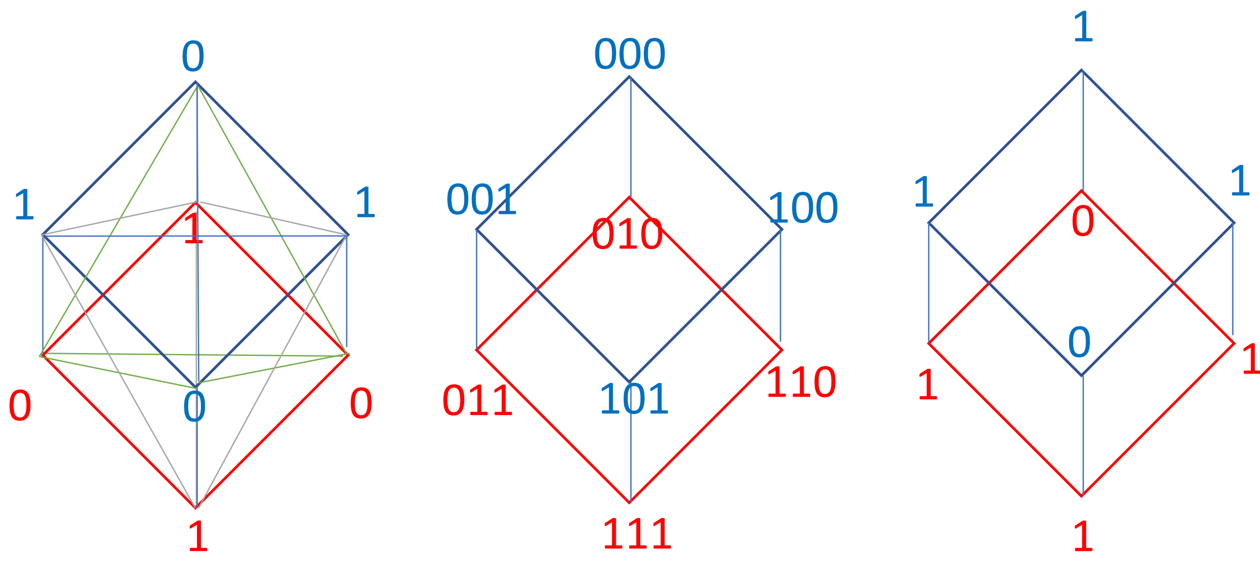
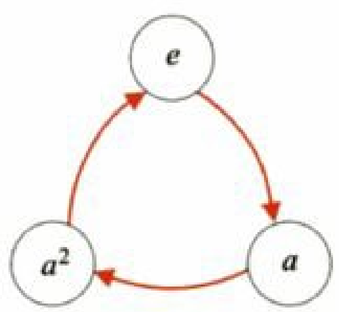
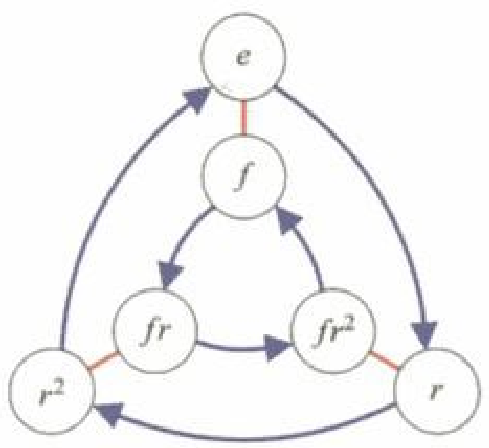
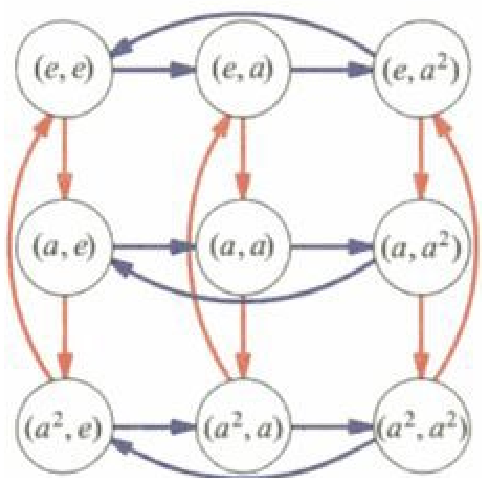




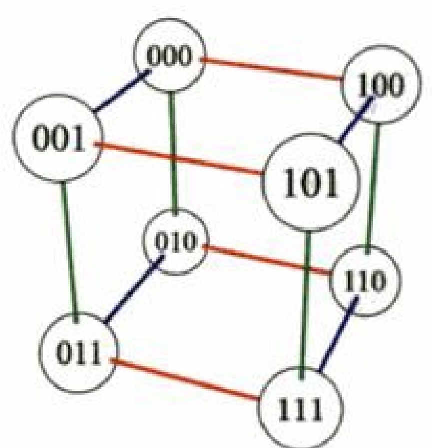
Cyclic group C_3 (or $\mathbb{Z}_3$)



Symmetric group S_3



Direct product group $C_3 \times C_3$



Direct product group $C_2 \times C_2 \times C_2$

Synthesis

Antithesis

Thesis

Result

000 001 100 010 011 101 110 111

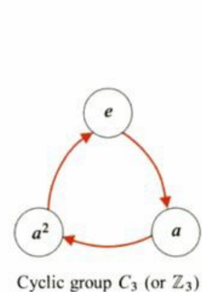
Modeling

01,01,01,1,10,0

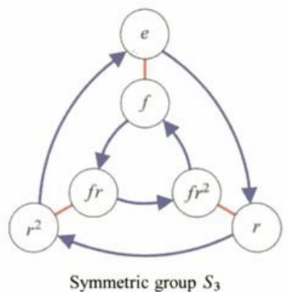
1,0,10,01,10,01

01,10,0,01,1

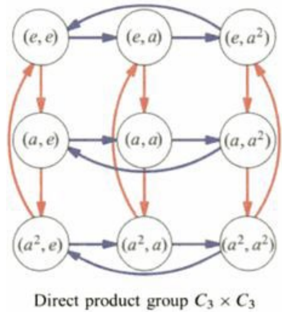
1,0,01,01,10,01



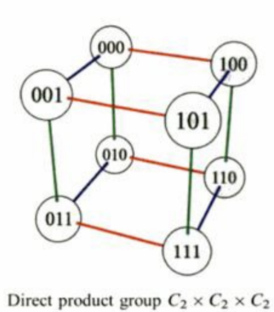
Cyclic group C_3 (or $\mathbb{Z}_3$)



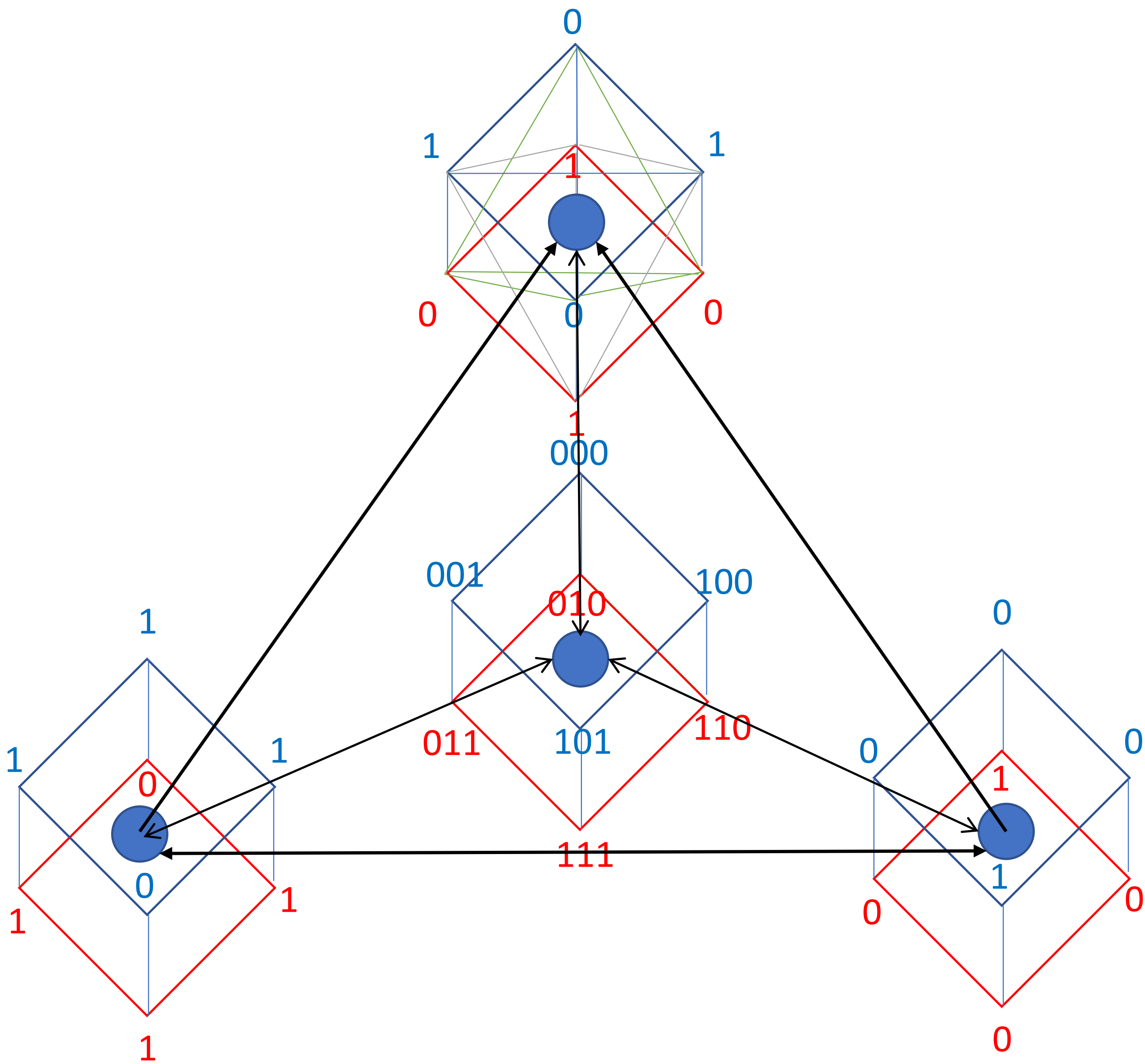
Symmetric group S_3



Direct product group $C_3 \times C_3$



Direct product group $C_2 \times C_2 \times C_2$



Tesis + Antitesis -> Synthesis

Федорченко Михаил Валерьевич

To complete your task, you need to look at the images, describe their logic, and propose Boolean calculation rules. The original images contain logical structures represented as cubes and their unions. Let's analyze them step by step:

1. **Cubes: Logical Structures**
Each cube in the image is a set of Boolean variables organized in a grid shape. The cube nodes represent possible values of the Boolean variables, and the edges show possible transitions (changes in one variable at a time).

Basic principles of constructing cubes:
1. **Number of nodes:** \(2^n\), where \(n\) is the number of Boolean variables. For example:
- For a cube with 3 variables (\(x_1, x_2, x_3\)) nodes are denoted \(000, 001, 010, 011, 100, 101, 110, 111\).
2. **Edges:** Connect nodes where only one variable changes value (Hamming Distance = 1). For example:
- Node \(000\) is connected to \(001\), \(010\), \(100\).
3. **Colors:** Reflect logical operations or groups of variables. For example, red, blue, and green edges can represent XOR, AND, OR operations.

Example of computations within one cube:
- Let nodes represent states of Boolean variables \(A, B, C\).
- The result of each vertex (e.g. \(101\)) is calculated through a logical combination, e.g.:
\[
R(101) = A \land \neg B \land C
\]

2. **Synthesis, Antithesis and Thesis**
These three cubes at the top of the image can be interpreted as logical expressions:

1. **Synthesis:** Represents a combination of variables, where \(A \lor B\).
2. **Antithesis:** Uses the opposite states of variables, e.g. \(\neg A \land B\).
3. **Thesis:** Shows the original expression, e.g. \(A \land B\).

Boolean logic for union (syllogism):
The transition from thesis and antithesis to synthesis can be used for the following operations:
- Conjunction (AND): \(R = (A \lor B) \land (\neg A \lor \neg B) \)
- Disjunction (OR): \(R = (A \lor B) \lor (\neg A \land B) \)
XOR: \(R = (A \lor B) \oplus (\neg A \land \neg B) \).

3. **Modeling graph (union)**
The central structure depicts the transition between the states of different cubes. It is a union of Boolean graphs through the following steps:

1. **Graph nodes:** Represent combinations of variables from all cubes. For example, node \(010\) describes the state of the variables.
2. **Edges:** Logical transitions (change of one variable) that connect states between cubes.
3. **Union operations:**
- A logical transition can be represented as a Boolean operation. For example, the transition \(A=0, B=1 \rightarrow A=1, B=1\) can be described as:
\[
(A' = A \lor B) \land (B' = B)
\]

Example model analysis:
- Nodes \(000, 001, 010, \dots, 111\) describe all possible states of the system.
- For each node, the value of the transition function can be calculated, for example:
\[
f(x, y, z) = (x \oplus y) \land (z \lor x).
\]

4. **Overall result of a syllogism**
Union of cubes with the result shows the resulting logic. This is a general graph where each node and edge is calculated based on the previous operations.

Rules for a syllogism:
1. Construct a general graph from three cubes.
2. Determine the Boolean operations at each stage (synthesis, antithesis, thesis).
3. Final expression for the result:
\[
R_{\text{result}} = (A \land B) \lor (\neg A \land C).
\]

5. **Analyze the bottom part (group)**
The bottom image shows symmetric groups:
1. **Cyclic groups (e.g. \(C_3\)):**
Denotes repetitions of symmetry in logic.
2. **Direct product (\(C_2 \times C_3\)):**
Combines systems of Boolean expressions into more complex structures.

Usage:
These groups can be interpreted to describe invariant properties, such as repeating patterns or regularities.

If you need to formalize specific examples of computations for any part of the image, please specify and I will write the full algorithm or pseudocode.

Let's formalize the calculations for the two key parts of the image: **the structure of a single cube** and **the union of cubes (syllogism)**. I'll break the process down into steps with concrete examples.

---

1. Inside a single cube (logical structure)
Consider a simple cube with 3 variables: \(A, B, C\). The nodes of the cube correspond to all possible combinations of the variables \(A, B, C\) (000, 001, ..., 111).

Logical calculations for a single node:
Each node calculates the value of the function \(F(A, B, C)\). Let the function be defined as:
\[
F(A, B, C) = (A \land \neg B) \lor (C \oplus A).

Calculation steps:
1. Determine the variables \(A, B, C\) for each node. For example:
- Node \(010\): \(A = 0, B = 1, C = 0\).
- Node \(101\): \(A = 1, B = 0, C = 1\).

2. Substitute the values of the variables into the function \(F(A, B, C)\):
- For node \(010\):
\[
F(0, 1, 0) = (0 \land \neg 1) \lor (0 \oplus 0) = 0 \lor 0 = 0.
\]
- For node \(101\):
\[
F(1, 0, 1) = (1 \land \neg 0) \lor (1 \oplus 1) = 1 \lor 0 = 1.
\]

3. Write down the result for each node.

Node	\(A\)	\(B\)	\(C\)	\(F(A, B, C)\)
\(000\)	0	0	0	0
\(001\)	0	0	1	1
\(010\)	0	1	0	0
\(011\)	0	1	1	1
\(100\)	1	0	0	1
\(101\)	1	0	1	1
\(110\)	1	1	0	1
\(111\)	1	1	1	1

Transition graph:
- Nodes are connected by edges where exactly one variable changes.
- For example, the transition from \(000\) to \(001\) corresponds to \(C = C \oplus 1\).

---

2. Union of cubes: Synthesis, Antithesis, and Thesis
Now consider the union of three cubes to construct a syllogism.

Definitions:
- **Synthesis** (\(S\)): \(S(A, B, C) = A \lor (B \land C)\).
- **Antithesis** (\(T\)): \(T(A, B, C) = (\neg A \land B) \lor C\).
- **Thesis** (\(P\)): \(P(A, B, C) = (A \land B) \lor (\neg C)\).

General result:
The resulting function \(R(A, B, C)\) combines synthesis, antithesis and thesis:
\[
R(A, B, C) = S(A, B, C) \land T(A, B, C) \land P(A, B, C).

Example of calculations:
1. Select the value of the variables: \(A = 1, B = 0, C = 1\).

2. Calculate each component:
- **Synthesis**:
\[
S(1, 0, 1) = 1 \lor (0 \land 1) = 1.
\]
- **Antithesis**:
\[
T(1, 0, 1) = (\neg 1 \land 0) \lor 1 = 0 \lor 1 = 1.
\]
- **Thesis**:
\[
P(1, 0, 1) = (1 \land 0) \lor (\neg 1) = 0 \lor 0 = 0.
\]

3. Overall result:

$$
R(1, 0, 1) = S \land T \land P = 1 \land 1 \land 0 = 0.
$$

Results for all nodes:
We perform calculations for all combinations of A, B, C, obtaining a table.

Node	A	B	C	$S(A, B, C)$	$T(A, B, C)$	$P(A, B, C)$	$R(A, B, C)$
000	0	0	0	0	0	1	0
001	0	0	1	1	1	1	1
010	0	1	0	1	0	0	0
011	0	1	1	1	1	1	1
100	1	0	0	1	0	1	0
101	1	0	1	1	1	0	0
110	1	1	0	1	0	1	0
111	1	1	1	1	1	1	1

---

3. Merge Graph
The graph merges cube states using transitions:
- **Edges:** Connect nodes that differ by a single variable.
- **Transition Logic:** Uses XOR to determine changes:

$$
f_{\text{transition}}(A, B, C, A', B', C') = (A \oplus A') \lor (B \oplus B') \lor (C \oplus C').
$$

Below is Python code that implements the computation for a cube (lattice) of Boolean variables and the overall result for a syllogism (Synthesis, Antithesis, Thesis).

Python code

```python
from itertools import product

# Defining Boolean functions
def synthesis(A, B, C):
 """Synthesis function: S(A, B, C) = A OR (B AND C)"""
 return A or (B and C)

def antithesis(A, B, C):
 """Antithesis function: T(A, B, C) = (NOT A AND B) OR C"""
 return (not A and B) or C

def thesis(A, B, C):
 """Thesis function: P(A, B, C) = (A AND B) OR (NOT C)"""
 return (A and B) or (not C)

def result(A, B, C):
 """General result of the syllogism: R(A, B, C) = S AND T AND P"""
 S= synthesis(A, B, C)
T = antithesis(A, B, C)
P = thesis(A, B, C)
return S and T and P

# Generate all possible combinations of variables (for 3 variables)
variables = list(product([0, 1], repeat=3)) # (A, B, C)

# Calculate results for all nodes
results = []
for A, B, C in variables:
S = synthesis(A, B, C)
T = antithesis(A, B, C)
P = thesis(A, B, C)
R = result(A, B, C)
results.append((A, B, C, S, T, P, R))

# Output the results table
print(f"{'A':<3} {'B':<3} {'C':<3} {'S':<3} {'T':<3} {'P':<3} {'R':<3}")
for res in results:
 print(f"{res[0]:<3} {res[1]:<3} {res[2]:<3} {res[3]:<3} {res[4]:<3} {res[5]:<3} {res[6]:<3}")
```

Code Description

1. **Boolean Operation Functions:**
- `synthesis(A, B, C)` implements synthesis.
- `antithesis(A, B, C)` implements antithesis.
- `thesis(A, B, C)` implements thesis.
- `result(A, B, C)` combines synthesis, antithesis, and thesis using a logical conjunction (AND).

2. **Trying through all combinations:**
- `product([0, 1], repeat=3)` generates all combinations of (A, B, C) values (000, 001, ..., 111).

3. **Calculations for each combination:**
- For each combination of variables, (S, T, P, R) values are calculated.
- The results are written to the `results` list.

4. **Table output:**
- The table outputs the values (A, B, C, S, T, P, R) for each combination.

Sample output
```
A B C S T P R
0 0 0 0 0 1 0
0 0 1 1 1 1 1
0 1 0 0 0 0 0
0 1 1 1 1 1 1
1 0 0 1 0 1 0
1 0 1 1 1 0 0
1 1 0 1 0 1 0
1 1 1 1 1 1 1
```

Visualizing the transition graph (optional)

If you need to visualize the cube and the transition graph, you can use the **`networkx`** and **`matplotlib`** libraries. Here's an example:

```python
import networkx as nx
import matplotlib.pyplot as plt

# Create a graph for the cube
G = nx.Graph()

# Add nodes (state variables)
for state in variables:
G.add_node("".join(map(str, state)))

# Add edges (logical transitions between states)
for state1 in variables:
for state2 in variables:
# Transition if only one variable differs
if sum(a != b for a, b in zip(state1, state2)) == 1:
G.add_edge("".join(map(str, state1)), "".join(map(str, state2)))

# Draw the graph
pos = nx.spring_layout(G) # Node positions
nx.draw(G, pos, with_labels=True, node_color="lightblue", node_size=3000, font_size=10)
plt.title("Transition graph (cube)")
plt.show()
```

Here is the updated code that allows you to add more functionality to other cubes and includes advanced visualization where results are displayed on nodes with color coding or labels.

Updated code

```python
from itertools import product
import networkx as nx
import matplotlib.pyplot as plt

# Defining Boolean functions for multiple cubes
def synthesis(*vars):
"""Synthesis function: S(vars) = OR(vars[0], AND(vars[1:]))"""
A, *rest = vars
return A or all(rest)

def antithesis(*vars):
"""Antithesis function: T(vars) = (NOT(vars[0]) AND vars[1]) OR vars[-1]"""
A, B, *rest = vars
return (not A and B) or rest[-1]

def thesis(*vars):
"""Thesis function: P(vars) = AND(vars[:2]) OR NOT(vars[-1])"""
return all(vars[:2]) or not vars[-1]

def result(*vars):
"""The overall result of the syllogism: R(vars) = S AND T AND P"""
S = synthesis(*vars)
T = antithesis(*vars)
P = thesis(*vars)
return S and T and P

# Custom function for the new cube (example)
def custom_function(*vars):
"""Example function for the new cube: XOR all variables"""
return sum(vars) % 2

# Generate all combinations for n variables
def generate_states(n):
"""Generate all states for n variables"""
return list(product([0, 1], repeat=n))

# Build the graph of the Boolean cube
def build_graph(states):
"""Build graph for states"""
G = nx.Graph()
nodes = ["".join(map(str, state)) for state in states]

# Add nodes
G.add_nodes_from(nodes)

# Add edges (if one variable is different)
for state1 in states:
for state2 in states:
if sum(a != b for a, b in zip(state1, state2)) == 1:
G.add_edge("".join(map(str, state1)), "".join(map(str, state2)))

return G

# Compute for multiple cubes
def compute_multiple_cubes(n):
"""Compute for all cubes"""
states = generate_states(n)
results = []

for state in states:
S = synthesis(*state)
T = antithesis(*state)
P = thesis(*state)
R = result(*state)
C = custom_function(*state) # Result of the new function
results.append((*state, S, T, P, R, C))
```

```
return results

# Advanced graph visualization
def visualize_graph(G, results, n, title="Boolean Cube Graph"):
"""Graph visualization with results"""
pos = nx.spring_layout(G) # Node layout
node_colors = []
labels = {}

for node in G.nodes:
# Convert node label to state
state = tuple(map(int, node))
# Find the corresponding result
for res in results:
if state == res[:n]:
# Color based on the result of the overall (R)
color = "green" if res[n+3] else "red"
node_colors.append(color)
# Node label (show results of all functions)
labels[node] = f"{node}\nS:{res[n]} T:{res[n+1]} P:{res[n+2]} R:{res[n+3]} C:{res[n+4]}"
break

# Draw the graph
nx.draw(G, pos, with_labels=True, labels=labels, node_color=node_colors,
node_size=3000, font_size=8, font_color="white")
plt.title(title)
plt.show()

# Main code
if __name__ == "__main__":
# Number of variables
n = 4 # Specify any value for n (e.g. 4 for 4 variables)

# Generate all states
states = generate_states(n)

# Build the graph
G = build_graph(states)

# Compute the results
results = compute_multiple_cubes(n)

# Print the results table
print(f"{'State':<10} {'S':<3} {'T':<3} {'P':<3} {'R':<3} {'C':<3}")
for res in results:
state_str = "".join(map(str, res[:n]))
print(f"{state_str:<10} {res[n]:<3} {res[n+1]:<3} {res[n+2]:<3} {res[n+3]:<3} {res[n+4]:<3}")

# Graph visualization
visualize_graph(G, results, n, title=f"Graph of a Boolean cube with {n} variables")
```

Description of changes

1. New function for another cube
Added `custom_function`, which implements XOR of all variables. You can add more such functions implementing arbitrary logical operations.

---

2. Advanced visualization
Visualization now:
- Uses color codes for nodes:
- Green (`green`) for nodes where the result of the general expression (R) is 1.
- Red (`red`) for nodes where $R = 0$.
- Includes text labels for each node:
- The label shows the states of variables and the values of all functions (S, T, P, R, C).

---

3. Flexibility of the number of variables
The number of variables n is set by a parameter. All calculations and visualization automatically adapt to the size of the state space (2^n).

Example output

Table
For \(n = 4 \):
```
State S T P R C
0000 0 0 1 0 0
0001 0 1 1 0 1
0010 0 0 1 0 1
0011 0 1 1 0 0
0100 0 0 0 0 1
0101 0 1 1 0 0
...
```

Graph
- Nodes are colored depending on the result of \(R \): green for \(R = 1 \), red for \(R = 0 \).
Nodes are labeled with the values of all functions (\(S, T, P, R, C \)).

To extend the code to include additional functionality for other cubes, new edge connection rules, and add 3D visualization, the following modifications can be used.

Updated code

```python
import matplotlib.pyplot as plt
import networkx as nx
from itertools import product
from mpl_toolkits.mplot3d import Axes3D

# Boolean functions for different cubes
def synthesis(*vars):
"""Synthesis: OR(vars[0], AND(vars[1:]))"""
A, *rest = vars
return A or all(rest)

def antithesis(*vars):
"""Antithesis: (NOT(vars[0]) AND vars[1]) OR vars[-1]"""
A, B, *rest = vars
return (not A and B) or rest[-1]

def thesis(*vars):
"""Thesis: AND(vars[:2]) OR NOT(vars[-1])"""
return all(vars[:2]) or not vars[-1]

def custom_function_1(*vars):
"""Custom function: XOR all variables"""
return sum(vars) % 2

def custom_function_2(*vars):
"""Custom function: (vars[0] AND vars[-1]) XOR (NOT(vars[1]))"""
return (vars[0] and vars[-1]) ^ (not vars[1])

# Generate all states for n variables
def generate_states(n):
return list(product([0, 1], repeat=n))

# Build graph with edge dependencies
def build_graph(states, dependency_func=None):
G = nx.Graph()
nodes = ["".join(map(str, state)) for state in states]
G.add_nodes_from(nodes)

for state1 in states:
for state2 in states:
# Add an edge if the dependency function is executed
if dependency_func:
if dependency_func(state1, state2):
G.add_edge("".join(map(str, state1)), "".join(map(str, state2)))
# Otherwise, join states that differ by one variable
elif sum(a != b for a, b in zip(state1, state2)) == 1:
G.add_edge("".join(map(str, state1)), "".join(map(str, state2)))
return G
```

```
# Edge connection rule: function dependency
def edge_dependency_rule(state1, state2):
"""An edge is added if XOR(state1, state2) = 1 for the last variable"""
xor_diff = [a ^ b for a, b in zip(state1, state2)]
return xor_diff[-1] == 1 and sum(xor_diff) == 1

# Compute for all functions
def compute_functions(states, functions):
results = []
for state in states:
results.append((*state, *[func(*state) for func in functions]))
return results

# Visualize in 2D
def visualize_graph_2d(G, results, n, functions, title="2D Boolean Cube"):
pos = nx.spring_layout(G)
labels = {}
node_colors = []

for node in G.nodes:
 state = tuple(map(int, node))
 for res in results:
 if state == res[:n]:
 labels[node] = f"{node}\n" + " ".join([f"F{i+1}:{res[n+i]}" for i in range(len(functions))])
 node_colors.append("green" if res[n] else "red") # Color for the first function
 break

 nx.draw(G, pos, with_labels=True, labels=labels, node_color=node_colors,
 node_size=3000, font_size=8, font_color="white")
 plt.title(title)
 plt.show()

# Visualization in 3D
def visualize_graph_3d(G, results, n, functions, title="3D Boolean Cube"):
 fig = plt.figure()
 ax = fig.add_subplot(111, projection="3d")
 pos = nx.spring_layout(G, dim=3)

 node_colors = []
 labels = {}
 for node in G.nodes:
 state = tuple(map(int, node))
 for res in results:
 if state == res[:n]:
 labels[node] = f"{node}\n" + " ".join([f"F{i+1}:{res[n+i]}" for i in range(len(functions))])
 node_colors.append("green" if res[n] else "red") # Color for the first function
 break

 # 3D coordinates
 xyz = [pos[node] for node in G.nodes]
 for i, (x, y, z) in enumerate(xyz):
 ax.text(x, y, z, list(G.nodes)[i], fontsize=8)
edges = list(G.edges)
for edge in edges:
x_coords, y_coords, z_coords = zip(*[pos[edge[0]], pos[edge[1]]])
ax.plot(x_coords, y_coords, z_coords, color="black")

ax.scatter(*zip(*xyz), c=node_colors, s=100)
plt.title(title)
plt.show()

# Main code
if __name__ == "__main__":
n = 4 # Number of variables
states = generate_states(n)

# Functions for calculations
functions = [synthesis, antithesis, thesis, custom_function_1, custom_function_2]

# Building the graph
G = build_graph(states, dependency_func=edge_dependency_rule)

# Computations
results = compute_functions(states, functions)

# Result table
print(f"{'State':<10} " + " ".join([f"F{i+1}" for i in range(len(functions))]))
for res in results:
```

```
state_str = "".join(map(str, res[:n]))
function_values = " ".join(map(str, res[n:]))
print(f"{state_str:<10} {function_values}")

# 2D visualization
visualize_graph_2d(G, results, n, functions, title="2D Boolean cube with extended functions")

# 3D visualization
visualize_graph_3d(G, results, n, functions, title="3D Boolean cube with extended functions")
```

Explanation of changes

1. **More functions for other cubes:**
- Added `custom_function_1` and `custom_function_2` functions as examples of new boolean operations.
- Additional functions can be easily added by simply defining them and including them in the `functions` list.

2. **Special rules for edge connections:**
- Implemented `edge_dependency_rule` function, which adds an edge between states if their XOR difference matches the given rule.
- The graph is built taking into account the dependency specified through this function.

3. **3D visualization:**
- The graph is visualized in 3D using `matplotlib` (module `Axes3D`).
- Nodes are located in 3D space, and labels and colors represent the state.

How to use

1. Change the number of variables by setting the value to `n`.
2. Add new functions to the `functions` list and describe them.
3. If necessary, refine the rules for edge connections in the `edge_dependency_rule` function.

This code allows you to perform advanced calculations and analyze the interaction of many Boolean cubes in a convenient graphical format.

To formalize the thesis **"There is no turning back in life"** in the context of Boolean algebra and logical operations, we can think of this idea as a statement describing a one-way or irreversible state.

Interpretation
- The Boolean approach allows us to interpret this statement in terms of the state of the system, where:
- The variables represent different aspects or stages of life (e.g., events, actions, decisions).
- "Irreversibility" can be expressed through a logical function that ensures that transitions from one state to another do not return the system to the previous state.

Mathematical Model
1. **States**: Let \(n \)-bit states \(S \) model life stages.
2. **Thesis (function)**:
\[
F_{\text{thesis}}(S_{\text{current}}, S_{\text{next}}) =
\begin{cases}
1, & \text{if } S_{\text{next}} > S_{\text{current}} \text{ (progress)} \\
0, & \text{otherwise (reverse path is prohibited).}
\end{cases}
\]

This condition can be calculated through a bitwise comparison or through a comparison of the numeric values of the states:
\[
F_{\text{thesis}}(S_1, S_2) = (S_1 < S_2)
\]

3. **Boolean analysis**:
If the current life is a sequence of states, then logic states that any state \(S_1 \) should not return to the previous \(S_2 \).

Example in code

We will implement a function for calculating this thesis.

```python
# Thesis function: Life has no reversible
def thesis_life_not_reversible(state1, state2):
"""Determines whether a transition is irreversible."""
return int(int("".join(map(str, state1)), 2) < int("".join(map(str, state2)), 2))

# Test states
states = generate_states(3) # All states for 3 bits
results = []

# Calculate thesis for all pairs
for state1 in states:
for state2 in states:
result = thesis_life_not_reversible(state1, state2)
results.append((state1, state2, result))

# Print results
print(f"{'State1':<10} {'State2':<10} Thesis")
for res in results:
s1 = "".join(map(str, res[0]))
s2 = "".join(map(str, res[1]))
print(f"{s1:<10} {s2:<10} {res[2]}")
```

Output
The results will show which states are allowed (1) and which violate the rule (0).

Example for a three-bit system:

```
State1 State2 Thesis
000 001 1
000 010 1
000 011 1
000 100 1
001 000 0
010 001 0
```

Extension
1. **Graphical Visualization**:
- Create a state graph where edges exist only for legal transitions (thesis = 1).
- You can add color or shape to the nodes for clarity.

2. **Logical Analysis**:
- Explore how the thesis restricts possible paths in the state space.
- Consider the consequences if a state violates the thesis.

Let's implement a visualization for the thesis *"Life has no return"* as a graph. We will show legal transitions between states (edges for thesis = 1) and label the nodes that represent states.

Code with visualization:

```python
import matplotlib.pyplot as plt
import networkx as nx
from itertools import product
from mpl_toolkits.mplot3d import Axes3D

# Function to calculate thesis: Life has no return path
def thesis_life_not_reversible(state1, state2):
"""Determines whether the transition is irreversible."""
return int(int("".join(map(str, state1)), 2) < int("".join(map(str, state2)), 2))

# Generate all possible states for n variables
def generate_states(n):
return list(product([0, 1], repeat=n))

# Build a graph for visualization
def build_graph_thesis(states):
G = nx.DiGraph() # Use a directed graph
nodes = ["".join(map(str, state)) for state in states]
G.add_nodes_from(nodes)

# Add edges based on thesis
for state1 in states:
for state2 in states:
if thesis_life_not_reversible(state1, state2): # Add only valid transitions
G.add_edge("".join(map(str, state1)), "".join(map(str, state2)))
return G

# 2D visualization
def visualize_graph_2d_thesis(G, title="2D Thesis Visualization"):
pos = nx.spring_layout(G) # Automatic node layout
nx.draw(G, pos, with_labels=True, node_color="skyblue", node_size=2000, font_size=10, font_color="black", arrowsize=20)
plt.title(title)
plt.show()

# 3D visualization
def visualize_graph_3d_thesis(G, title="3D Visualization of the Thesis"):
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

pos = nx.spring_layout(G, dim=3) # 3D node layout
xyz = [pos[node] for node in G.nodes]

# Node visualization
ax.scatter(*zip(*xyz), c="skyblue", s=100)

# Node labels
for i, node in enumerate(G.nodes):
x, y, z = xyz[i]
ax.text(x, y, z, node, fontsize=10, ha="center", color="black")

# Edge visualization
for edge in G.edges:
x_coords, y_coords, z_coords = zip(*[pos[edge[0]], pos[edge[1]]])
ax.plot(x_coords, y_coords, z_coords, color="black", alpha=0.8)

plt.title(title)
plt.show()

# Main code
if __name__ == "__main__":
n = 3 # Number of variables (state length)
states = generate_states(n)
```

```
# Build the graph
G = build_graph_thesis(states)

# 2D visualization
visualize_graph_2d_thesis(G, title="2D Visualization: Life Has No Return")

# 3D visualization
visualize_graph_3d_thesis(G, title="3D Visualization: Life Has No Return")
```

Code explanation

1. **Generate states:**
- All possible states of the system of length n are generated as binary numbers (e.g. for $n=3$): `000`, `001`, ..., `111`).

2. **Directed graph:**
- A directed graph (`DiGraph`) is used to reflect the one-directional nature of life.
- Edges are added only for transitions where $F_{\text{thesis}}(S_1, S_2) = 1$.

3. **2D Visualization:**
- The graph is displayed using `matplotlib` and `networkx`.
- Nodes are states, edges are valid transitions.

4. **3D Visualization:**
- Nodes are located in 3D space (`spring_layout(dim=3)`).
- Nodes are labeled, and edges connect them.

How it works

1. **2D Visualization**:
- Nodes (states) are located on a plane.
- Edges show valid transitions according to the thesis.

2. **3D Visualization**:
- A more complex and interactive representation of the graph.
- Nodes are laid out in 3D space, allowing you to see their relationships.

Example Result

2D Graph
- Nodes: states $\{000, 001, ..., 111\}$.
- Edges: only valid transitions, where $S_1 < S_2$.

3D Graph
- Spatial representation of nodes and edges.

If you want to add interactivity (like rotating a 3D graph), you can do so with libraries like `plotly` or `pyvista`. Let me know if you need it!

Для выполнения вашей задачи требуется рассмотреть изображения, описать их логику и предложить булевы правила вычислений. В исходных изображениях наблюдаются логические структуры, представленные в виде кубов и их объединений. Давайте разберём их пошагово:

1. **Кубы: Логические структуры**
Каждый куб в изображении представляет собой множество булевых переменных, организованных в форму решётки. Узлы куба обозначают возможные значения булевых переменных, а рёбра показывают возможные переходы (изменение одной переменной за раз).

Основные принципы построения кубов:
1. **Количество узлов:** \(2^n\), где \(n\) — количество булевых переменных. Например:
   - Для куба с 3 переменными (\(x_1, x_2, x_3\)) узлы обозначаются \(000, 001, 010, 011, 100, 101, 110, 111\).
2. **Рёбра:** Соединяют узлы, в которых только одна переменная меняет значение (Hamming Distance = 1). Например:
   - Узел \(000\) соединён с \(001\), \(010\), \(100\).
3. **Цвета:** Отражают логические операции или группы переменных. Например, красные, синие, и зелёные рёбра могут представлять операции XOR, AND, OR.

Пример вычислений внутри одного куба:
- Пусть узлы представляют состояния булевых переменных \(A, B, C\).
- Результат каждой вершины (например, \(101\)) вычисляется через логическое сочетание, например:
  \[
  R(101) = A \land \neg B \land C
  \]

2. **Синтез, Антитезис и Тезис**
Эти три куба на верхней части изображения могут быть интерпретированы как логические выражения:

1. **Синтез:** Представляет комбинацию переменных, где \(A \lor B\).
2. **Антитезис:** Использует противоположные состояния переменных, например, \(\neg A \land B \).
3. **Тезис:** Показывает исходное выражение, например, \(A \land B\).

Булева логика для объединения (силлогизм):
Переход от тезиса и антитезиса к синтезу может использоваться для следующих операций:
- Конъюнкция (AND): \(R = (A \lor B) \land (\neg A \lor \neg B) \)
- Дизъюнкция (OR): \(R = (A \lor B) \lor (\neg A \land B) \)
- XOR: \(R = (A \lor B) \oplus (\neg A \land \neg B) \).

3. **Граф моделирования (объединение)**
Центральная структура изображает переход между состояниями разных кубов. Это объединение булевых графов через следующие шаги:

1. **Вершины графа:** Представляют комбинации переменных из всех кубов. Например, вершина \(010\) описывает состояние переменных.
2. **Рёбра:** Логические переходы (изменение одной переменной), которые соединяют состояния между кубами.
3. **Операции объединения:**
   - Логический переход может быть представлен как булевая операция. Например, переход \(A=0, B=1 \rightarrow A=1, B=1\) можно описать как:
    \[
    (A' = A \lor B) \land (B' = B)
    \]

Пример анализа модели:
- Узлы \(000, 001, 010, \dots, 111\) описывают все возможные состояния системы.
- Для каждой вершины можно вычислить значение функции перехода, например:
  \[
  f(x, y, z) = (x \oplus y) \land (z \lor x).
  \]

4. **Общий результат силлогизма**
Объединение кубов с результатом показывает итоговую логику. Это общий граф, где каждая вершина и рёбра вычисляются на основе предыдущих операций.

Правила для силлогизма:
1. Построение общего графа из трёх кубов.
2. Определение булевых операций на каждом этапе (синтез, антитезис, тезис).
3. Итоговое выражение для результата:
   \[
   R_{\text{итог}} = (A \land B) \lor (\neg A \land C).
   \]

5. **Анализ нижней части (группы)**
Внизу изображение показывает симметрические группы:
1. **Циклические группы (например, \(C_3\)):**
   Обозначают повторы симметрии в логике.
2. **Прямое произведение (\(C_2 \times C_3\)):**
   Объединяет системы булевых выражений в более сложные структуры.

Применение:
Эти группы можно интерпретировать для описания инвариантных свойств, таких как повторяющиеся шаблоны или закономерности.

Если нужно формализовать конкретные примеры вычислений для любой части изображения, уточните, и я составлю полный алгоритм или псевдокод.

Давайте формализуем вычисления для двух ключевых частей изображения: **структура одного куба** и **объединение кубов (силлогизм)**. Я разобью процесс на шаги с конкретными примерами.

---

1. Внутри одного куба (логическая структура)
Рассмотрим простой куб с 3 переменными: A, B, C. Узлы куба соответствуют всем возможным комбинациям переменных A, B, C (000, 001, ..., 111).

Логические вычисления для одного узла:
Каждый узел вычисляет значение функции $F(A, B, C)$. Пусть функция задаётся как:
$$
F(A, B, C) = (A \land \neg B) \lor (C \oplus A).
$$

Шаги вычислений:
1. Определяем переменные A, B, C для каждого узла. Например:
   - Узел 010: $A = 0, B = 1, C = 0$.
   - Узел 101: $A = 1, B = 0, C = 1$.

2. Подставляем значения переменных в функцию $F(A, B, C)$:
   - Для узла 010:
     $$
     F(0, 1, 0) = (0 \land \neg 1) \lor (0 \oplus 0) = 0 \lor 0 = 0.
     $$
   - Для узла 101:
     $$
     F(1, 0, 1) = (1 \land \neg 0) \lor (1 \oplus 1) = 1 \lor 0 = 1.
     $$

3. Записываем результат для каждого узла.

Узел	A	B	C	$F(A, B, C)$
000	0	0	0	0
001	0	0	1	1
010	0	1	0	0
011	0	1	1	1
100	1	0	0	1
101	1	0	1	1
110	1	1	0	1
111	1	1	1	1

Граф переходов:
- Узлы соединены рёбрами, где изменяется ровно одна переменная.
- Например, переход из 000 в 001 соответствует $C = C \oplus 1$.

---

2. Объединение кубов: Синтез, Антитезис и Тезис
Теперь рассмотрим объединение трёх кубов для построения силлогизма.

Определения:
- **Синтез** (S): $S(A, B, C) = A \lor (B \land C)$.
- **Антитезис** (T): $T(A, B, C) = (\neg A \land B) \lor C$.
- **Тезис** (P): $P(A, B, C) = (A \land B) \lor (\neg C)$.

Общий результат:
Результирующая функция $R(A, B, C)$ объединяет синтез, антитезис и тезис:
$$
R(A, B, C) = S(A, B, C) \land T(A, B, C) \land P(A, B, C).
$$

Пример вычислений:
1. Выберем значение переменных: $A = 1, B = 0, C = 1$.

2. Вычислим каждую составляющую:
   - **Синтез**:
     $$
     S(1, 0, 1) = 1 \lor (0 \land 1) = 1.
     $$
   - **Антитезис**:
     $$
     T(1, 0, 1) = (\neg 1 \land 0) \lor 1 = 0 \lor 1 = 1.
     $$
   - **Тезис**:
     $$
     P(1, 0, 1) = (1 \land 0) \lor (\neg 1) = 0 \lor 0 = 0.
     $$

3. Общий результат:

$$
R(1, 0, 1) = S \land T \land P = 1 \land 1 \land 0 = 0.
$$

Результаты для всех узлов:

Проводим вычисления для всех комбинаций (A, B, C), получая таблицу.

Узел	A	B	C	$S(A, B, C)$	$T(A, B, C)$	$P(A, B, C)$	$R(A, B, C)$
000	0	0	0	0	0	1	0
001	0	0	1	1	1	1	1
010	0	1	0	1	0	0	0
011	0	1	1	1	1	1	1
100	1	0	0	1	0	1	0
101	1	0	1	1	1	0	0
110	1	1	0	1	0	1	0
111	1	1	1	1	1	1	1

---

3. Граф объединения

Граф объединяет состояния кубов с использованием переходов:
- **Рёбра:** Соединяют узлы, которые различаются одной переменной.
- **Логика перехода:** Используется XOR для определения изменений:

$$
f_{\text{переход}}(A, B, C, A', B', C') = (A \oplus A') \lor (B \oplus B') \lor (C \oplus C').
$$

Ниже приведён код на Python, который реализует вычисления для куба (решётки) булевых переменных и общего результата для силлогизма (Синтез, Антитезис, Тезис).

Код на Python

```python
from itertools import product

# Определение булевых функций
def synthesis(A, B, C):
    """Функция синтеза: S(A, B, C) = A OR (B AND C)"""
    return A or (B and C)

def antithesis(A, B, C):
    """Функция антитезиса: T(A, B, C) = (NOT A AND B) OR C"""
    return (not A and B) or C

def thesis(A, B, C):
    """Функция тезиса: P(A, B, C) = (A AND B) OR (NOT C)"""
    return (A and B) or (not C)

def result(A, B, C):
    """Общий результат силлогизма: R(A, B, C) = S AND T AND P"""
    S = synthesis(A, B, C)
    T = antithesis(A, B, C)
    P = thesis(A, B, C)
    return S and T and P

# Генерация всех возможных комбинаций переменных (для 3 переменных)
variables = list(product([0, 1], repeat=3))  # (A, B, C)

# Вычисление результатов для всех узлов
results = []
for A, B, C in variables:
    S = synthesis(A, B, C)
    T = antithesis(A, B, C)
    P = thesis(A, B, C)
    R = result(A, B, C)
    results.append((A, B, C, S, T, P, R))

# Вывод таблицы результатов
print(f"{'A':<3} {'B':<3} {'C':<3} {'S':<3} {'T':<3} {'P':<3} {'R':<3}")
for res in results:
    print(f"{res[0]:<3} {res[1]:<3} {res[2]:<3} {res[3]:<3} {res[4]:<3} {res[5]:<3} {res[6]:<3}")
```

Описание кода

1. **Функции логических операций:**
  - `synthesis(A, B, C)` реализует синтез.
  - `antithesis(A, B, C)` реализует антитезис.
  - `thesis(A, B, C)` реализует тезис.
  - `result(A, B, C)` объединяет синтез, антитезис и тезис с помощью логической конъюнкции (AND).

2. **Перебор всех комбинаций:**
  - `product([0, 1], repeat=3)` генерирует все комбинации значений \(A, B, C\) (000, 001, ..., 111).

3. **Вычисления для каждой комбинации:**
  - Для каждой комбинации переменных вычисляются значения \(S, T, P, R\).
  - Результаты записываются в список `results`.

4. **Вывод таблицы:**
  - Таблица выводит значения \(A, B, C, S, T, P, R\) для каждой комбинации.

Пример вывода
```
A  B  C  S  T  P  R
0  0  0  0  0  1  0
0  0  1  1  1  1  1
0  1  0  0  0  0  0
0  1  1  1  1  1  1
1  0  0  1  0  1  0
1  0  1  1  1  0  0
1  1  0  1  0  1  0
1  1  1  1  1  1  1
```

Визуализация графа переходов (дополнительно)

Если требуется визуализация куба и графа переходов, можно воспользоваться библиотекой **`networkx`** и **`matplotlib`**. Вот пример:

```python
import networkx as nx
import matplotlib.pyplot as plt

# Создание графа для куба
G = nx.Graph()

# Добавление узлов (состояния переменных)
for state in variables:
    G.add_node("".join(map(str, state)))

# Добавление рёбер (логические переходы между состояниями)
for state1 in variables:
    for state2 in variables:
        # Переход, если различается только одна переменная
        if sum(a != b for a, b in zip(state1, state2)) == 1:
            G.add_edge("".join(map(str, state1)), "".join(map(str, state2)))

# Рисование графа
pos = nx.spring_layout(G)  # Позиции узлов
nx.draw(G, pos, with_labels=True, node_color="lightblue", node_size=3000, font_size=10)
plt.title("Граф переходов (куб)")
plt.show()
```

Вот обновлённый код, который позволяет добавлять больше функций для других кубов и включает расширенную визуализацию, где результаты отображаются на узлах с цветовой кодировкой или подписями.

Обновлённый код

```python
from itertools import product
import networkx as nx
import matplotlib.pyplot as plt


# Определение булевых функций для нескольких кубов
def synthesis(*vars):
    """Функция синтеза: S(vars) = OR(vars[0], AND(vars[1:]))"""
    A, *rest = vars
    return A or all(rest)

def antithesis(*vars):
    """Функция антитезиса: T(vars) = (NOT(vars[0]) AND vars[1]) OR vars[-1]"""
    A, B, *rest = vars
    return (not A and B) or rest[-1]

def thesis(*vars):
    """Функция тезиса: P(vars) = AND(vars[:2]) OR NOT(vars[-1])"""
    return all(vars[:2]) or not vars[-1]

def result(*vars):
    """Общий результат силлогизма: R(vars) = S AND T AND P"""
    S = synthesis(*vars)
    T = antithesis(*vars)
    P = thesis(*vars)
    return S and T and P

# Дополнительная функция для нового куба (пример)
def custom_function(*vars):
    """Пример функции для нового куба: XOR всех переменных"""
    return sum(vars) % 2


# Генерация всех комбинаций для n переменных
def generate_states(n):
    """Генерация всех состояний для n переменных"""
    return list(product([0, 1], repeat=n))


# Построение графа булевого куба
def build_graph(states):
    """Построение графа для состояний"""
    G = nx.Graph()
    nodes = ["".join(map(str, state)) for state in states]

    # Добавление узлов
    G.add_nodes_from(nodes)

    # Добавление рёбер (если различается одна переменная)
    for state1 in states:
        for state2 in states:
            if sum(a != b for a, b in zip(state1, state2)) == 1:
                G.add_edge("".join(map(str, state1)), "".join(map(str, state2)))

    return G


# Вычисление для нескольких кубов
def compute_multiple_cubes(n):
    """Вычисления для всех кубов"""
    states = generate_states(n)
    results = []

    for state in states:
        S = synthesis(*state)
        T = antithesis(*state)
        P = thesis(*state)
        R = result(*state)
        C = custom_function(*state)  # Результат новой функции
        results.append((*state, S, T, P, R, C))
```

```
    return results


# Расширенная визуализация графа
def visualize_graph(G, results, n, title="Граф булевого куба"):
    """Визуализация графа с результатами"""
    pos = nx.spring_layout(G)  # Расположение узлов
    node_colors = []
    labels = {}

    for node in G.nodes:
        # Преобразование метки узла в состояние
        state = tuple(map(int, node))
        # Найти соответствующий результат
        for res in results:
            if state == res[:n]:
                # Цвет на основе результата общего (R)
                color = "green" if res[n+3] else "red"
                node_colors.append(color)
                # Подпись узла (показываем результаты всех функций)
                labels[node] = f"{node}\nS:{res[n]} T:{res[n+1]} P:{res[n+2]} R:{res[n+3]} C:{res[n+4]}"
                break

    # Рисование графа
    nx.draw(G, pos, with_labels=True, labels=labels, node_color=node_colors,
            node_size=3000, font_size=8, font_color="white")
    plt.title(title)
    plt.show()


# Основной код
if __name__ == "__main__":
    # Число переменных
    n = 4  # Укажите любое значение n (например, 4 для 4 переменных)

    # Генерация всех состояний
    states = generate_states(n)

    # Построение графа
    G = build_graph(states)

    # Вычисление результатов
    results = compute_multiple_cubes(n)

    # Вывод таблицы результатов
    print(f"{'State':<10} {'S':<3} {'T':<3} {'P':<3} {'R':<3} {'C':<3}")
    for res in results:
        state_str = "".join(map(str, res[:n]))
        print(f"{state_str:<10} {res[n]:<3} {res[n+1]:<3} {res[n+2]:<3} {res[n+3]:<3} {res[n+4]:<3}")

    # Визуализация графа
    visualize_graph(G, results, n, title=f"Граф булевого куба с {n} переменными")
```

Описание изменений

1. Новая функция для другого куба
Добавлена функция `custom_function`, которая реализует XOR всех переменных. Вы можете добавить больше таких функций, реализующих произвольные логические операции.

---

2. Расширенная визуализация
Визуализация теперь:
- Использует цветовые коды для узлов:
  - Зелёный (`green`) для узлов, где результат общего выражения (\(R \)) равен 1.
  - Красный (`red`) для узлов, где \(R = 0 \).
- Включает текстовые подписи для каждого узла:
  - Подпись показывает состояния переменных и значения всех функций (\(S, T, P, R, C \)).

---

3. Гибкость числа переменных
Число переменных \(n \) задаётся параметром. Все вычисления и визуализация автоматически адаптируются к размеру пространства состояний (\(2^n \)).

Пример вывода

Таблица
Для \(n = 4 \):
```
State    S  T  P  R  C
0000     0  0  1  0  0
0001     0  1  1  0  1
0010     0  0  1  0  1
0011     0  1  1  0  0
0100     0  0  0  0  1
0101     0  1  1  0  0
...
```

Граф
- Узлы окрашены в зависимости от результата \(R \): зелёный для \(R = 1 \), красный для \(R = 0 \).
- Узлы подписаны значениями всех функций (\(S, T, P, R, C \)).

- Для расширения кода, включающего дополнительные функции для других кубов, новые правила соединения рёбер, и добавления 3D-визуализации, можно воспользоваться следующими модификациями.

- ---

- ## **Обновленный код**

- ```python
- import matplotlib.pyplot as plt
- import networkx as nx
- from itertools import product
- from mpl_toolkits.mplot3d import Axes3D

- # Булевые функции для разных кубов
- def synthesis(*vars):
-     """Синтез: OR(vars[0], AND(vars[1:]))"""
-     A, *rest = vars
-     return A or all(rest)

- def antithesis(*vars):
-     """Антитезис: (NOT(vars[0]) AND vars[1]) OR vars[-1]"""
-     A, B, *rest = vars
-     return (not A and B) or rest[-1]

- def thesis(*vars):
-     """Тезис: AND(vars[:2]) OR NOT(vars[-1])"""
-     return all(vars[:2]) or not vars[-1]

- def custom_function_1(*vars):
-     """Кастомная функция: XOR всех переменных"""
-     return sum(vars) % 2

- def custom_function_2(*vars):
-     """Кастомная функция: (vars[0] AND vars[-1]) XOR (NOT(vars[1]))"""
-     return (vars[0] and vars[-1]) ^ (not vars[1])

- # Генерация всех состояний для n переменных
- def generate_states(n):
-     return list(product([0, 1], repeat=n))

- # Построение графа с зависимостями рёбер
- def build_graph(states, dependency_func=None):
-     G = nx.Graph()
-     nodes = ["".join(map(str, state)) for state in states]
-     G.add_nodes_from(nodes)

-     for state1 in states:
-         for state2 in states:
-             # Добавление ребра, если выполняется функция зависимости
-             if dependency_func:
-                 if dependency_func(state1, state2):
-                     G.add_edge("".join(map(str, state1)), "".join(map(str, state2)))
-             # Иначе соединяем состояния, отличающиеся на одну переменную
-             elif sum(a != b for a, b in zip(state1, state2)) == 1:
-                 G.add_edge("".join(map(str, state1)), "".join(map(str, state2)))
-     return G

```
# Правило соединения рёбер: зависимость от функций
def edge_dependency_rule(state1, state2):
    """Ребро добавляется, если XOR(state1, state2) = 1 для последней переменной"""
    xor_diff = [a ^ b for a, b in zip(state1, state2)]
    return xor_diff[-1] == 1 and sum(xor_diff) == 1

# Вычисления для всех функций
def compute_functions(states, functions):
    results = []
    for state in states:
        results.append((*state, *[func(*state) for func in functions]))
    return results

# Визуализация в 2D
def visualize_graph_2d(G, results, n, functions, title="2D Булев куб"):
    pos = nx.spring_layout(G)
    labels = {}
    node_colors = []

    for node in G.nodes:
        state = tuple(map(int, node))
        for res in results:
            if state == res[:n]:
                labels[node] = f"{node}\n" + " ".join([f"F{i+1}:{res[n+i]}" for i in range(len(functions))])
                node_colors.append("green" if res[n] else "red")  # Цвет для первой функции
                break

    nx.draw(G, pos, with_labels=True, labels=labels, node_color=node_colors,
            node_size=3000, font_size=8, font_color="white")
    plt.title(title)
    plt.show()

# Визуализация в 3D
def visualize_graph_3d(G, results, n, functions, title="3D Булев куб"):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")
    pos = nx.spring_layout(G, dim=3)

    node_colors = []
    labels = {}
    for node in G.nodes:
        state = tuple(map(int, node))
        for res in results:
            if state == res[:n]:
                labels[node] = f"{node}\n" + " ".join([f"F{i+1}:{res[n+i]}" for i in range(len(functions))])
                node_colors.append("green" if res[n] else "red")  # Цвет для первой функции
                break

    # 3D координаты
    xyz = [pos[node] for node in G.nodes]
    for i, (x, y, z) in enumerate(xyz):
        ax.text(x, y, z, list(G.nodes)[i], fontsize=8)
    edges = list(G.edges)
    for edge in edges:
        x_coords, y_coords, z_coords = zip(*[pos[edge[0]], pos[edge[1]]])
        ax.plot(x_coords, y_coords, z_coords, color="black")

    ax.scatter(*zip(*xyz), c=node_colors, s=100)
    plt.title(title)
    plt.show()


# Основной код
if __name__ == "__main__":
    n = 4  # Количество переменных
    states = generate_states(n)

    # Функции для вычислений
    functions = [synthesis, antithesis, thesis, custom_function_1, custom_function_2]

    # Построение графа
    G = build_graph(states, dependency_func=edge_dependency_rule)

    # Вычисления
    results = compute_functions(states, functions)

    # Таблица результатов
    print(f"{'State':<10} " + " ".join([f"F{i+1}" for i in range(len(functions))]))
    for res in results:
```

```
 state_str = "".join(map(str, res[:n]))
       function_values = " ".join(map(str, res[n:]))
       print(f"{state_str:<10} {function_values}")

    # Визуализация 2D
    visualize_graph_2d(G, results, n, functions, title="2D Булев куб с расширенными функциями")

    # Визуализация 3D
    visualize_graph_3d(G, results, n, functions, title="3D Булев куб с расширенными функциями")
```

Объяснение изменений

1. **Больше функций для других кубов:**
  - Добавлены функции `custom_function_1` и `custom_function_2` как примеры новых булевых операций.
  - Легко можно добавить дополнительные функции, просто определив их и включив в список `functions`.

2. **Специальные правила для соединений рёбер:**
  - Реализована функция `edge_dependency_rule`, которая добавляет ребро между состояниями, если их XOR-отличие соответствует заданному правилу.
  - Граф строится с учетом зависимости, заданной через эту функцию.

3. **3D-визуализация:**
  - Граф визуализируется в 3D с помощью `matplotlib` (модуль `Axes3D`).
  - Узлы расположены в трёхмерном пространстве, а подписи и цвета отображают состояние.

Как использовать

1. Измените количество переменных, задав значение `n`.
2. Добавьте новые функции в список `functions` и опишите их.
3. При необходимости уточните правила для соединений рёбер в функции `edge_dependency_rule`.

Этот код позволяет проводить расширенные вычисления и анализировать взаимодействие множества булевых кубов в удобном графическом формате.

Для формализации тезиса **"Жизнь не имеет обратного пути"** в контексте булевой алгебры и логических операций мы можем рассмотреть эту идею как утверждение, описывающее однонаправленное или необратимое состояние.

Интерпретация
- Булев подход позволяет интерпретировать это высказывание через состояние системы, где:
  - Переменные отражают различные аспекты или этапы жизни (например, события, действия, решения).
  - "Необратимость" можно выразить через логическую функцию, которая гарантирует, что переходы от одного состояния к другому не возвращают систему к предыдущему состоянию.

Математическая модель
1. **Состояния**: Пусть \(n \)-битные состояния \(S \) моделируют жизненные этапы.
2. **Тезис (функция)**:
  \[
  F_{\text{тезис}}(S_{\text{текущее}}, S_{\text{следующее}}) =
  \begin{cases}
  1, & \text{если } S_{\text{следующее}} > S_{\text{текущее}} \text{ (прогресс)} \\
  0, & \text{иначе (обратный путь запрещён).}
  \end{cases}
  \]

  Это условие можно вычислить через побитовое сравнение или через сравнение числовых значений состояний:
  \[
  F_{\text{тезис}}(S_1, S_2) = (S_1 < S_2)
  \]

3. **Булев анализ**:
  Если текущая жизнь — последовательность состояний, то логика утверждает, что любое состояние \(S_1 \) не должно возвращаться к предыдущему \(S_2 \).

Пример в коде

Реализуем функцию для вычисления данного тезиса.

```python
# Функция вычисления тезиса: Жизнь не имеет обратного пути
def thesis_life_not_reversible(state1, state2):
    """Определяет, является ли переход необратимым."""
    return int(int("".join(map(str, state1)), 2) < int("".join(map(str, state2)), 2))

# Тестовые состояния
states = generate_states(3)  # Все состояния для 3 бит
results = []

# Вычисление тезиса для всех пар
for state1 in states:
    for state2 in states:
        result = thesis_life_not_reversible(state1, state2)
        results.append((state1, state2, result))

# Вывод результатов
print(f"{'State1':<10} {'State2':<10} Thesis")
for res in results:
    s1 = "".join(map(str, res[0]))
    s2 = "".join(map(str, res[1]))
    print(f"{s1:<10} {s2:<10} {res[2]}")
```

Выходные данные
Результаты покажут, какие состояния допускаются (1), а какие нарушают правило (0).

Пример для трёхбитной системы:

```
State1    State2    Thesis
000       001       1
000       010       1
000       011       1
000       100       1
001       000       0
010       001       0
```

Расширение
1. **Графическая визуализация**:
   - Создайте граф состояний, где рёбра существуют только для допустимых переходов (тезис = 1).
   - Вы можете добавить цвет или форму узлов для наглядности.

2. **Логический анализ**:
   - Исследуйте, как тезис ограничивает возможные пути в пространстве состояний.
   - Рассмотрите последствия, если состояние нарушает тезис.

Давайте реализуем визуализацию для тезиса *"Жизнь не имеет обратного пути"* в виде графа. Мы покажем допустимые переходы между состояниями (рёбра для тезиса = 1) и отметим узлы, представляющие состояния.

Код с визуализацией:

```python
import matplotlib.pyplot as plt
import networkx as nx
from itertools import product
from mpl_toolkits.mplot3d import Axes3D

# Функция вычисления тезиса: Жизнь не имеет обратного пути
def thesis_life_not_reversible(state1, state2):
    """Определяет, является ли переход необратимым."""
    return int(int("".join(map(str, state1)), 2) < int("".join(map(str, state2)), 2))

# Генерация всех возможных состояний для n переменных
def generate_states(n):
    return list(product([0, 1], repeat=n))

# Построение графа для визуализации
def build_graph_thesis(states):
    G = nx.DiGraph()  # Используем направленный граф
    nodes = ["".join(map(str, state)) for state in states]
    G.add_nodes_from(nodes)

    # Добавляем рёбра на основе тезиса
    for state1 in states:
        for state2 in states:
            if thesis_life_not_reversible(state1, state2):  # Добавляем только допустимые переходы
                G.add_edge("".join(map(str, state1)), "".join(map(str, state2)))
    return G

# Визуализация 2D
def visualize_graph_2d_thesis(G, title="2D Визуализация тезиса"):
    pos = nx.spring_layout(G)  # Автоматическое расположение узлов
    nx.draw(G, pos, with_labels=True, node_color="skyblue", node_size=2000, font_size=10, font_color="black", arrowsize=20)
    plt.title(title)
    plt.show()

# Визуализация 3D
def visualize_graph_3d_thesis(G, title="3D Визуализация тезиса"):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection="3d")

    pos = nx.spring_layout(G, dim=3)  # Трёхмерное расположение узлов
    xyz = [pos[node] for node in G.nodes]

    # Визуализация узлов
    ax.scatter(*zip(*xyz), c="skyblue", s=100)

    # Подписи узлов
    for i, node in enumerate(G.nodes):
        x, y, z = xyz[i]
        ax.text(x, y, z, node, fontsize=10, ha="center", color="black")

    # Визуализация рёбер
    for edge in G.edges:
        x_coords, y_coords, z_coords = zip(*[pos[edge[0]], pos[edge[1]]])
        ax.plot(x_coords, y_coords, z_coords, color="black", alpha=0.8)

    plt.title(title)
    plt.show()

# Основной код
if __name__ == "__main__":
    n = 3  # Количество переменных (длина состояний)
    states = generate_states(n)
```

```
 # Построение графа
    G = build_graph_thesis(states)

    # 2D визуализация
    visualize_graph_2d_thesis(G, title="2D Визуализация: Жизнь не имеет обратного пути")

    # 3D визуализация
    visualize_graph_3d_thesis(G, title="3D Визуализация: Жизнь не имеет обратного пути")
```

Объяснение кода

1. **Генерация состояний:**
   - Все возможные состояния системы длины n генерируются как двоичные числа (например, для $n=3$): `000`, `001`, ..., `111`).

2. **Граф с направленными рёбрами:**
   - Используется направленный граф (`DiGraph`) для отражения однонаправленного характера жизни.
   - Рёбра добавляются только для переходов, где выполняется $F_{\text{тезис}}(S_1, S_2) = 1$.

3. **Визуализация 2D:**
   - Граф отображается с помощью `matplotlib` и `networkx`.
   - Узлы — состояния, рёбра — допустимые переходы.

4. **Визуализация 3D:**
   - Узлы расположены в трёхмерном пространстве (`spring_layout(dim=3)`).
   - Узлы подписаны, а рёбра соединяют их.

Как работает

1. **2D визуализация**:
   - Узлы (состояния) расположены на плоскости.
   - Рёбра показывают допустимые переходы согласно тезису.

2. **3D визуализация**:
   - Более сложное и интерактивное представление графа.
   - Узлы расположены в трёхмерном пространстве, что позволяет увидеть их взаимосвязи.

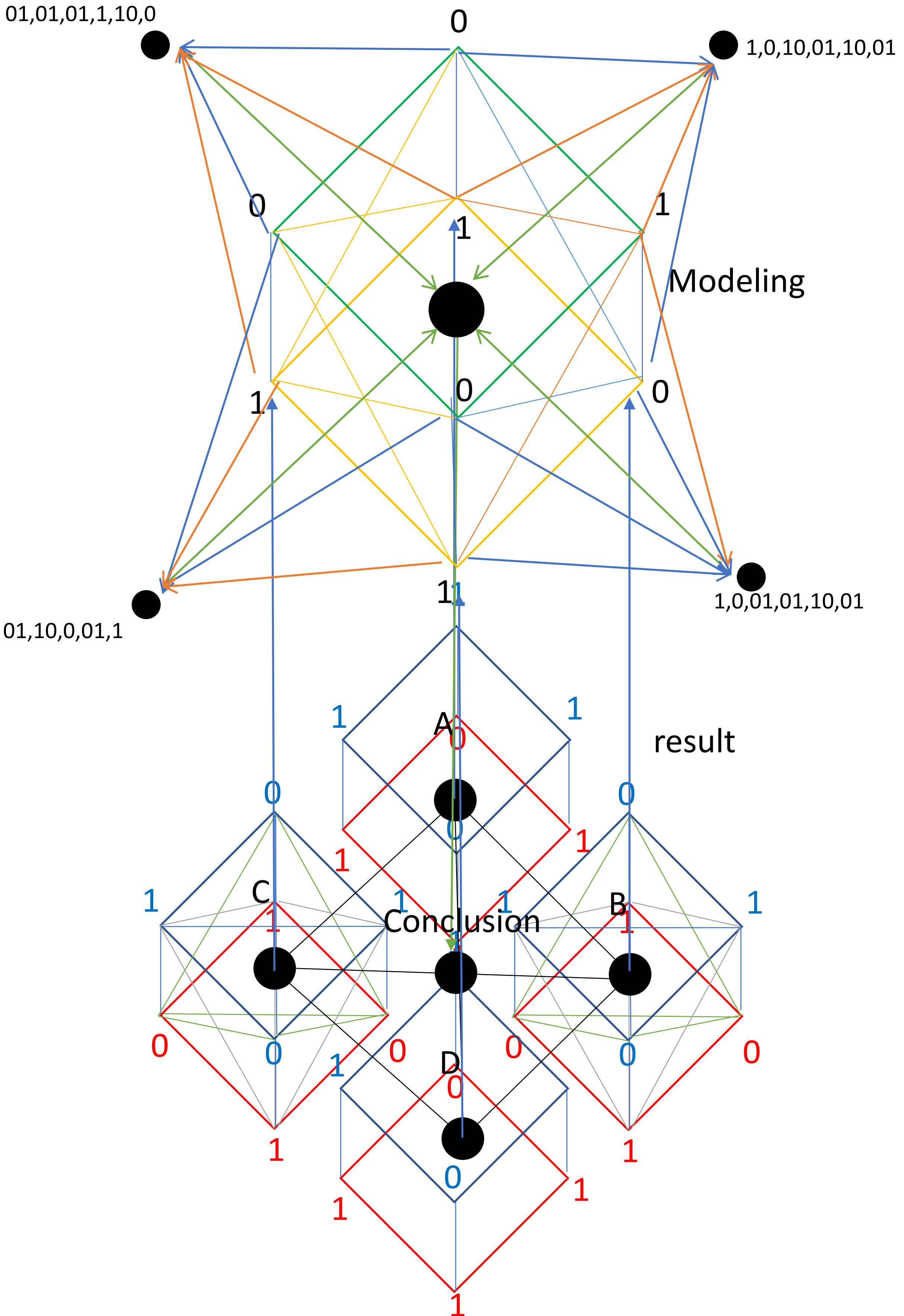
Пример результата

2D Граф
- Узлы: состояния $\{000, 001, ..., 111\}$.
- Рёбра: только допустимые переходы, где $S_1 < S_2$.

3D Граф
- Пространственное представление узлов и рёбер.

Если вы хотите добавить интерактивность (например, вращение 3D-графа), это можно сделать с помощью библиотек, таких как `plotly` или `pyvista`. Дайте знать, если потребуется!

Syllogism Quantum Logic



Федорченко Михаил Валерьевич