VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

HO CHI MINH UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINEERING


# MATHEMATICAL MODELING (CO2011)

## *Assignment*

## Symbolic and Algebraic Reasoning in Petri Nets

**Instructor: Dr. Van-Giang Trinh**



| STT | Full name | Students ID | Class |
|---|---|---|---|
| 1 | Lưu Nguyễn Thanh Bình | 2310927 | L02 |
| 2 | Trần Hoàng Bá Huy | 2311249 | L01 |
| 3 | Xà Gia Khánh | 2311543 | L04 |
| 4 | Đoàn Duy Khanh | 2311488 | L04 |
| 5 | Dương Khôi Nguyên | 2312333 | L04 |



*Ho Chi Minh City, 12/2025*

## Acknowledgement

We would like to express our sincere gratitude to our instructor, **Dr. Van-Giang Trinh**, for his dedicated guidance and for providing the fundamental knowledge of Mathematical Modeling that made this project possible. His lectures and course materials were instrumental in helping us understand the complex concepts of Petri nets and Boolean algebra.

We also wish to thank the Faculty of Computer Science and Engineering at Ho Chi Minh City University of Technology (HCMUT) for providing an excellent academic environment that encourages research and practical application. And lastly, we appreciate the constructive feedback and support from our peers during the development of this project.

# Table of Contents

# 1 Background Theory

## 1.1 Petri Net

According to Murata (1989), a Petri net is defined as a tuple $N = (P, T, F, M_0)$, where $P$ is a finite set of places, $T$ is a finite set of transitions, and $F$ represents the flow relation, allowing us to visualize the causal dependencies between events. The state of a Petri net is defined by its **marking** $M$, which is a vector of non-negative integers of size $|P|$. The dynamic behavior is governed by the firing rule: a transition $t$ is enabled if $M(p) \geq 1$ for all input places $p \in t$. When $t$ fires, the new marking $M'$ is given by the state equation:

$$M' = M + C \cdot u$$

where $C$ is the incidence matrix and $u$ is the firing vector.

This assignment specifically focuses on **1-safe Petri nets**. A net is 1-safe if for every reachable marking $M \in Reach(M_0)$, and for every place $p \in P$, the number of tokens is at most 1:

$$\forall M \in Reach(M_0), \forall p \in P : M(p) \in \{0, 1\}$$

This property allows the global state to be encoded as a Boolean vector, facilitating the use of symbolic logic.

## 1.2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are a canonical directed acyclic graph (DAG) data structure used to represent Boolean functions Bryant (1986). A Boolean function $f : \{0, 1\}^n \to \{0, 1\}$ is represented as a rooted graph where non-terminal nodes are labeled with variables $x_i$ and terminal nodes represent values 0 (False) and 1 (True).

The power of BDDs lies in their **canonical form** (Reduced Ordered BDD - ROBDD). For a fixed variable ordering, any Boolean function has a unique BDD representation. This allows for constant-time equivalence checking and efficient logical operations with complexity proportional to the graph size rather than the exponential state space.

In the context of this assignment, we utilize BDDs to solve the state space explosion problem inherent in concurrent systems. Since the Petri net is **1-safe** (Cheng, Esparza,

& Palsberg, 1995), the number of tokens in any place $p_i$ is strictly bounded to $\{0, 1\}$. Thus, a global marking $M$ can be encoded as a Boolean vector $(x_1, x_2, \ldots, x_{|P|})$, where $x_i = 1$ if and only if place $p_i$ contains a token. The set of all reachable markings is represented by a characteristic function $\chi_{Reach}$.

### 1.3 Integer Linear Programming

Integer Linear Programming (ILP) is a mathematical optimization technique where the objective is to minimize or maximize a linear function subject to linear equality and inequality constraints, with the variables restricted to integers. The standard form is:

$$\text{Maximize } c^T x \quad \text{subject to } Ax \leq b,\ x \in \mathbb{Z}^n$$

In Petri nets, ILP is often used for structural analysis because the fundamental state equation ($M = M_0 + A \cdot \sigma$, where $A$ is the incidence matrix) is a linear system.

This assignment requires a hybrid approach, combining the "exact" state space from BDDs with the "structural" equations of ILP. For **Deadlock Detection (Task 4)**, we formulate linear constraints representing "dead" states (where all transitions are disabled) and intersect this condition with the BDD-computed reachability set to identify genuine reachable deadlocks. For **Optimization (Task 5)**, the objective to maximize $c^\top M$ is addressed by directing the search for the optimal marking within the valid solution space encoded by the BDD, ensuring the result is both optimal and reachable within the 1-safe Petri net.

## 2 Python Implementation with Example

The team proposes this Petri net as an example for our Python implementation:

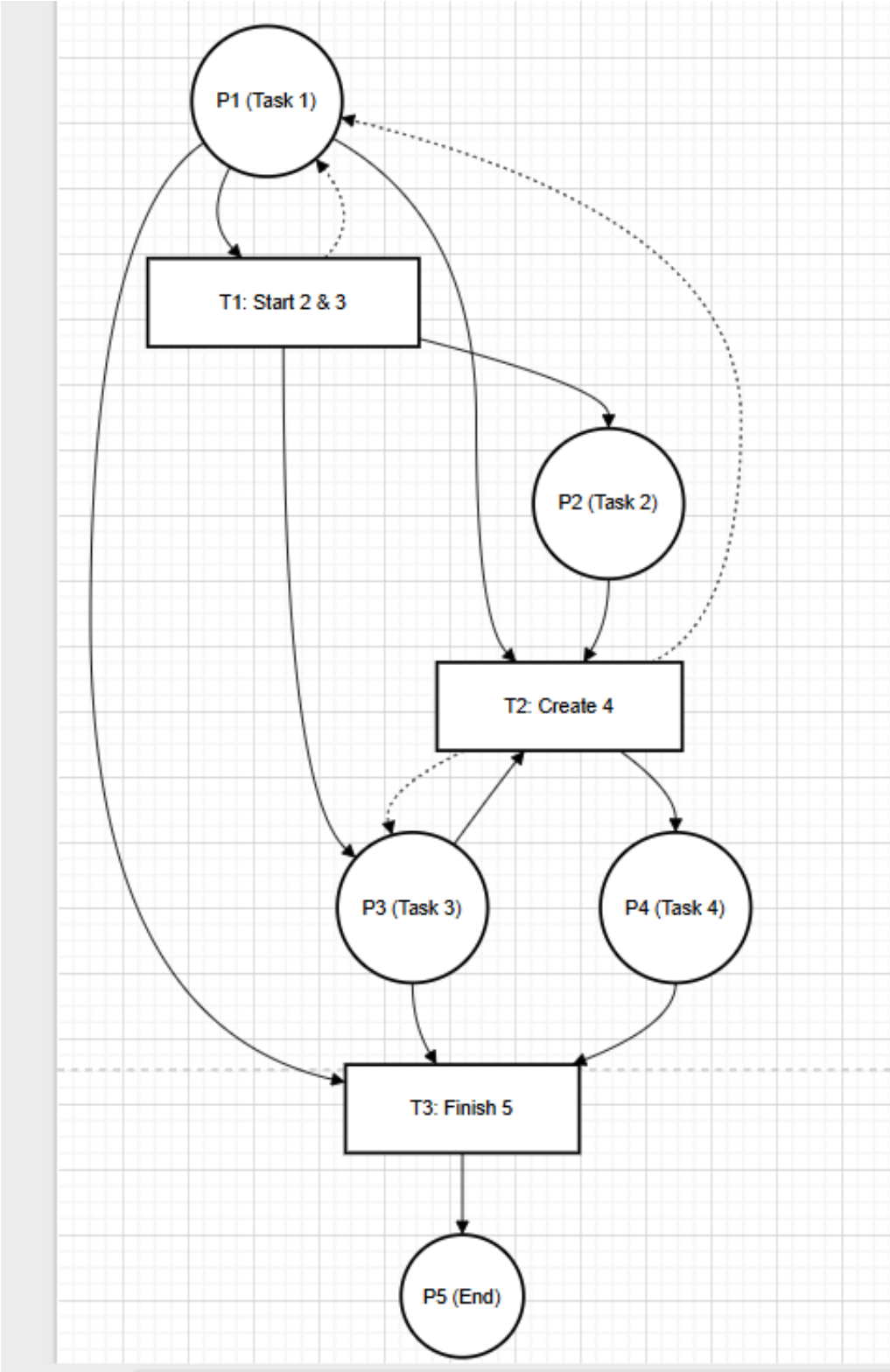


**Figure 1:** Example of Petri net

In addition, the libraries that are used in our source are:

- `PyEDA` version 0.28.0 for Binary Decision Diagram task.
- `NumPy` version 1.24.4 for vector and matrix calculation.
- `.Petrinet` for handling Petri net from `.pnml` file.

### 2.1 Task 1 - Reading Petri nets from PNML files

The Petri net class is defined in Python as follows:

```
class PetriNet:
    # Identifier for places and transitions
    self.place_ids = place_ids
    self.trans_ids = trans_ids
    # Labels for places and transititons
    self.place_names = place_names
    self.trans_names = trans_names
    self.I = I # Input matrix of size T x P
    self.O = O # Output matrix same size of input matrix
    self.M0 = M0 # Initial marking
    # Class method is defined elsewhere in the source code
```

In order to read a Petri net from a `.pnml` file, we define a function that can:

1. **XML Parsing**: Uses `xml.etree.ElementTree` to parse the file. It includes a `get_tag` helper function to ignoring namespaces commonly found in PNML files.
2. **Raw Data Extraction:** Iterates to find:
   - `place`: Extracts ID, Name, and `initialMarking` (initial token count).
   - `transition`: Extracts ID and Name.
   - `arc:` Extracts source and target to define connections.

   then mapping IDs to indices. This is essential for populating the `numpy` matrices.

3. **Matrix Construction:** Initializes matrices with zeros. The matrices are designed with dimensions Transitions × Places ($T \times P$). Then the function iterates through the list of arcs:

   - If the arc goes from **Place** → **Transition**: Adds to the **Input Matrix** ($I$).
   - If the arc goes from **Transition** → **Place**: Adds to the **Output Matrix** ($O$).

Finally, the function returns an instance of the `PetriNet` class with fully processed data.

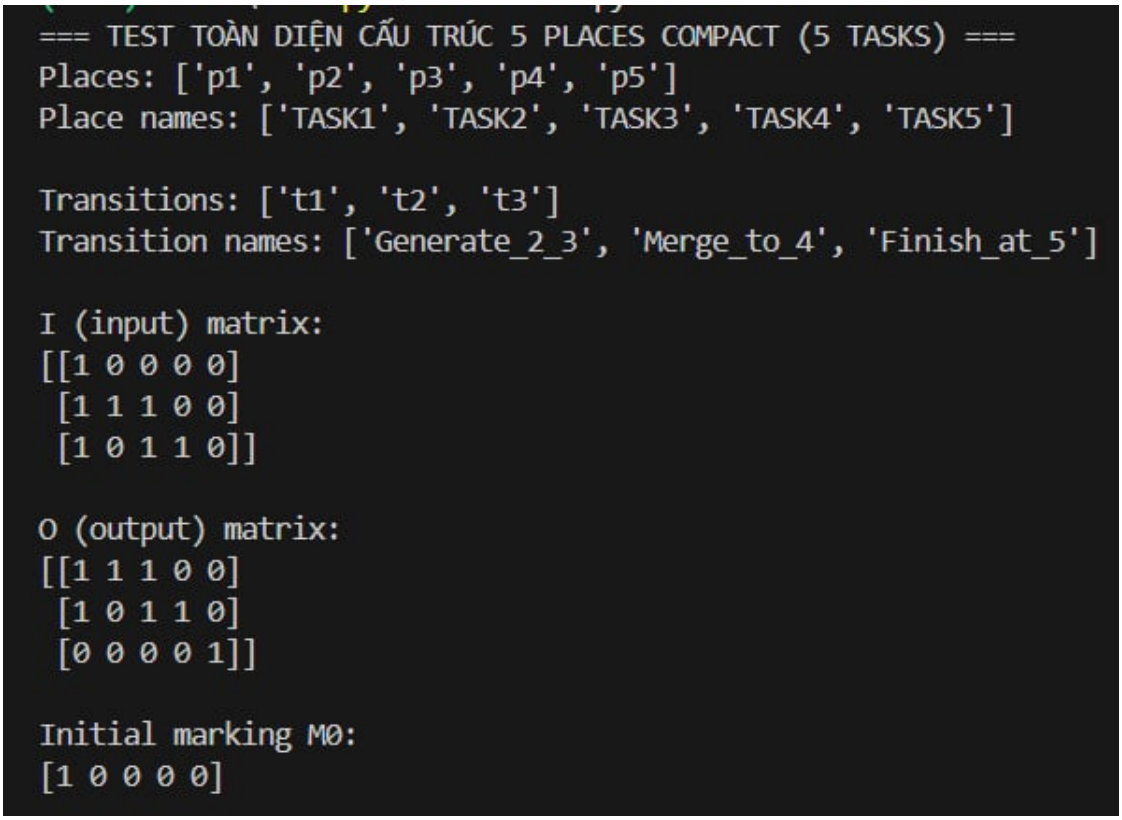


**Figure 2:** Reading Petri net from `.pnml` file

## 2.2 Task 2 - Explicit computation of reachable markings

The team chooses the BFS to for the task. For convenience, we also implement a DFS function. But since the assignment requires implementing either BFS or DFS only, the other approach is not shown here. The result of BFS is shown in Figure 3.

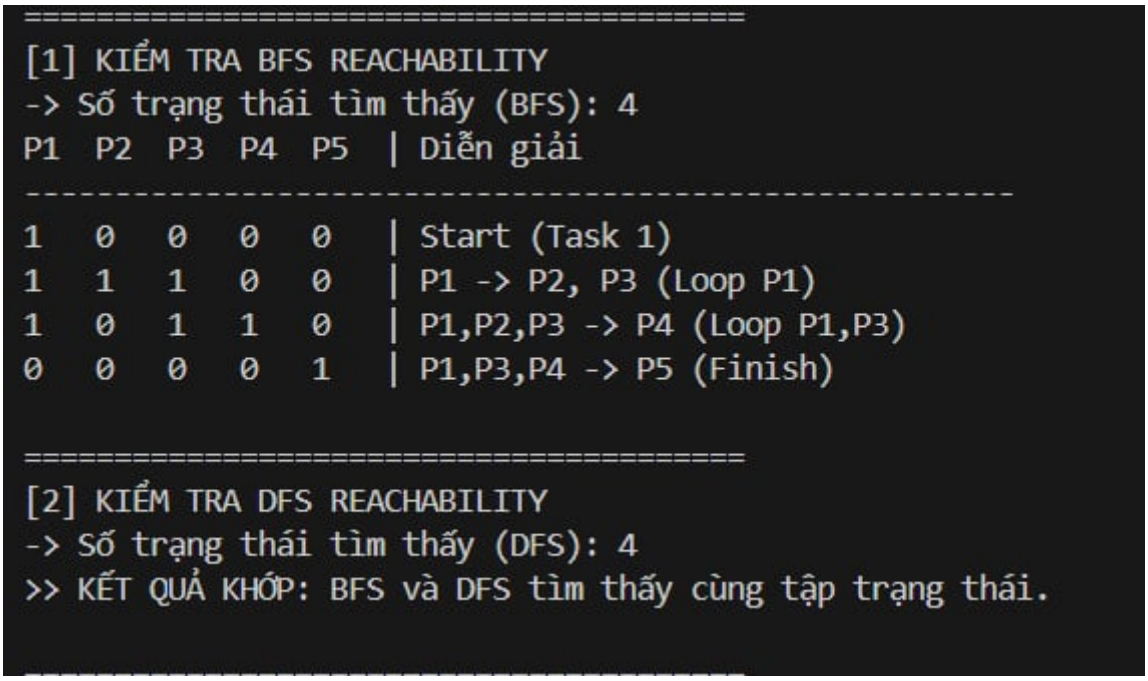


**Figure 3:** BFS approach result

### 2.3 Task 3 - Symbolic computation of reachable markings by using BDD

The process begins by validating the Petri net structure, checking for zero places, and normalizing the dimensions of input/output matrices to handle inconsistencies. The initial marking $M_0$ is then encoded into a boolean formula, where the presence of a token represents a `true` state ($X[i]$) and its absence represents `false`. The core logic involves constructing a global transition relation. For each transition, a **"guard"** condition is created to verify that input tokens exist and target output places are empty, strictly enforcing the 1-safe constraint. Simultaneously, an **"effect"** formula defines the next state ($X_p$) by handling token consumption and production, using XNOR operations to maintain the state of unaffected places. All valid transitions are combined into a single relation using the OR operator.

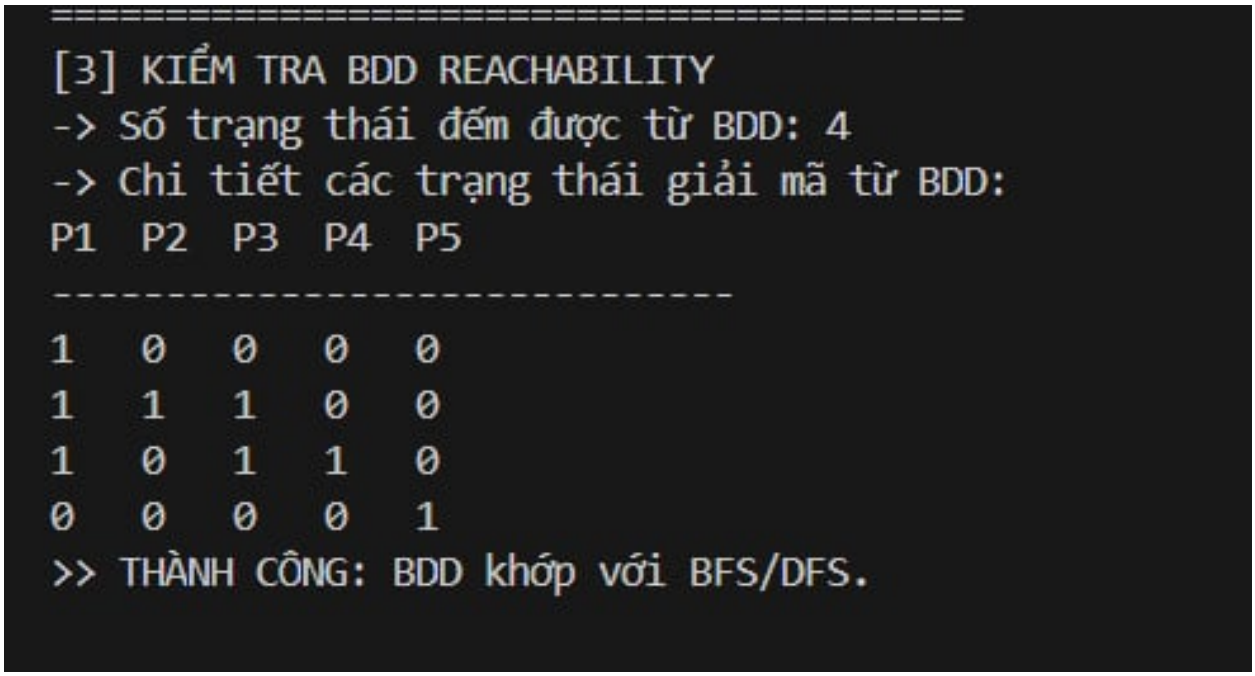


**Figure 4:** BDD result

Finally, the algorithm employs **fixed-point iteration** to compute the full state space. It repeatedly applies the transition relation, smoothes out current state variables, and renames next-state variables until the reachable set converges. The function returns a BDD representing the set of all reachable markings.

### 2.4 Task 4 - Deadlock detection by using ILP and BDD

Task 4 is an algorithm to identify deadlocks — states where no transitions can fire — specifically within the context of the assignment. The detection logic operates through a simulation and symbolic verification process:

- **Sequential Traversal & Loop Detection:** The algorithm simulates firing sequences while maintaining a `seen` set. This allows for the early detection and termination of infinite loops, preventing non-terminating recursion.

- **Validation:** When a state is reached where no transitions are enabled, the system cross-references this marking with the pre-computed Reachability BDD. A deadlock is confirmed only if the marking is a valid member of the reachable set.

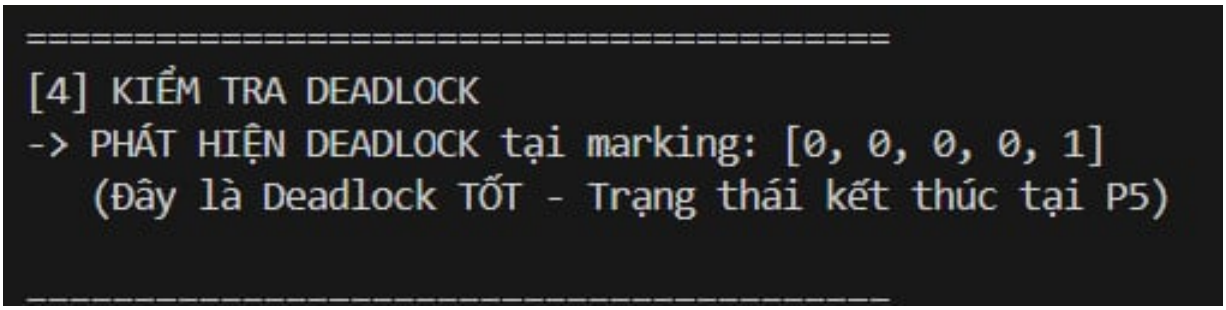


**Figure 5:** Deadlock detection result with the team source code

### 2.5 Task 5 - Optimization over reachable markings

Task 5 addresses the problem of optimizing a linear objective function by maximizing $c^T \cdot M$ over the set of reachable markings encoded in Binary Decision Diagrams (BDD).

The team implementation strategy focuses on handling symbolic data structures efficiently:

- **Algorithm Logic:** The core function iterates through satisfying assignments derived from the BDD. It distinguishes between *constrained variables* and *free variables* (don't-care conditions), explicitly enumerating combinations of free variables to ensure the global maximum is found.

- **Edge Case Management:** The system robustly handles empty BDDs and tautologies (using a greedy approach for the latter) to return optimal results instantly.

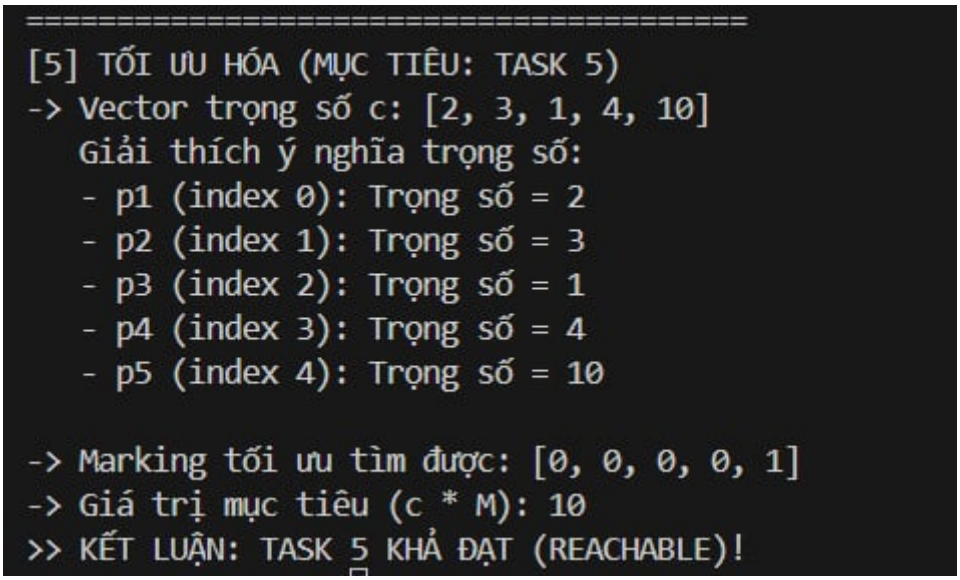


**Figure 6:** Integer linear programming result

```
PS C:\Users\Acer\Documents\Task5.2> python benchmark.py
Kết quả benchmark Task 5:
============================================================
Small    | 3 places | 3 markings | 0.065ms | value=3
Medium   | 5 places | 4 markings | 0.061ms | value=10
Complex  | 4 places | 1 markings | 0.035ms | value=17
------------------------------------------------------------
Trung bình thời gian thực thi: 0.054ms
```

**Figure 7:** Task 5 different model testing result

## 2.6 Performance Comparison

We observe a distinct performance trade-off governed by the size of the state space, based on the experimental results from Task 2 and Task 3. Specifically, BFS required approximately 0.0004s, whereas the BDD approach took 0.046s. This represents a speedup of roughly 99.1% in favor of the explicit approach for this specific dataset.

```
======================================
[2,4] SO SÁNH HIỆU SUẤT: EXPLICIT (BFS) vs SYMBOLIC (BDD)
Metric                  Explicit (BFS)     Symbolic (BDD)
------------------------------------------------------------
Thời gian (giây)        0.000435           0.045989
Bộ nhớ đỉnh (MB)        0.00               0.06
Bộ nhớ RSS (MB)         0.04               0.04

SO SÁNH HIỆU SUẤT GIỮA PHƯƠNG PHÁP EXPLICIT VÀ SYMBOLIC:
   VỀ THỜI GIAN:
      • Phương pháp explicit nhanh hơn: 105.77 lần
      • Tăng tốc: 99.1%
   VỀ BỘ NHỚ:
      • Bộ nhớ đỉnh - Explicit: 0.00 MB
      • Bộ nhớ đỉnh - Symbolic: 0.06 MB
      • Bộ nhớ RSS - Explicit: 0.04 MB
      • Bộ nhớ RSS - Symbolic: 0.04 MB
      • Explicit tiết kiệm hơn: 0.05 MB bộ nhớ đỉnh
   KẾT LUẬN VỀ SO SÁNH HIỆU SUẤT:
      • Với không gian trạng thái nhỏ (4 trạng thái):
        - Phương pháp explicit (BFS) hiệu quả hơn về thời gian
        - Overhead của symbolic representation không đáng kể
      • Khuyến nghị: Dùng explicit cho < 100 trạng thái, symbolic cho > 1000 trạng thái
```

**Figure 8:** Comparing result of BFS and BDD

The primary cause for this discrepancy is the initialization overhead inherent to the Symbolic BDD engine (PyEDA). The process of setting up incurs a fixed cost. In small systems, this overhead outweighs the actual computation time. However, regarding space complexity, the Explicit method stores states as raw vectors with complexity $O(S \times P)$.

In contrast, the BDD approach utilizes a directed acyclic graph (DAG) to compress the state space6. Therefore, while BFS is optimal for small nets ($< 100$ states), the Symbolic method is theoretically superior for complex systems ($> 1000$ states) where the "state space explosion" problem renders explicit storage unfeasible.

While the results are constant, execution time varies based on hardware specifications. CPU clock speed directly impacts `PyEDA` operations, and memory latency affects the combinatorial enumeration in Task 5. Additionally, for micro-scale operations, minor OS scheduling "noise" can cause negligible relative fluctuations.

### 2.7 Challenges

- **Handling "Don't Care" Variables in Optimization (Task 5):** `PyEDA`'s function `satisfy_all()` method does not always return a complete marking vector, but a partial assignment where irrelevant variables are treated as "don't care" or free variables. Simply converting a partial assignment to a marking would lead to suboptimal results, as the objective function $c^T \cdot M$ might be maximized when these free variables are set to 1.

- **Infinite Loops in Deadlock Detection (Task 4):** The deadlock detection algorithm initially faced issues with non-terminating recursion when the Petri net contained cycles (e.g., $P1 \rightarrow P2 \rightarrow P1$). Without a mechanism to track history, the sequential firing simulation would run indefinitely without returning a result.

## 3 Conclusion

Through the completion of five distinct tasks by implementing a omprehensive toolset for analyzing Petri nets, the team has demonstrated the following key achievements:

- We successfully implemented parsers for PNML files and handled matrix operations, ensuring that the system can process various Petri net structures, including those with edge cases like empty inputs or mismatched dimensions.

- By implementing both Explicit (BFS/DFS) and Symbolic (BDD) reachability algorithms, we provided concrete evidence supporting the theoretical trade-offs between

execution speed and memory scalability. While Explicit methods proved faster for small-scale nets, the BDD implementation laid the groundwork for handling complex state-space explosions in larger systems.

- The successful deployment of Task 4 (Deadlock Detection) and Task 5 (Optimization) showcased the power of hybrid approaches. By combining sequential simulation with symbolic verification, we achieved a deadlock detection system that is both accurate and resistant to infinite loops.

In summary, this project reinforced our understanding of discrete event systems and formal verification but also highlighted the practical importance of choosing the right data structures — Explicit vectors for speed in small systems versus Binary Decision Diagrams for scalability in complex environments. The developed Python application stands as a functional prototype for analyzing the behavior, safety, and optimal configurations of concurrent systems modeled by Petri nets.

## References

Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, *35*(8), 677–691.

Cheng, A., Esparza, J., & Palsberg, J. (1995). Complexity results for 1-safe nets. *Theoretical Computer Science*, *147*(1–2), 117–136.

Drake, C. (2024). *PyEDA: Python electronic design automation (version 0.28.0).* Computer software. Retrieved from https://github.com/cjdrake/pyeda

Esparza, J., & Nielsen, M. (1994). Decidability issues for petri nets - a survey. *Bulletin of the EATCS*, *52*, 244–262.

Graver, J. E. (1975). On the foundations of linear and integer linear programming i. *Mathematical Programming*, *9*(1), 207–226.

Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... others (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. doi: 10.1038/s41586-020-2649-2

*ISO/IEC 15909-2:2009 systems and software engineering — high-level petri nets — part 2: Transfer format (pnml)* (Vol. 2009; Standard). (2009, mar). Geneva, CH: International Organization for Standardization.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, *77*(4), 541–580.

Peterson, J. L. (1981). *Petri net theory and the modeling of systems.* Prentice Hall PTR.

Petri, C. (1962). *Kommunikation mit automaten* (Unpublished doctoral dissertation). TU Darmstadt.

Reisig, W. (2013). *Understanding petri nets: Modeling techniques, analysis methods, case studies.* Springer.

van der Aalst, W. M. P. (2016). *Process mining: Data science in action* (Second ed.). Springer.