ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

MÔ HÌNH HOÁ TOÁN HỌC - CO2011 - HK251

BÁO CÁO BÀI TẬP LỚN

Symbolic and Algebraic Reasoning in Petri Nets

Giảng viên hướng dẫn: ThS. Mai Xuân Toàn

Thành viên nhóm:

Đinh Quốc Thịnh	ID 2413309
Lữ Lê Kiều Diễm	ID 2410468
Hà Ngọc Khải	ID 2411553
Ngô Diệu Vy	ID 2414045

THÀNH PHỐ HỒ CHÍ MINH, THÁNG 12 NĂM 2025

LỜI MỞ ĐẦU

Mạng Petri (Petri Nets) là một mô hình toán học cơ bản và tinh tế được sử dụng rộng rãi để mô tả các hệ thống đồng thời, phân tán và hướng sự kiện. Chúng cung cấp một phương pháp trực quan để biểu diễn sự tương tác giữa các điều kiện (places) và sự kiện (transitions) thông qua luồng token , trở thành nền tảng trong kiểm chứng hệ thống và mô hình hóa quy trình.

Tuy nhiên, việc phân tích Mạng Petri gặp phải thách thức lớn là sự bùng nổ không gian trạng thái. Đây là hiện tượng số lượng các trạng thái khả tri (reachable markings) tăng theo cấp số mũ do tính đồng thời của hệ thống , khiến việc khám phá trạng thái tường minh trở nên bất khả thi ngay cả đối với các hệ thống nhỏ.

Để vượt qua giới hạn về khả năng mở rộng, bài tập lớn này tập trung vào việc tích hợp các kỹ thuật phân tích tiên tiến: Sơ đồ Quyết định Nhị phân (BDDs) để mã hóa không gian trạng thái một cách ký hiệu, và Quy hoạch Tuyến tính Nguyên (ILP) cho mục đích suy luận và tối ưu hóa.

Trong khuôn khổ môn Mô hình hóa Toán học, nhóm sẽ xây dựng một ứng dụng phân tích Mạng Petri 1-safe. Nhiệm vụ đầu tiên và cơ bản nhất là đọc Mạng Petri từ tệp PNML để xây dựng mô hình nội bộ. Các mục tiêu chính sau đó là:

1. Tính toán tập hợp các đánh dấu khả tri một cách tượng trưng bằng BDDs.

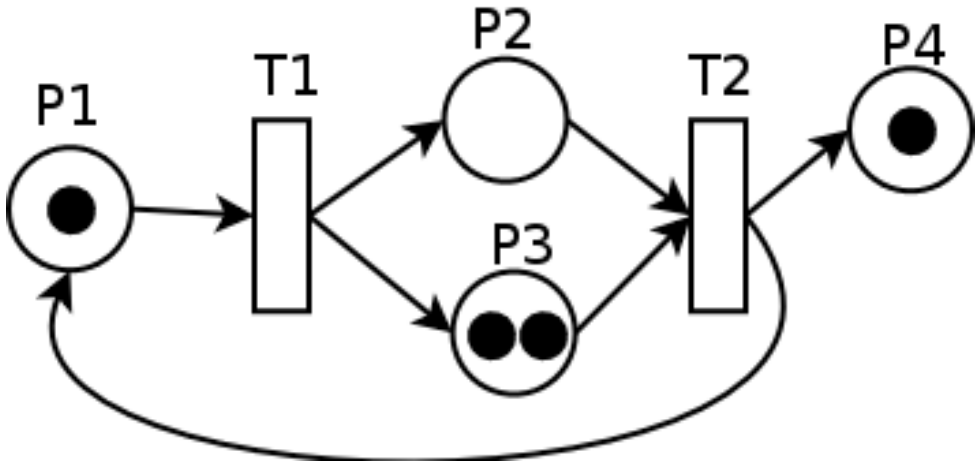
2. Phát hiện bế tắc (deadlock) thông qua sự kết hợp của công thức ILP và BDD.

3. Thực hiện tối ưu hóa tuyến tính trên các đánh dấu khả tri (Maximize $c^\top M$).

MỤC LỤC

1 Kiến thức lý thuyết

1.1 Petri nets



Hình 1.1: Sơ đồ minh họa 1 petri net

Mạng Petri ($PN = (P, T, F, W, M_0)$) là một mô hình toán học và đồ họa để mô tả các hệ thống đồng thời.

- P là tập **vị trí (places)** (điều kiện/tài nguyên).
- T là tập **chuyển đổi (transitions)** (sự kiện).
- F là tập **cung có hướng (directed arcs)**.
- W là hàm **trọng số (weight function)** gán giá trị cho cung.
- M_0 là **đánh dấu khởi tạo (initial marking)**.

Hoạt động của mạng được xác định qua trạng thái **đánh dấu** (*Marking*). Chuyển đổi t được **kích hoạt** (*enabled*) khi vị trí đầu vào có đủ token. Mạng **1-Safe** là mạng mà không vị trí nào chứa quá một token trong mọi trạng thái khả tri.

1.2 Tính Khả tri (Reachability) và Thách thức

- Đánh dấu Khả tri ($M \in Reach(M_0)$): Là bất kỳ trạng thái nào đạt được từ trạng thái M_0.
- Thách thức: **Bùng nổ trạng thái** khiến số lượng đánh dấu tăng theo cấp số mũ, làm các phương pháp phân tích tường minh kém hiệu quả.

1.3 Các cách tính toán khả tri

- **Tính toán Tường minh (*Explicit*)**: Sử dụng **BFS/DFS** để khám phá từng trạng thái. Đơn giản nhưng không hiệu quả khi không gian trạng thái lớn.
- **Tính toán Ký hiệu (*Symbolic*)**: Sử dụng **Binary Decision Diagrams (BDDs)** để nén và biểu diễn tập hợp trạng thái lớn một cách hiệu quả, khắc phục vấn đề bùng nổ trạng thái.

1.4 Quy hoạch tuyến tính nguyên (ILP)

- **ILP (*Integer Linear Programming*)**: Khuôn khổ tối ưu hóa sử dụng hệ thống ràng buộc tuyến tính, với biến quyết định là số nguyên.
- **Ứng dụng**: Dùng để **Phát hiện Deadlock** (tìm đánh dấu chết $M \in Reach(M_0)$) và **Tối ưu hóa** (tìm M^* lớn nhất tối ưu hóa $c^\top M$).

2 Thiết kế triển khai và Cấu trúc dữ liệu

2.1 Thiết kế triển khai

Thiết kế triển khai tập trung vào việc phân chia chương trình thành các lớp và chức năng có trách nhiệm rõ ràng. Lớp **PetriNet** là mô hình cơ sở, được sử dụng để đọc file PNML qua hàm `parse_pnml` (Task 1). Sau đó, nó được chuyển đổi thành **IndexedPetriNet** (đã được ánh xạ Place thành chỉ mục) để phục vụ các bài toán yêu cầu biểu diễn bằng vector nhị phân. Các module sử dụng các giải thuật chuyên biệt:

- **Task 2** sử dụng **Tìm kiếm theo chiều rộng (BFS)** tường minh để tìm reachable markings.
- **Task 3** sử dụng kỹ thuật **BDD (Binary Decision Diagram)** kết hợp với **lặp điểm cố định** để biểu diễn và tính tập khả đạt một cách hiệu quả đối với không gian trạng thái lớn.
- Cuối cùng, **Task 4** (Phát hiện Deadlock) và **Task 5** (Tối ưu hóa) đều dựa trên việc mô hình hóa bài toán thành **Lập trình Tuyến tính Nguyên (ILP)**, sử dụng solver Pulp để tìm kiếm lời giải trong tập các trạng thái khả đạt đã được tính trước đó.

Dòng chảy dữ liệu tuân theo trình tự:

Đọc file $\rightarrow$ Xây dựng mô hình $\rightarrow$ Tính khả đạt $\rightarrow$ Phân tích và Tối ưu.

2.2 Cấu trúc dữ liệu

Chương trình sử dụng các cấu trúc dữ liệu được lựa chọn kỹ lưỡng để tối ưu hóa việc lưu trữ và truy xuất dữ liệu trong từng giai đoạn phân tích. Ngôn ngữ lập trình được sử dụng là Python phiên bản 3.13.0.

- **Mô hình Mạng Petri:** Mạng Petri được mô hình hóa chủ yếu bằng `Set` và `defaultdict(set)` (cho ma trận kề Pre/Post), cho phép thao tác truy vấn và cập nhật đồ thị với độ phức tạp $O(1)$ trung bình.
- **Phân tích Khả đạt (BFS):** Trong giai đoạn này, `deque` được dùng làm hàng đợi và `Set` của `frozenset` được dùng để lưu trữ các trạng thái đã thăm nhằm kiểm tra tính duy nhất.Trạng thái được biểu diễn dưới dạng **Vector Nhị phân** (List các giá trị 0/1).
- **Biểu diễn Nén:** Đặc biệt, **BDD (Binary Decision Diagram)** là cấu trúc dữ liệu then chốt trong Task 3, cho phép nén và quản lý tập hợp lớn các trạng thái một cách hiệu quả, khác biệt hoàn toàn với việc lưu trữ tường minh.

3 Giải thích code và testcase

3.1 Các thư viện sử dụng

```
import xml.etree.ElementTree as ET
from collections import deque, defaultdict
import os
import itertools
from dd import autoref as _bdd
import pulp
import time
```

Listing 3.1: Các thư viện chính được sử dụng

- `xml.etree.ElementTree (ET)`: Đọc và phân tích (parse) file PNML ở dạng XML; dùng để duyệt cây XML, lấy thông tin `place`, `transition`, `arc`, `initialMarking` và xây dựng đối tượng `PetriNet`.
- `collections.deque`: Hàng đợi hai đầu dùng cho thuật toán BFS trong `compute_reachability`, hỗ trợ `append`/`popleft` với độ phức tạp $O(1)$.
- `collections.defaultdict`: Dùng `defaultdict(set)` để lưu `pre` và `post` của mạng Petri, giúp thêm cung (arc) mà không cần tự khởi tạo tập rỗng cho từng key.
- `os`: Kiểm tra sự tồn tại của file đầu vào bằng `os.path.exists(file_path)` trước khi parse PNML.
- `itertools`: Thư viện hỗ trợ thao tác tổ hợp/hoán vị; trong phiên bản code hiện tại chưa sử dụng trực tiếp.
- `dd.autoref (BDD)`: Cung cấp lớp `BDD` để làm việc với Binary Decision Diagram trong `BDDReachability`: khai báo biến `x`/`xp`, xây dựng quan hệ chuyển tiếp $R(x, x')$, tính reachable set và liệt kê marking từ một node BDD.
- `pulp`: Thư viện mô hình hoá và giải bài toán quy hoạch tuyến tính/quy hoạch nguyên (LP/ILP). Dùng trong `DeadlockDetector` và `ReachableOptimizer` để tìm deadlock và tối ưu marking.
- `time`: Đo thời gian thực thi các hàm (ví dụ `optimize_ilp`, `optimize_scan`) bằng cách lấy `time.time()` trước và sau khi chạy.

3.2 Task 1 – Parser PNML

```python
class PetriNet:
    def __init__(self): ...
    def add_place(self, p_id): ...
    def add_transition(self, t_id): ...
    def add_arc(self, source, target): ...
    def set_initial_marking(self, marking): ...
    def get_enabled_transitions(self, current_marking): ...
    def fire(self, current_marking, t): ...

def parse_pnml(file_path): ...
```

Listing 3.2: Các thành phần chính trong Task 1

Mục tiêu Task 1 đọc file PNML và xây dựng mô hình mạng Petri tương ứng trong Python.

Lớp `PetriNet` Lưu cấu trúc mạng Petri:

- `places`, `transitions`: tập các place/transition.
- `pre[t]`, `post[t]`: tập place đầu vào/đầu ra của mỗi transition.
- `initial_marking`: tập các place có token ban đầu.

Các phương thức hỗ trợ thêm phần tử, xác định các transition đang enabled và bắn (fire) một transition để sinh marking mới.

Hàm `parse_pnml(file_path)` Kiểm tra file tồn tại, parse XML, duyệt các thẻ `place`, `transition`, `arc`, `initialMarking` và điền dữ liệu vào một đối tượng `PetriNet`. Hàm trả về mạng Petri đã được dựng từ file PNML.

3.3 Task 2 – Reachability bằng BFS

```python
def compute_reachability(net): ...
```

Listing 3.3: Hàm reachability chính trong Task 2

Mục tiêu Tính tập marking reachable từ marking ban đầu bằng thuật toán BFS trên không gian trạng thái.

Ý tưởng chính

- Khởi tạo marking ban đầu `M0`, đưa vào `queue` và `visited`.
- Lặp: lấy một marking ra khỏi queue, tìm các transition enabled, bắn từng transition để sinh marking mới.
- Nếu marking mới chưa được thăm thì thêm vào `visited` và `queue`. Đồng thời tăng bộ đếm số cạnh.
- Kết quả: trả về tập `visited` (tất cả marking reachable) và số cạnh `edges_count`.

3.4 Task 3 – BDD Reachability

```
class IndexedTransition:
    def __init__(self, pre=None, post=None): ...

class IndexedPetriNet:
    def __init__(self, places, init_marking, transitions): ...

def convert_to_indexed(net): ...

class BDDReachability:
    def __init__(self, net: IndexedPetriNet): ...
    def marking_to_bdd(self, marking, prime=False): ...
    def build_transition_relation(self): ...
    def compute_reachable_bdd(self): ...
    def enumerate_markings(self, bdd_node, limit=1_000_000): ...
```

Listing 3.4: Các thành phần chính trong Task 3

Mục tiêu Biểu diễn và tính toán tập reachable markings bằng BDD để tận dụng tính nén và các phép toán tập hợp trên BDD, thay vì liệt kê toàn bộ bằng BFS.

Chuyển sang dạng chỉ số `convert_to_indexed(net)`:

- Sinh `place_list` và ánh xạ place → chỉ số.
- Mỗi transition được đổi sang `IndexedTransition` với `pre`, `post` là danh sách chỉ số place.
- Tạo `IndexedPetriNet` với số place, marking ban đầu dạng vector 0/1 và danh sách transition đã đánh chỉ số.

Lớp `BDDReachability`

- Khai báo biến BDD `x0,...,x(n-1)` cho trạng thái hiện tại và `xp0,...,xp(n-1)` cho trạng thái kế tiếp.
- `build_transition_relation()`: xây dựng quan hệ $R(x, x')$ tương ứng với tất cả các transition của mạng.
- `compute_reachable_bdd()`: lặp cố định

$$S_{k+1} = S_k \cup \text{Post}(S_k)$$

  trên BDD cho đến khi hội tụ, trong đó $\text{Post}(S_k)$ được tính bằng phép $\exists x.\,(S_k \wedge R)$ rồi đổi tên $x' \to x$.
- `enumerate_markings()`: dùng `bdd.pick()` để trích một số marking cụ thể từ BDD (tối đa `limit` marking).

3.5 Task 4 – Phát hiện deadlock bằng ILP

```
class DeadlockDetector:
    def __init__(self, net: IndexedPetriNet, reachableMarkings): ...
    def find_deadlock_ilp(self): ...
```

Listing 3.5: Lớp phát hiện deadlock

Mục tiêu: Tìm một marking trong tập reachable sao cho **không còn transition nào enabled** (deadlock) bằng cách mô hình hóa điều kiện này thành bài toán ILP.

Ý tưởng chính

- Dùng biến `M[p]` để biểu diễn marking được chọn, và `y[i]` để chọn một marking trong `reachableMarkings`, tương tự như Task 5.
- Ràng buộc $\sum_i y[i] = 1$ và $M[p] = \sum_i y[i] \cdot \text{reach}[i][p]$ đảm bảo `M` là một marking reachable.
- Với mỗi transition `t` có `pre` không rỗng, thêm ràng buộc

$$\sum_{p \in pre(t)} M[p] \leq |pre(t)| - 1$$

để chắc chắn rằng `t` không thể enabled (không đủ token ở tất cả input places).

- Hàm mục tiêu bằng 0, chỉ cần tìm một nghiệm thoả ràng buộc. Nếu ILP có nghiệm, marking tương ứng là một deadlock reachable.

3.6 Task 5 – Reachable Optimization

```
class ReachableOptimizer:
    def __init__(self, net, reachableMarkings, weights): ...
    def optimize_ilp(self): ...
    def optimize_scan(self): ...
```

Listing 3.6: Lớp tối ưu hoá trên tập reachable

Mục tiêu Tìm một marking reachable sao cho **tổng trọng số các place đang có token là lớn nhất**, với trọng số được cho bởi dict `weights`.

Khởi tạo `ReachableOptimizer` Lưu lại:

- `net`: mạng Petri đã đánh chỉ số.
- `reachableMarkings`: danh sách các marking reachable.
- `self.w`: vector trọng số độ dài `net.nPlaces`, ánh xạ từng place trong `place_list` sang trọng số tương ứng (mặc định 0 nếu không có trong `weights`).

Hàm `optimize_ilp()`

- Xây dựng bài toán ILP với:
  - `M[p]`: marking được chọn.
  - `y[i]`: chọn đúng một marking trong `reachableMarkings`.
- Hàm mục tiêu:
$$\max \sum_p w_p \cdot M[p]$$
kèm ràng buộc $\sum_i y[i] = 1$ và $M[p] = \sum_i y[i] \cdot \text{reach}[i][p]$.
- Giải bằng `pulp`, trả về marking tối ưu, giá trị tối ưu và thời gian thực thi.

Hàm `optimize_scan()`

- Duyệt tuần tự từng marking `m` trong `reachableMarkings`.
- Với mỗi `m`, tính giá trị $\sum_p w_p \cdot m[p]$ và giữ lại marking có giá trị lớn nhất.
- Trả về marking tốt nhất kèm giá trị tương ứng và thời gian chạy.

Ghi chú Task 5 cho phép so sánh hai cách tiếp cận:

- Dùng ILP (`optimize_ilp`) với mô hình toán rõ ràng, dễ mở rộng thêm ràng buộc.
- Dùng quét tuyến tính (`optimize_scan`) đơn giản, không phụ thuộc solver nhưng tốn thời gian tuyến tính theo số marking.

3.7 Chạy thử chương trình

3.7.1 Trường hợp 1: Simple Fork/Split Pattern(test_002.pnml)

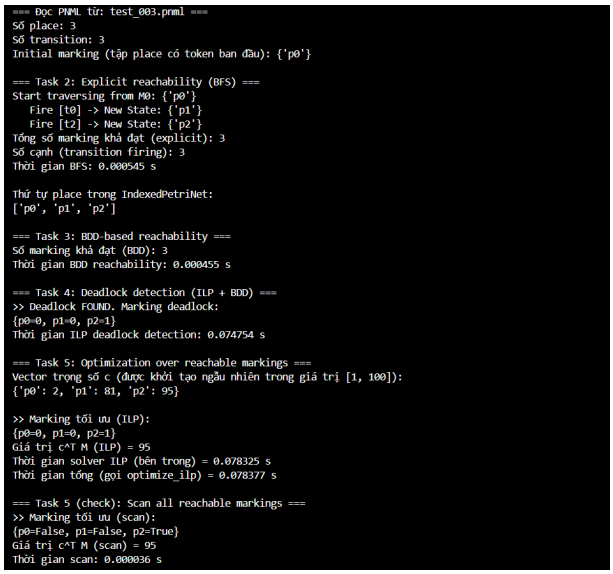
3 places, 1 transition. Token từ p0 split thành 2 token tại p1 và p2. Reachability set: 2 trạng thái.

```
=== Đọc PNML từ: test_002.pnml ===
Số place: 3
Số transition: 1
Initial marking (tập place có token ban đầu): {'p0'}

=== Task 2: Explicit reachability (BFS) ===
Start traversing from M0: {'p0'}
   Fire [t0] -> New State: {'p2', 'p1'}
Tổng số marking khả đạt (explicit): 2
Số cạnh (transition firing): 1
Thời gian BFS: 0.000204 s

Thứ tự place trong IndexedPetriNet:
['p0', 'p1', 'p2']

=== Task 3: BDD-based reachability ===
Số marking khả đạt (BDD): 2
Thời gian BDD reachability: 0.000988 s

=== Task 4: Deadlock detection (ILP + BDD) ===
>> Deadlock FOUND. Marking deadlock:
{p0=0, p1=1, p2=1}
Thời gian ILP deadlock detection: 0.086140 s

=== Task 5: Optimization over reachable markings ===
Vector trọng số c (được khởi tạo ngẫu nhiên trong giá trị [1, 100]):
{'p0': 80, 'p1': 6, 'p2': 76}

>> Marking tối ưu (ILP):
{p0=0, p1=1, p2=1}
Giá trị c^T M (ILP) = 82
Thời gian solver ILP (bên trong) = 0.078001 s
Thời gian tổng (gọi optimize_ilp) = 0.078072 s

=== Task 5 (check): Scan all reachable markings ===
>> Marking tối ưu (scan):
{p0=False, p1=True, p2=True}
Giá trị c^T M (scan) = 82
Thời gian scan: 0.000013 s
```

Hình 3.1: Output khi chạy test_002.pnml

3.7.2 Trường hợp 2: Cycle with Choice(test_003.pnml)

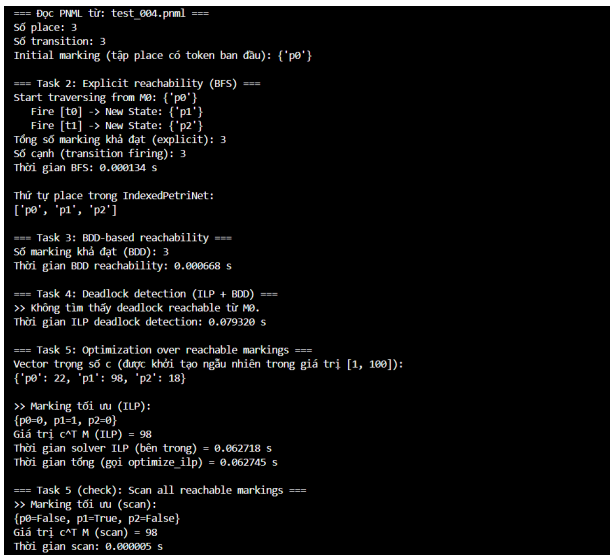
Vòng lặp $p0 \to t0 \to p1 \to t1 \to p0$. Có conflict tại p1, khả năng deadlock tại p2.



Hình 3.2: Output khi chạy test_003.pnml

3.7.3 Trường hợp 3: Simple Cycle (test_004.pnml)

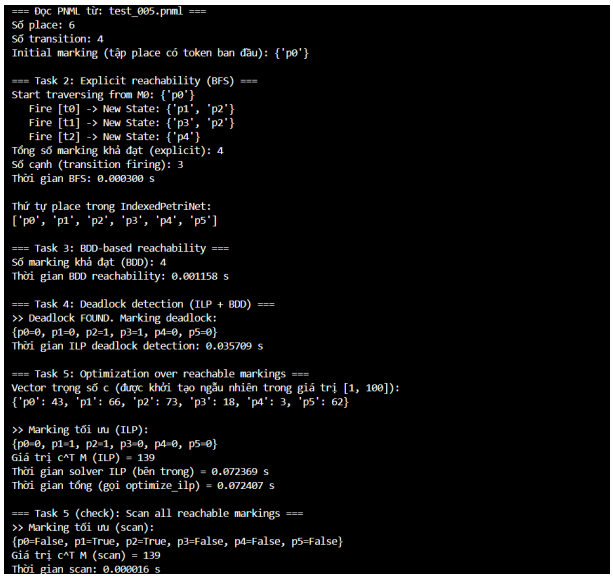
Vòng lặp hoàn chỉnh: $p0 \to t0 \to p1 \to t1 \to p2 \to t2 \to p0$.



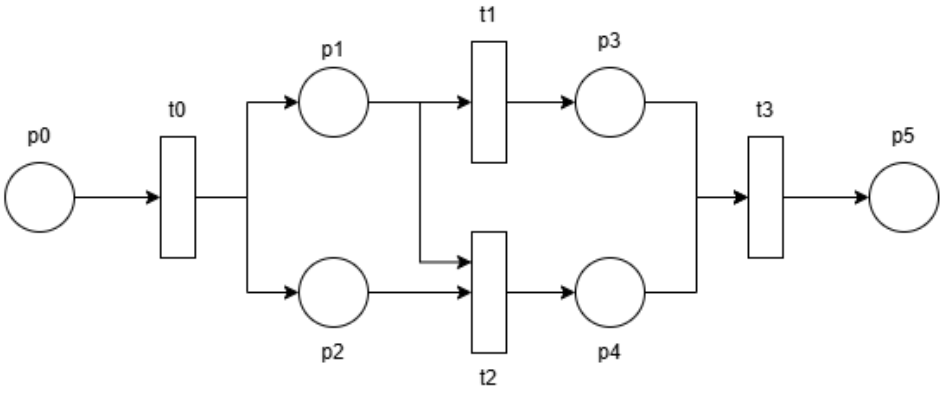
Hình 3.3: Output khi chạy test_004.pnml

3.7.4 Trường hợp 4: Fork-Join with Synchronization(test_005.pnml)

Fork tại t0, conflict tại p1, join tại t3. Có khả năng deadlock nếu chọn sai sequence.



Hình 3.4: Output khi chạy test_005.pnml



Hình 3.5: Sơ đồ minh họa test 5

Lưu ý: Phần chạy thử được đề cập trong báo cáo chỉ trình bày các testcases đơn giản để đảm bảo tính đúng đắn trong việc implement của nhóm. Các testcases phức tạp hơn sẽ bao gồm trong file zip được gửi cùng báo cáo này (có kèm theo ghi chú cụ thể).

4 Kết luận

4.1 Kết quả đạt được

Bài tập lớn đã được hoàn thành với các mục tiêu chính:

- **Xây dựng Mô hình Toàn diện:** Đã phát triển thành công một ứng dụng phân

tích Mạng Petri 1-an toàn, bao gồm đọc file PNML, biểu diễn nội bộ các thành phần (places, transitions, flow relations).

- **Phân tích Reachability:** Triển khai cả phương pháp khám phá hiển nhiên (BFS/DFS) và phương pháp tính toán tượng trưng sử dụng BDDs. Kết quả thực nghiệm cho thấy BDDs cung cấp một cách mã hóa và khám phá không gian trạng thái hiệu quả hơn về mặt thời gian và bộ nhớ so với cách tiếp cận hiển nhiên.
- **Giải quyết bài toán nâng cao:**
  - Thành công trong việc kết hợp công thức ILP với BDD để phát hiện tắc nghẽn (deadlock detection).
  - Đã giải quyết bài toán tối ưu hóa tuyến tính trên tập hợp các trạng thái đạt được $Reach(M_0)$.

4.2 Thách thức và Cải tiến tiềm năng

Thách thức chính: Khó khăn tập trung vào việc tối ưu hóa cấu trúc BDD (thông qua lựa chọn thứ tự biến) để giữ cho biểu diễn tập hợp trạng thái đạt được luôn nhỏ gọn. Bên cạnh đó, việc tích hợp hiệu quả ràng buộc từ BDD vào mô hình ILP cũng đòi hỏi sự tinh chỉnh về mặt thuật toán.

Cải tiến chính:

- **Tối ưu hóa BDD nâng cao:** Trong các ứng dụng sử dụng BDD để biểu diễn các tập hợp lớn (như tập trạng thái có thể đạt được $Reach(M_0)$), thứ tự của các biến (tương ứng với các places trong Mạng Petri) ảnh hưởng cực kỳ lớn đến kích thước của BDD. Nếu thứ tự biến không tối ưu, BDD có thể phình to theo cấp số nhân, làm mất đi lợi thế của phương pháp tượng trưng. Có thể áp dụng kỹ thuật sắp xếp lại biến động (Dynamic Variable Reordering), kỹ thuật được tích hợp trong các thư viện BDD tiên tiến (như CUDD). Thay vì giữ một thứ tự cố định ban đầu, thư viện sẽ tự động và liên tục điều chỉnh thứ tự các biến trong quá trình xây dựng BDD.
- **Mở rộng phạm vi mô hình:** Mở rộng mô hình để xử lý các lớp Mạng Petri tổng quát hơn (ví dụ: các mạng k-an toàn với $k > 1$). Việc này đòi hỏi sử dụng các công cụ biểu diễn tượng trưng tiên tiến hơn như Biểu đồ Quyết định Số nguyên (IDD) hoặc ADD, vốn được thiết kế để mã hóa các giá trị số nguyên thay vì chỉ Boolean.