 2022.

[Hae14] E. H. Haeusler. Propositional logics complexity and the subformula property. In *Proceedings of the Tenth International Workshop on Developments in Computational Models DCM*, 2014.

[Hae22] Edward Hermann Haeusler. Exponentially huge natural deduction proofs are redundant: Preliminary results on $\mathrm{m}_{\supset}$. *FLAP*, 9(1):287–326, 2022.

[Hud93] J. Hudelmaier. An $o\,(n \log n)$-space decision procedure for intuitionistic propositional logic. *Journal of Logic and Computation*, 3:1–13, 1993.

[NC00] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[Sta79] R. Statman. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science*, 9:67–72, 1979.