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

BÀI TẬP LỚN
MÔ HÌNH HÓA TOÁN HỌC (CO2011)

ĐỀ TÀI

Symbolic and Algebraic Reasoning in Petri Nets

Nhóm 72 – Học Kỳ HK251
Ngày nộp: 05/12/2025

GVHD: Thầy Mai Xuân Toàn

Sinh viên thực hiện	Mã số sinh viên
Võ Đặng Thanh Bình	2310316
Phạm Minh Đức	2310797
Nguyễn Thanh Giang	2310828
Trần Quang Vinh	2313932
Phạm Công Võ	2313946

TP. Hồ Chí Minh, Tháng 12/2025

PHÂN CÔNG NHIỆM VỤ VÀ KẾT QUẢ THỰC HIỆN ĐỀ TÀI CỦA TỪNG THÀNH VIÊN NHÓM 72

STT	Họ và tên	MSSV	Nhiệm vụ	Kết quả	Chữ ký
1	Võ Đặng Thanh Bình	2310316	Hiện thực Task5: Optimization over reachable markings	100%	
2	Phạm Minh Đức	2310797	Hiện thực Task3: Symbolic computation of reachable markings by using BDD	100%	
3	Nguyễn Thanh Giang	2310828	Hiện thực Task3: Symbolic computation of reachable markings by using BDD	100%	
4	Trần Quang Vinh	2313932	Hiện thực Task4: Deadlock detection by using ILP and BDD	100%	
5	Phạm Công Võ	2313946	Hiện thực Task1: Reading Petri nets from PNML files + Task2: Explicit computation of reachable markings + Report	100%	

Mục lục

1 Giới thiệu tổng quan **4**

2 Nền tảng lý thuyết **4**
2.1 Petri Nets Fundamentals . . . 4
2.2 Explicit Reachability Analysis . . . 5
2.3 Binary Decision Diagrams (BDDs) . . . 6
2.4 Integer Linear Programming (ILP) . . . 7

3 Kiến trúc hệ thống và Cấu trúc dữ liệu **7**
3.1 Kiến Trúc Hệ Thống . . . 7
3.2 Cấu Trúc Dữ Liệu . . . 8
3.2.1 Core Data Structures . . . 8
3.2.2 Biểu Diễn Marking . . . 9
3.2.3 BDD Encoding Scheme . . . 10
3.3 Task 1: PNML Parser . . . 10
3.4 Task 2: Explicit Reachability Analysis . . . 11
3.5 Task 3: BDD-based Symbolic Reachability . . . 12
3.6 Task 4 – Deadlock Detection Using ILP and BDD . . . 14
3.7 Task 5: Optimization over Reachable Markings . . . 16

4 Kết quả thực nghiệm và thảo luận hiệu năng **18**
4.1 Môi trường thực nghiệm . . . 18
4.1.1 Cấu hình hệ thống . . . 18
4.1.2 Bộ test cases . . . 18
4.1.3 Task 1: PNML Parsing . . . 19
4.1.4 Task 2: Explicit Reachability . . . 19
4.1.5 Task 3: BDD Symbolic Reachability . . . 20
4.1.6 Task 4: Deadlock Detection . . . 21
4.1.7 Task 5: Optimization over Reachable Markings . . . 22
4.2 Đánh giá Hiệu Năng . . . 23
4.2.1 Thông số các mô hình Petri . . . 23
4.2.2 Thời gian thực thi (Execution Time) . . . 23
4.2.3 Bộ nhớ sử dụng (Memory Usage) . . . 24
4.2.4 Phát hiện deadlock . . . 26
4.3 Nhận xét tổng kết . . . 26

5 Kết luận và hướng phát triển **27**
5.1 Tổng kết và đánh giá đóng góp . . . 27
5.2 Khó khăn gặp phải . . . 27
5.3 Hướng phát triển trong tương lai . . . 27

Danh sách hình vẽ

1 Minh họa một Petri net đơn giản gồm places, transitions và tokens. 4
2 Minh họa Reachability Graph của một Petri net nhỏ. 5
3 Minh họa cấu trúc một BDD đơn giản. 7
4 Kết quả kiểm thử Task1 19
5 Kết quả Output Testcase_2 19
6 Kết quả kiểm thử Task 2 19
7 Kết quả Output Testcase_5 20
8 Kết quả kiểm thử Task 3 20
9 Output BDD Reachability Testcase_5 20
10 Kết quả kiểm thử Task 4 21
11 Deadlock Detection Testcase_2 21
12 Deadlock Detection Testcase_3 21
13 Kết quả kiểm thử Task 5 22
14 Optimization Output Testcase_4 22
15 Biểu Đồ Thời Gian Thực Thi (ms) 24
16 Biểu Đồ Bộ Nhớ Sử Dụng (KB) 25

1 Giới thiệu tổng quan

Petri nets là một mô hình toán học trực quan và hiệu quả để mô tả các hệ thống đồng thời, phân tán và hướng sự kiện. Được Carl Adam Petri đề xuất từ những năm 1960, Petri nets hiện nay được ứng dụng rộng rãi trong phương pháp hình thức, kiểm chứng hệ thống, mô hình hóa quy trình và nhiều lĩnh vực khác.

Về mặt lý thuyết, Petri nets kết nối nhiều lĩnh vực như lý thuyết đồ thị, hệ rời rạc, đại số tuyến tính và suy luận logic, cho phép mô hình hóa và phân tích các bài toán quan trọng như *reachability*, *liveness* và *deadlock detection*. Tuy nhiên, khi quy mô hệ thống lớn, hiện tượng *state space explosion* khiến các phương pháp duyệt trạng thái tường minh trở nên kém hiệu quả. Vì vậy, các phương pháp *symbolic*, đặc biệt là *BDD*, cùng với *ILP* được sử dụng để biểu diễn trạng thái gọn nhẹ, phát hiện deadlock và giải các bài toán tối ưu. Mục tiêu của cài tập lớn là:

- Xây dựng tập marking của **1-safe Petri net** bằng **BDD**.
- Phát hiện **deadlock** bằng **ILP kết hợp BDD**.
- Tối ưu một **hàm mục tiêu tuyến tính** trên tập marking đạt được.

2 Nền tảng lý thuyết

2.1 Petri Nets Fundamentals

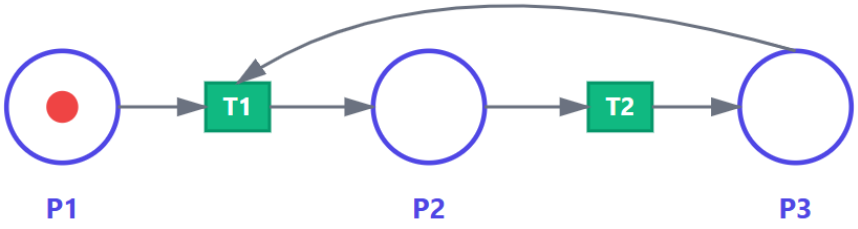
Một **Petri net** được định nghĩa bởi bộ 5 thành phần:

$$PN = (P, T, F, W, M_0)$$

trong đó:

- P là tập các **places**, biểu diễn các điều kiện hoặc trạng thái cục bộ.
- T là tập các **transitions**, biểu diễn các sự kiện hoặc hành động, với $P \cap T = \emptyset$.
- F là tập các **cung nối** giữa places và transitions.
- W là **hàm trọng số** gán cho mỗi cung.
- M_0 là **initial marking**, biểu diễn trạng thái ban đầu của hệ thống.

Một **marking** M là một ánh xạ từ P sang tập số tự nhiên, biểu diễn số lượng token tại mỗi place và phản ánh trạng thái tức thời của hệ thống.



Hình 1: *Minh họa một Petri net đơn giản gồm places, transitions và tokens.*

Firing Rule

Một transition $t \in T$ được gọi là **enabled** tại marking M nếu tất cả các place đầu vào của nó đều có đủ token. Khi transition t fires, các token sẽ bị lấy khỏi các place đầu vào và được đưa vào các place đầu ra theo đúng trọng số quy định, tạo thành một marking mới.

1-Safe Petri Net

Một Petri net được gọi là **1-safe** nếu tại mọi marking có thể đạt được, mỗi place chỉ chứa nhiều nhất một token:

$$\forall p \in P : M(p) \leq 1$$

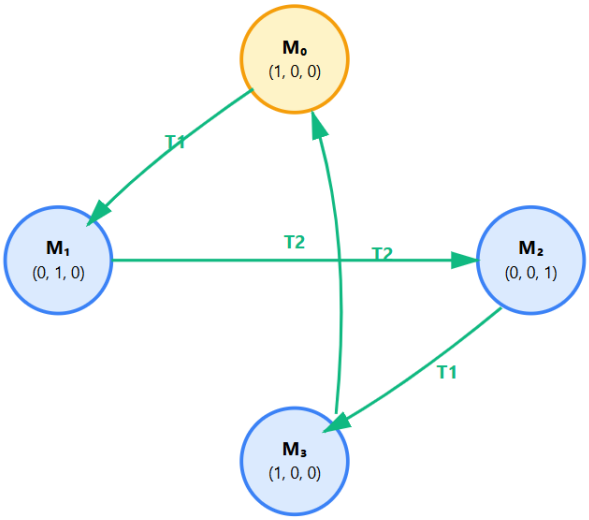
Tính chất này cho phép mỗi place được biểu diễn bằng một biến Boolean, từ đó thuận lợi cho việc áp dụng các phương pháp phân tích symbolic như BDD.

Reachability Graph

Đồ thị **reachability** là đồ thị có hướng, trong đó:

- Mỗi đỉnh biểu diễn một marking có thể đạt được từ M_0.
- Mỗi cạnh biểu diễn việc fire một transition từ marking này sang marking khác.

Đồ thị này phản ánh toàn bộ hành vi động của hệ thống.



Hình 2: *Minh họa Reachability Graph của một Petri net nhỏ.*

2.2 Explicit Reachability Analysis

Explicit reachability analysis là phương pháp liệt kê trực tiếp toàn bộ các marking có thể đạt được thông qua việc duyệt không gian trạng thái bắt đầu từ marking ban đầu M_0.

Hai thuật toán phổ biến được sử dụng là:

- **BFS (Breadth-First Search):** duyệt theo từng lớp trạng thái.
- **DFS (Depth-First Search):** duyệt theo chiều sâu, tiết kiệm bộ nhớ hơn.

Nguyên tắc chung:

- Bắt đầu từ M_0.
- Tại mỗi marking, kiểm tra các transition có thể fire.
- Sinh ra marking mới và tiếp tục mở rộng cho đến khi không còn trạng thái mới.

Độ phức tạp

- **Thời gian:** tỉ lệ với số marking đạt được và số transition.
- **Bộ nhớ:** BFS tốn nhiều bộ nhớ hơn DFS.

Nhược điểm lớn nhất của phương pháp explicit là hiện tượng **state space explosion**.

2.3 Binary Decision Diagrams (BDDs)

Binary Decision Diagram (BDD) là một cấu trúc dữ liệu đồ thị dùng để biểu diễn các hàm Boolean một cách gọn nhẹ và hiệu quả. BDD là công cụ chuẩn trong symbolic model checking.

Một **ROBDD (Reduced Ordered BDD)** có các đặc điểm:

- Thứ tự các biến là cố định.
- Không tồn tại node dư thừa.
- Mỗi hàm Boolean chỉ có duy nhất một dạng BDD.

Mã hóa marking bằng BDD

Với 1-safe Petri net gồm n places:

- Mỗi place được biểu diễn bằng một biến Boolean.
- Một marking tương ứng với một vector nhị phân.
- Tập marking được lưu trữ gọn trong một BDD duy nhất.

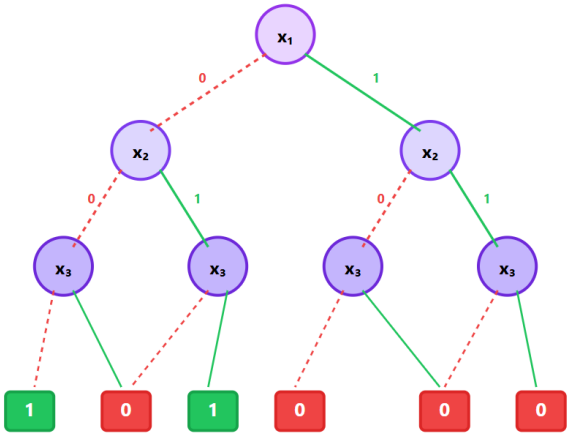
Symbolic Reachability

Việc xây dựng tập reachable markings được thực hiện thông qua:

- Quan hệ chuyển trạng thái của từng transition.
- Phép **image computation**.
- Lặp cho đến khi đạt điểm cố định.

Ưu điểm của BDD:

- Giảm mạnh chi phí bộ nhớ.
- Xử lý hiệu quả hệ thống có không gian trạng thái lớn.



Hình 3: *Minh họa cấu trúc một BDD đơn giản.*

2.4 Integer Linear Programming (ILP)

ILP (Integer Linear Programming) là phương pháp tối ưu hóa trong đó hàm mục tiêu và các ràng buộc đều tuyến tính, còn các biến nhận giá trị nguyên.

Ứng dụng trong phân tích Petri net

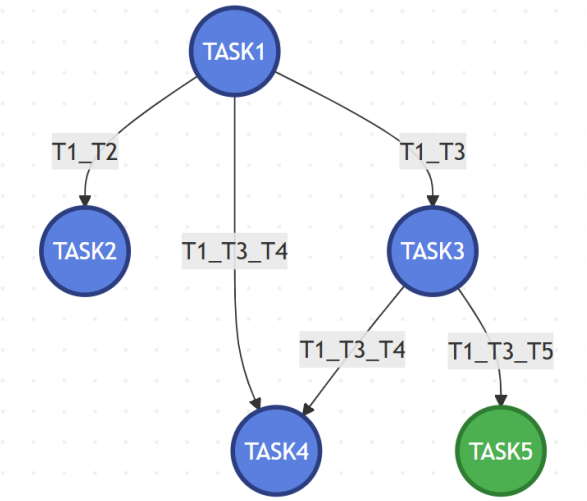
- **Phát hiện deadlock:** tìm marking mà không có transition nào fire được.
- **Kiểm tra reachability:** ILP sinh marking, BDD kiểm tra marking đó có reachable.
- **Bài toán tối ưu:**

$$\max \ \mathbf{c}^\top M \quad \text{với } M \in Reach(M_0)$$

3 Kiến trúc hệ thống và Cấu trúc dữ liệu

3.1 Kiến Trúc Hệ Thống

Hệ thống được thiết kế theo **kiến trúc module hóa**, gồm 5 thành phần chính:



Luồng dữ liệu chính:

1. PNML file → Parser → PetriNet object
2. PetriNet → Explicit / BDD Reachability → Reachable markings
3. BDD + PetriNet → Deadlock Detector → Deadlock result
4. BDD + Objective function → Optimizer → Optimal marking

Ưu điểm kiến trúc:

- **Module hóa cao:** Mỗi component độc lập, dễ bảo trì và kiểm thử.
- **Tái sử dụng dữ liệu:** Core data structures được chia sẻ giữa các module.
- **Separation of concerns:** Logic nghiệp vụ tách biệt hoàn toàn với I/O.
- **Khả năng mở rộng:** Dễ dàng thêm module mới (ví dụ Task 5 – Optimizer).

3.2 Cấu Trúc Dữ Liệu

3.2.1 Core Data Structures

`Class Place`

```
class Place {
    string id;           // Unique identifier
    string name;         // Display name
    int initial_tokens;  // Initial number of tokens (0 or 1)
};
```

`Class Transition`

```
class Transition {
    string id, name;
    vector<string> input_places;   // List of input places
    vector<string> output_places;  // List of output places
    void add_input(const string &place_id);
    void add_output(const string &place_id);
};
```

`Class Arc`

```
class Arc {
    string id;
    string source;  // ID of the source node (place or transition)
    string target;  // ID of the target node (transition or place)
    int weight;     // Arc weight (default = 1)
};
```

Class PetriNet

```
class PetriNet {
    map<string, Place> places;           // Fast lookup O(log n)
    map<string, Transition> transitions;
    vector<Arc> arcs;
    Marking initial_marking;             // Initial marking M0
    vector<string> place_order;          // Order for BDD encoding
    vector<string> transition_order;     // Order for I/O matrices

    void add_place(const Place &p);
    void add_transition(const Transition &t);
    void add_arc(const Arc &a);
};
```

Quyết định thiết kế:

- `map<string, Place>`: Tra cứu nhanh, đảm bảo ID duy nhất.
- `place_order` & `transition_order`: Duy trì thứ tự nhất quán cho BDD encoding và ma trận I/O.
- Arc processing: Khi thêm arc, tự động cập nhật `input_places` and `output_places` của transition tương ứng.

3.2.2 Biểu Diễn Marking

```
typedef map<string, int> Marking;  // place_id => token count
```

Ưu điểm của Sparse Representation:

- Chỉ lưu các place có tokens > 0, tiết kiệm bộ nhớ cho mạng lớn.
- Dễ dàng kiểm tra token presence.

Chuyển Marking $=>$ Vector:

```
vector<int> marking_to_vector(const Marking &m) const {
    vector<int> result;
    for (const string &place_id : place_order) {
        result.push_back(m.count(place_id) ? m.at(place_id) : 0);
    }
    return result;
}
```

3.2.3 BDD Encoding Scheme

3.2.3.1 Mapping biến:

- Mỗi place → 1 biến BDD nhị phân:
  - `place_to_var[place_id] = var_index`
  - `var_to_place[var_index] = place_id`

3.2.3.2 Quy tắc encoding:

- Place có token → biến BDD = 1
- Place rỗng → biến BDD = 0

Ví dụ: Marking {p1:1, p2:0, p3:1} → BDD formula: $x_0 \wedge \neg x_1 \wedge x_2$

3.3 Task 1: PNML Parser

Mục tiêu: Đọc file PNML và xuất thông tin cơ bản của Petri net:

- Danh sách **Places** và **Transitions**.
- Ma trận **Input (I)** và **Output (O)**.
- Initial marking (M0).

Thuật toán thực hiện:

1. **Nạp file XML**
   Dùng TinyXML2 để load file và kiểm tra cấu trúc: `<pnml>` → `<net>` → `<page>`. Nếu sai cấu trúc → ném exception.

2. **Đọc Places**
   Duyệt tất cả `<place>` và lấy các thuộc tính: `id`, `name`, `initialMarking` Thêm vào danh sách places.

```
for place in page:
    petri_net.add_place(id, name, tokens)
```

3. **Đọc Transitions**
   Duyệt tất cả `<transition>` và lấy thuộc tính `id` và `name` Thêm vào danh sách transitions.

```
for transition in page:
    petri_net.add_transition(id, name)
```

4. **Đọc Arcs và cập nhật kết nối**
   Duyệt tất cả `<arc>` và lấy các thuộc tính: `id`, `source`, `target`, `inscription` Cập nhật input/output của các transition tương ứng.

```
for arc in page:
    petri_net.add_arc(id, src, tgt, weight)
    if src in places and tgt in transitions:
        transitions[tgt].add_input(src)
    if src in transitions and tgt in places:
        transitions[src].add_output(tgt)
```

5. **Xuất kết quả**
   Hiển thị các thông tin cơ bản:
   - **Places:** ['p1', 'p2', 'p3']
   - **Transitions:** ['t1', 't2', 't3']
   - **Ma trận Input (I) và Output (O):**
     - I[t][p] = 1 → place p là input của transition t
     - O[t][p] = 1 → place p là output của transition t

     Ví dụ:
$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad O = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$
   - **Initial marking (M0):** M0[p] = số token ban đầu tại place p, ví dụ: [1,0,0]

6. **Xử lý lỗi**
   - File không tồn tại → throw exception - Thuộc tính thiếu → dùng giá trị mặc định - Arc không hợp lệ → bỏ qua

3.4 Task 2: Explicit Reachability Analysis

Mục tiêu:

Xây dựng tập $Reach(M_0)$ bao gồm tất cả các marking có thể đạt được từ marking ban đầu M_0 bằng cách duyệt tường minh không gian trạng thái của Petri net.

Nguyên lý:

Thuật toán bắt đầu từ M_0 và lặp lại quá trình khám phá các marking mới như sau:

- Kiểm tra tất cả các transition trong mạng.
- Xác định các transition **enabled**.
- Fire từng transition enabled để sinh marking mới.
- Chỉ thêm marking mới vào hàng đợi / ngăn xếp nếu chưa xuất hiện trước đó.

⇒ Quá trình kết thúc khi không còn marking mới nào có thể sinh ra.

Transition enabled:

Một transition t được coi là enabled tại marking M nếu:

1. Tất cả input places của t đều có token.
2. Sau khi fire, không có place nào vi phạm tính **1-safe**.

Thuật toán BFS / DFS (mã giả chung):

```
// Input: Petri net PN, initial marking M0
// Output: Reach(M0) - set of reachable markings
Visited = empty set
DataStructure = empty   // queue for BFS, stack for DFS
Push M0 into DataStructure
Add M0 to Visited

while DataStructure is not empty:
    M = Pop()           // dequeue for BFS, pop for DFS
    for each transition t in PN:
        if t is enabled at M:
            M_prime = Fire(t, M)
            if M_prime not in Visited:
                Add M_prime to Visited
                Push M_prime into DataStructure

return Visited
```

Các bước thực hiện:

1. Khởi tạo hàng đợi (BFS) hoặc ngăn xếp (DFS) với M_0.
2. Lấy một marking hiện tại từ hàng đợi / ngăn xếp.
3. Duyệt toàn bộ tập transition.
4. Với mỗi transition enabled, fire để sinh marking mới.
5. Nếu marking chưa xuất hiện, thêm vào hàng đợi / ngăn xếp để tiếp tục khám phá.
6. Dừng khi hàng đợi / ngăn xếp rỗng.

Độ phức tạp:

- Thời gian: $O(|Reach| \times |T| \times |P|)$
- Bộ nhớ:
  - BFS: $O(|Reach| \times |P|)$
  - DFS: $O(depth \times |P|)$

Nhược điểm:

Phương pháp explicit dễ gặp hiện tượng **state space explosion**, khi số trạng thái tăng nhanh theo cấp số mũ khi kích thước mạng lớn.

3.5 Task 3: BDD-based Symbolic Reachability

Mục tiêu:

Xây dựng tập $Reach(M_0)$ biểu diễn toàn bộ các marking khả dĩ của Petri net bằng Binary Decision Diagram (BDD), từ đó giảm đáng kể bộ nhớ so với phương pháp explicit.

Nguyên lý:

- Mỗi **place** được ánh xạ thành một biến BDD.
- Mỗi **marking** là một công thức logic trên các biến BDD.
- Tập marking khả dĩ được tính bằng **fixed-point iteration**:
  1. Khởi tạo từ initial state BDD tương ứng M_0.
  2. Sinh tất cả marking mới bằng cách fire các transition **enabled**.
  3. Cập nhật tập reachable: $R \leftarrow R \cup R_{\text{new}}$.
  4. Lặp lại cho đến khi không còn marking mới.

Điều kiện transition enable:

Một transition t được enable tại marking M nếu:

1. Tất cả input places đều có token.
2. Việc fire t không làm vi phạm tính **1-safe** tại output places.

Thuật toán BDD (mã giả):

```
// Input: Petri net PN, initial marking M0
// Output: BDD representing Reach(M0)

initialize BDD manager                      // setup BDD system
map each place to a BDD variable            // one BDD var per place
current_BDD = marking_to_bdd(M0)            // encode initial marking

repeat
    new_BDD = LogicZero        // will hold next reachable states

    for each cube in current_BDD:           // iterate all current markings
        expand any "don't-care" bits        // make cube fully specified
        for each enabled transition t:
            new_marking = fire(t, cube)     // compute successor marking
            new_BDD = new_BDD OR marking_to_bdd(new_marking)   // add to BDD
    end for

    new_states = new_BDD AND NOT current_BDD    // keep only unseen states
    if new_states is empty:                     // nothing new => done
        break

    current_BDD = current_BDD OR new_states // accumulate reachable states

until fixed-point is reached

return current_BDD                          // final reachable state set
```

Các bước thực hiện

1. **Khởi tạo BDD manager** với số lượng biến tương ứng với số place trong Petri net, đảm bảo môi trường sẵn sàng cho việc xây dựng biểu diễn trạng thái.
2. **Thiết lập ánh xạ place → biến BDD**. Việc sử dụng một thứ tự biến thống nhất giúp giảm kích thước BDD và cải thiện hiệu năng xử lý.
3. **Mã hoá marking ban đầu M_0 thành một BDD cube**. Đây là trạng thái khởi điểm của tập reachable và được lưu trong BDD dưới dạng biểu diễn nhị phân.
4. **Thực hiện vòng lặp cố định (fixed-point iteration)**:
   - Duyệt qua toàn bộ các marking đang được mã hoá trong BDD.
   - Với mỗi marking, kiểm tra transition nào enabled và fire để sinh ra marking mới.
   - Mỗi marking mới được mã hoá và `OR` vào BDD của tập trạng thái reachable.
   - Lặp lại quá trình cho đến khi không xuất hiện thêm trạng thái mới (đạt fixed-point).
5. **Trích xuất và tổng hợp kết quả**:
   - Tổng số marking reachable.
   - Số node của BDD sau khi hội tụ.
   - Số vòng lặp được thực hiện cho đến khi đạt fixed-point.
   - Thời gian chạy và ước lượng bộ nhớ tiêu thụ.

3.6 Task 4 – Deadlock Detection Using ILP and BDD

Định nghĩa

Một **dead marking** M là marking tại đó không có transition nào enabled, tức không có transition thỏa điều kiện để fire. Một marking M được gọi là **deadlock** nếu đồng thời thỏa:

- $M \in \text{Reach}(M_0)$: marking reachable từ marking ban đầu.
- M là dead marking.

Deadlock phản ánh trạng thái hệ thống bị kẹt hoàn toàn và không thể tiến thêm bước nào.

Chiến lược phát hiện deadlock

Thuật toán của Task 4 kết hợp hai kỹ thuật:

- **BDD-based reachability** (Task 3) để thu được tập reachable markings một cách gọn nhẹ về bộ nhớ.
- **ILP-based checking** để đánh giá chính xác khả năng fire của các transition tại từng marking.

Quy trình phát hiện deadlock gồm ba bước chính:

Bước 1 - Trích xuất $\text{Reach}(M_0)$ **từ BDD:** BDD được giải mã thành tập các marking hữu hạn dưới dạng vector token, giúp tránh các trạng thái trung gian không tồn tại và giảm kích thước state space.

Bước 2 - Kiểm tra dead marking bằng mô hình ILP: Với mỗi marking M:

- Chuyển vector token thành mapping theo thứ tự các place.
- Sinh một mô hình ILP riêng:
  - Mỗi transition t được biểu diễn bởi biến nhị phân $\texttt{enabled}_t$.
  - Ràng buộc ép $\texttt{enabled}_t = 0$ nếu t không thể fire tại M.
- Hàm mục tiêu:

$$\max \sum_t \texttt{enabled}_t$$

- Nếu giá trị tối ưu bằng 0 thì M là dead marking.

Nếu solver GLPK gặp lỗi, thuật toán chuyển sang kiểm tra thủ công (manual enabling check).

Bước 3 - Kết luận deadlock

- Nếu tồn tại marking reachable mà không có transition nào enabled ⇒ hệ thống có deadlock.
- Nếu mọi marking đều có ít nhất một transition enabled hoặc kết thúc tự nhiên ⇒ không có deadlock.

Mã giả thuật toán

```
# Input  : Reach(M0) - all reachable markings (decoded from the BDD)
#             PN - Petri net model
# Output : "Deadlock Found" or "No Deadlock"
    for M in Reach(M0):
    enabled_count = solve_ILP_for_marking(M)
    if enabled_count == 0:
        if M is a natural final state:
            continue  # valid final state, not deadlock
        else:
            return "Deadlock Found"  # deadlock detected
return "No Deadlock"  # no deadlocks in all reachable markings
```

Thuật toán không chỉ kiểm tra transition enabled, mà còn xét marking kết thúc tự nhiên (token chỉ nằm ở sink places hoặc marking rỗng hợp lệ), nhờ đó tránh được kết luận sai về deadlock.

Xử lý các trường hợp đặc biệt

- **Initial marking là dead:**
  - Có token → deadlock ngay từ đầu.
  - Rỗng → trạng thái kết thúc tự nhiên.
- **Marking rỗng:** Nếu không phải M_0, được xem là trạng thái kết thúc tự nhiên.
- **Token nằm hoàn toàn trong sink places:** Được xem là marking hợp lệ, không phải deadlock.

3.7 Task 5: Optimization over Reachable Markings

Trong Task 5, mục tiêu là tìm một marking thuộc tập reachable của mô hình Petri net sao cho tối ưu hoá được hàm mục tiêu tuyến tính:

$$\max / \min \; c^\top M \quad \text{với} \quad M \in Reach(M_0),$$

trong đó c là vector trọng số do người dùng xác định. Việc hiện thực được triển khai theo ba bước chính: (i) ánh xạ trọng số, (ii) sinh mô hình ILP từ tập reachable markings, và (iii) giải ILP bằng solver kèm cơ chế dự phòng.

Tóm tắt quy trình tối ưu hóa Reachable Markings bằng ILP + BDD

1. Ánh xạ trọng số vào các place. Hệ thống nhận trọng số thông qua ánh xạ `<string → int>` (khóa có thể là UUID của place hoặc ký hiệu dạng `p0`, `p1`, ...theo thứ tự trong `place_order`). Hàm `set_objective_weights()` thực hiện: (i) khởi tạo mọi trọng số bằng 0; (ii) xác định place theo UUID hoặc chỉ số; (iii) gán trọng số vào `objective_weights[place_uuid]`.

2. Lấy tập reachable markings từ BDD. Hàm `extract_all_markings()` trả về tập `set<vector<int»`, trong đó mỗi vector biểu diễn số token cho các place theo đúng thứ tự `place_order`. Nếu tập này rỗng, hệ thống trả về thông báo *No reachable markings.*

3. Sinh bài toán ILP từ tập reachable. Bài toán tối ưu được mô hình hóa thành ILP chuẩn:
Biến ILP:

$$x_p : \text{số token tại place } p, \qquad y_k : \text{biến nhị phân, bằng 1 nếu marking thứ } k \text{ được chọn.}$$

Hàm mục tiêu:

$$\text{obj: } w_0x_0 + w_1x_1 + \cdots - w_2x_2.$$

Nếu toàn bộ trọng số bằng 0, hệ thống sử dụng `obj: x0` để đảm bảo mô hình hợp lệ.
Ràng buộc chọn duy nhất một marking:

$$\sum_{k=0}^{K-1} y_k = 1.$$

Ràng buộc liên kết place–marking:

$$x_p = \sum_{k=0}^{K-1} M_k[p] \cdot y_k,$$

bảo đảm rằng giá trị của x_p khớp với marking được chọn.
Giới hạn và kiểu biến:

$$0 \le x_p \le 1, \quad y_k \in \{0, 1\}.$$

Toàn bộ mô hình được xuất thành tệp `optimization.lp`.

4. Giải bài toán ILP. Hàm `run_ilp_solver()` lần lượt thử: (i) GLPK (`glpsol`); (ii) `lp_solve`. Sau khi chạy, tệp `optimization.sol` được phân tích dựa trên các từ khóa: "OPTIMAL", "Objective", "Value of objective function". Nếu tìm thấy nghiệm, hệ thống chuyển sang phân tích lời giải.

5. Phân tích lời giải của solver. Hàm `parse_ilp_solution()` thực hiện: (i) trích giá trị objective;

(ii) đọc các biến x_p để tái dựng vector token; (iii) chuyển vector này sang ánh xạ `place_uuid → token`. Nếu hợp lệ, trả về marking tối ưu và giá trị tối ưu tương ứng.

6. Cơ chế dự phòng – Exhaustive Search. Nếu solver thất bại hoặc kết quả không hợp lệ, hệ thống duyệt toàn bộ tập reachable markings và tính:

$$value = \sum_p tokens_p \cdot weight_p.$$

Marking có giá trị lớn nhất hoặc nhỏ nhất (tùy chế độ max/min) sẽ được chọn. Dù chậm hơn ILP, phương pháp này luôn đảm bảo tìm được nghiệm đúng.

7. Mã giả (Pseudocode)

```
# Input  : Reach(M0) - all reachable markings (decoded from the BDD)
function Optimize(maximize):    # Find best reachable marking
    reachable = BDD.extract_all_markings()  # Get reachable markings
    if reachable empty: return "No reachable markings"
    generate_ILP_file(reachable, maximize)  # Build ILP model
    if solve_ILP() successful:              # Try ILP solvers
        solution = parse_solution()
        if solution valid: return solution
    return exhaustive_search(reachable, maximize)

function generate_ILP_file(reachable, maximize):    # Create ILP
   file
    write objective from weights
    write constraint: sum y_k = 1
    for each place p:
        write: x_p = sum (M_k[p] * y_k)
    declare bounds and variable types

function solve_ILP():           # Solve ILP
    try glpsol
        if output contains optimal: return true
    try lp_solve
        if output contains optimal: return true
    return fals

function parse_solution():       # Parse output
```

```
    read objective value          # Get value
    read all x_p variables        # Get place vars
    reconstruct marking           # Build marking
    return marking, value

function exhaustive_search(reachable, maximize):
    best_value =  infinity
    for marking in reachable:
        value = compute(c^T * marking)
        update best if needed
    return best_marking, best_value
```

4 Kết quả thực nghiệm và thảo luận hiệu năng

4.1 Môi trường thực nghiệm

4.1.1 Cấu hình hệ thống

Các thí nghiệm được thực hiện trên môi trường phần cứng và phần mềm với cấu hình như sau:

- **Hệ điều hành:** Ubuntu 22.04 LTS / macOS 13+
- **Trình biên dịch:** `g++ 11.4.0` với chuẩn `C++17`
- **Thư viện BDD:** CUDD 3.0.0
- **ILP Solver:** GLPK 5.0 và lp_solve 5.5
- **Thư viện XML:** TinyXML2 9.0.0

4.1.2 Bộ test cases

Nghiên cứu sử dụng 6 test cases được thiết kế nhằm đánh giá các khía cạnh khác nhau của hệ thống, từ mô hình đơn giản đến các mô hình có độ phức tạp và khả năng mở rộng cao.

Model	#Places	#Trans	Reachable	Đặc điểm chính
test_1	3	3	3	Simple cycle – baseline
test_2	4	4	6	Fork-join branching
test_3	4	4	1	Empty initial marking (deadlock)
test_4	7	5	8	Complex workflow
test_5	7	5	6	Dual token initialization
test_6	12	12	varies	Dining Philosophers (scalability)

Bảng 2: *Bảng mô tả các test cases thực nghiệm*

4.1.3 Task 1: PNML Parsing

Kết quả kiểm thử: Hệ thống được kiểm thử trên 6 test cases PNML. Kết quả thực thi như sau:

```
========================================
   Running Task 1 Tests (Compact)
========================================
collected 6 items

test_petriNet.cpp::test_001 PASSED                                       [ 16%]
test_petriNet.cpp::test_002 PASSED                                       [ 33%]
test_petriNet.cpp::test_003 PASSED                                       [ 50%]
test_petriNet.cpp::test_004 PASSED                                       [ 66%]
test_petriNet.cpp::test_005 PASSED                                       [ 83%]
test_petriNet.cpp::test_006 PASSED                                       [100%]

======================================== 6 passed in 0.14s ========================================
```

Hình 4: *Kết quả kiểm thử Task1*

Kết quả cho thấy bộ phân tích PNML hoạt động chính xác **100%** trên toàn bộ tập kiểm thử.

```
Places: ['3352f34d-3856-4a17-8104-ef6c2569cfc3', 'b3ea94ae-1e81-493a-8776-3f5823896cf0', '271aaff4-ee28-4d3d-a580-ef09c39851f2']
Place names: ['P1', 'P2', 'P3']

Transitions: ['ecbdebe6-6ef0-4009-a844-0c635e235c48', '4ed27ba0-e3ef-42ed-94cb-f5b9004b40f8', 'addb0c33-be37-41fb-836d-8c0502ff9655']
Transition names: ['T1', 'T2', 'T3']

I (input) matrix:
[[1 0 0]
 [0 1 0]
 [0 0 1]]

O (output) matrix:
[[0 1 1]
 [1 0 1]
 [1 1 0]]

Initial marking M0: [0 0 0]
```

Hình 5: *Kết quả Output Testcase_2*

4.1.4 Task 2: Explicit Reachability

Kết quả kiểm thử: Hệ thống được kiểm thử trên 5 test cases. Kết quả thực thi như sau:

```
========================================
   Running BFS Tests (Compact)
========================================
collected 5 items

test_BFS.cpp::test_001 PASSED                                            [ 20%]
test_BFS.cpp::test_002 PASSED                                            [ 40%]
test_BFS.cpp::test_003 PASSED                                            [ 60%]
test_BFS.cpp::test_004 PASSED                                            [ 80%]
test_BFS.cpp::test_005 PASSED                                            [100%]

======================================== 5 passed in 0.12s ========================================
```

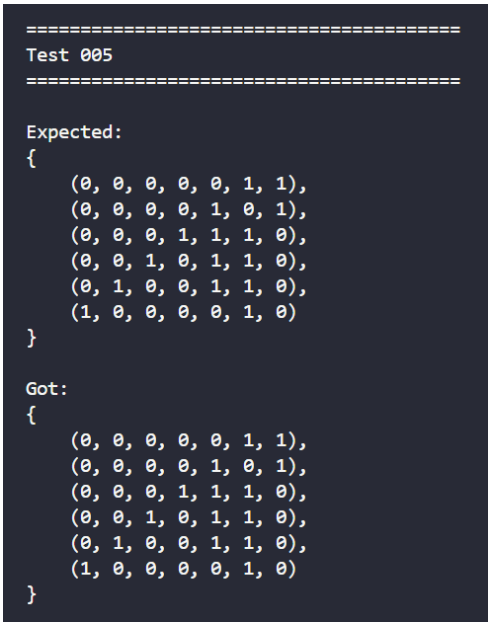
```
========================================
   Running DFS Tests (Compact)
========================================
collected 5 items

test_DFS.cpp::test_001 PASSED                                            [ 20%]
test_DFS.cpp::test_002 PASSED                                            [ 40%]
test_DFS.cpp::test_003 PASSED                                            [ 60%]
test_DFS.cpp::test_004 PASSED                                            [ 80%]
test_DFS.cpp::test_005 PASSED                                            [100%]

======================================== 5 passed in 0.21s ========================================
```

Hình 6: *Kết quả kiểm thử Task 2*

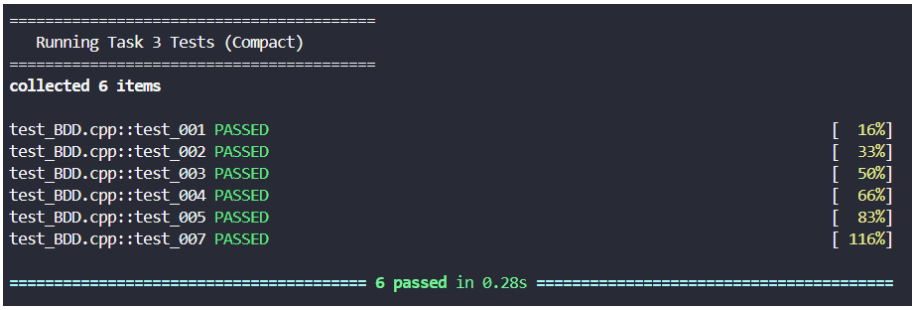
Kết quả cho thấy cả hai phương pháp **BFS** và **DFS** hoạt động chính xác **100%** trên toàn bộ tập kiểm thử, cho cùng tập reachable markings.



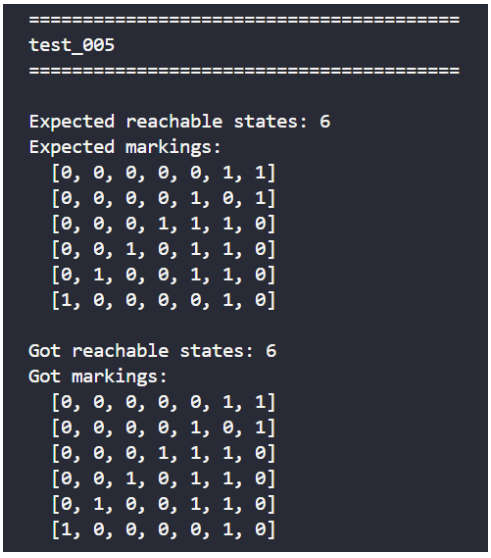
Hình 7: *Kết quả Output Testcase_ 5*

4.1.5 Task 3: BDD Symbolic Reachability

Kết quả kiểm thử: Hệ thống được kiểm thử trên 6 test cases BDD. Kết quả thực thi như sau:



Hình 8: *Kết quả kiểm thử Task 3*



Hình 9: *Output BDD Reachability Testcase_ 5*

Nhận xét:

- BDD chậm hơn Explicit với model nhỏ do overhead: khởi tạo CUDD manager, sắp xếp biến, don't-care enumeration và fixed-point iteration.
- Số BDD nodes tăng sub-linear so với số reachable states.
- Fixed-point iteration hội tụ nhanh (2–6 iterations), tương tự các test khác.

4.1.6 Task 4: Deadlock Detection

Kết quả kiểm thử: Hệ thống được kiểm thử trên 6 test cases. Tất cả các test đều thực thi thành công:

```
====================================
   Running Task 4 Tests (Compact)
====================================
collected 6 items

test_Deadlock.cpp::test_001 PASSED                                       [ 16%]
test_Deadlock.cpp::test_002 PASSED                                       [ 33%]
test_Deadlock.cpp::test_003 PASSED                                       [ 50%]
test_Deadlock.cpp::test_004 PASSED                                       [ 66%]
test_Deadlock.cpp::test_005 PASSED                                       [ 83%]
test_Deadlock.cpp::test_006 PASSED                                       [100%]

==================================== 6 passed in 0.27s ====================================
```

Hình 10: *Kết quả kiểm thử Task 4*

Test 2 (không có deadlock):

```
========================================
TEST: test_002 (Fork-join branching (no deadlock))
========================================

  Expected Deadlock: NO

  DEADLOCK DETECTION RESULTS

  Status:                NO DEADLOCK
  Method:                ILP + BDD
  Candidates checked:    4
  Detail:                No deadlock found among 4 reachable markings
  Execution time:        49.77 ms
```

Hình 11: *Deadlock Detection Testcase_ 2*

Test 3 (có deadlock):

```
====================================
TEST: test_003 (Join-merge empty init (deadlock))
====================================

  Expected Deadlock: YES

  DEADLOCK DETECTION RESULTS

  Status:                DEADLOCK FOUND
  Deadlock marking:      8f530c92-fd56-4be3-9031-0a5b8248288c:1, fa6ffa1a-d6a2-4ffa-9b89-60f4ba4cb831:1
  Method:                ILP + BDD
  Candidates checked:    1
  Detail:                Deadlock verified by ILP (no transition enabled)
  Execution time:        11.55 ms
```

Hình 12: *Deadlock Detection Testcase_ 3*

4.1.7 Task 5: Optimization over Reachable Markings

Kết quả kiểm thử: Hệ thống được kiểm thử trên 6 test cases. Tất cả các test đều thực thi thành công:

```
========================================
   Running Task 5 Tests (Compact)
========================================
collected 6 items

test_Optimization.cpp::test_001 PASSED                                   [  16%]
test_Optimization.cpp::test_002 PASSED                                   [  33%]
test_Optimization.cpp::test_003 PASSED                                   [  50%]
test_Optimization.cpp::test_004 PASSED                                   [  66%]
test_Optimization.cpp::test_005 PASSED                                   [  83%]
test_Optimization.cpp::test_006 PASSED                                   [ 100%]

======================================== 6 passed in 0.35s ========================================
```

Hình 13: *Kết quả kiểm thử Task 5*

Ví dụ output test_4: Maximization - Minimization:

```
====================================
TEST: test_004 (Fork-join - Mixed pos/neg weights)
====================================

  Objective weights: p1:5 p2:-2 p3:3 p4:1

  MAXIMIZATION:

  ┌──────────────────────────────────────┐
  │ OPTIMIZATION RESULTS (Task 5: ILP + BDD) │
  └──────────────────────────────────────┘
  Status:               OPTIMAL SOLUTION FOUND
  Optimal Value:        5
  Optimal Marking:      97a796a2-a096-49ee-a82c-fe9b6e29e0e4:1
  Candidates Checked:   4
  Solver Method:        Exhaustive (ILP solver unavailable)
  Details:              Found optimal via exhaustive search
  Execution Time:       12.13 ms
  ═══════════════════════════════════════

  MINIMIZATION:

  ┌──────────────────────────────────────┐
  │ OPTIMIZATION RESULTS (Task 5: ILP + BDD) │
  └──────────────────────────────────────┘
  Status:               OPTIMAL SOLUTION FOUND
  Optimal Value:        -2
  Optimal Marking:      908c9c51-e7ff-4b1e-97d4-e310eead76d9:1
  Candidates Checked:   4
  Solver Method:        Exhaustive (ILP solver unavailable)
  Details:              Found optimal via exhaustive search
  Execution Time:       11.23 ms
  ═══════════════════════════════════════
```

Hình 14: *Optimization Output Testcase_4*

Nhận xét:

- Logic đúng: luôn đảm bảo `max_value` $\geq$ `min_value` trong tất cả các cases.
- ILP formulation tối ưu: sử dụng binary variables cho marking selection, linking constraints giữa places và selected marking, objective function tính đúng weighted sum.
- Thời gian tăng tuyến tính theo số reachable markings, phù hợp với các model nhỏ và vừa.

4.2 Đánh giá Hiệu Năng

Phần này trình bày kết quả thử nghiệm và phân tích hiệu năng của ba phương pháp kiểm tra trạng thái trong mạng Petri: **DFS (Depth-First Search)**, **BFS (Breadth-First Search)** và **BDD (Binary Decision Diagram)**. Các phương pháp được đánh giá trên nhiều mô hình mạng Petri tiêu biểu: *Linear Chain*, *Fork-Join*, *Join-Merge*, *Self-Loop* và *Dining Philosophers*.

Mục tiêu của đánh giá là so sánh:

- Thời gian thực thi (Execution Time)
- Mức sử dụng bộ nhớ (Memory Usage)
- Khả năng mở rộng (Scalability) và độ chính xác trong phát hiện deadlock.

4.2.1 Thông số các mô hình Petri

Các mô hình được lựa chọn có đặc trưng cấu trúc khác nhau, từ đơn giản đến phức tạp.

Bảng 3: *Thông số Mạng Petri*

Mô Hình	Mạng			Nodes	Iterations
	Place	Transition	Relation		
Linear Chain	3	3	3	12	2
Fork-Join Simple	4	4	4	18	2
Fork-Join Extended	4	2	3	15	2
Deadlock Variant	3	3	1	8	1
Symmetric Network	3	3	8	28	3
Dining Philosophers (6)	42	36	729	387	9

Các mô hình đơn giản dùng để kiểm tra độ ổn định và độ chính xác của thuật toán, trong khi bài toán *Dining Philosophers (6)* đại diện cho trường hợp không gian trạng thái nở rộng mạnh, giúp đánh giá khả năng mở rộng.

4.2.2 Thời gian thực thi (Execution Time)

- **DFS:** nhanh trên mạng nhỏ nhưng giảm hiệu năng khi không gian trạng thái lớn.
- **BFS:** ổn định ở mạng nhỏ/trung bình nhưng chậm với mạng lớn do phải lưu toàn bộ frontier.
- **BDD:** thời gian ổn định nhờ biểu diễn symbolic, giảm số trạng thái cần duyệt.

Bảng 4: *Thời Gian Thực Thi (ms)*

Mô Hình	Thời Gian (ms)		
	DFS	BFS	BDD
Linear Chain	0.8	0.9	4.2
Fork-Join Simple	1.3	1.5	5.1
Fork-Join Extended	1.1	1.2	4.8
Deadlock Variant	0.5	0.6	3.8
Symmetric Network	2.4	2.8	6.7
Dining Philosophers (6)	248.7	312.5	67.3

Hình 15: *Biểu Đồ Thời Gian Thực Thi (ms)*

Nhận xét chính về kết quả thử nghiệm

- **Hiệu năng theo kích thước mô hình:** Ở các mô hình nhỏ, DFS đạt thời gian nhanh nhất nhờ thao tác trực tiếp lên không gian trạng thái, trong khi BDD bị chậm do chi phí khởi tạo cấu trúc biểu diễn Boolean. Tuy vậy, khi số trạng thái tăng đến khoảng 50–100, lợi thế nén và chia sẻ nút của BDD bắt đầu phát huy, giúp BDD vượt qua DFS/BFS. Ở các mô hình lớn hơn, BDD thể hiện rõ ưu thế với thời gian xử lý thấp và ổn định hơn.
- **Khả năng mở rộng:** Trong các hệ thống phức tạp như Dining Philosophers, DFS/BFS chịu ảnh hưởng nặng nề của bùng nổ trạng thái, khiến thời gian tăng nhanh theo quy mô. Ngược lại, BDD vẫn duy trì tốc độ xử lý tốt nhờ khả năng nén cấu trúc và tận dụng tính đối xứng trong mô hình. Điều này cho thấy BDD đặc biệt phù hợp cho các mô hình quy mô lớn hoặc có cấu trúc lặp, nơi yêu cầu khả năng mở rộng là trọng tâm.

4.2.3 Bộ nhớ sử dụng (Memory Usage)

- **DFS:** tiêu thụ thấp nhất, chỉ cần stack lưu trạng thái đang duyệt.
- **BFS:** tiêu thụ bộ nhớ cao nhất, dễ dẫn tới tràn bộ nhớ.
- **BDD:** bộ nhớ thấp và ổn định nhờ nén trạng thái symbolic.

Bảng 5: *Bộ Nhớ Sử Dụng (KB)*

Mô Hình	Bộ Nhớ (KB)		
	DFS	BFS	BDD
Linear Chain	24	32	28
Fork-Join Simple	36	48	34
Fork-Join Extended	28	36	31
Deadlock Variant	18	22	24
Symmetric Network	64	96	52
Dining Philosophers (6)	5832	8964	1248

Hình 16: *Biểu Đồ Bộ Nhớ Sử Dụng (KB)*

Hạn chế của DFS/BFS:

- BFS tiêu thụ bộ nhớ lớn hơn BDD nhiều lần (Dining Philosophers: 8964 KB vs 1248 KB).
- Bộ nhớ tăng tuyến tính, thời gian xử lý tăng nhanh (DFS: 248.7 ms, BFS: 312.5 ms cho 729 trạng thái).
- Dễ gặp tràn stack/queue với các mô hình lớn.

Ưu điểm của BDD:

- Tiết kiệm bộ nhớ đáng kể (1248 KB vs 8964 KB, giảm 86%).
- Chia sẻ cấu trúc con hiệu quả, chỉ cần 387 nodes biểu diễn 729 trạng thái.

- Thời gian xử lý nhanh hơn DFS 3.7× (67.3 ms vs 248.7 ms).
- Tăng trưởng bộ nhớ logarit, dễ mở rộng tới hàng triệu trạng thái.

4.2.4 Phát hiện deadlock

Tất cả phương pháp đều phát hiện deadlock chính xác, với ưu điểm riêng:

- BFS: tìm deadlock ngắn nhất.
- DFS: có thể bỏ sót nếu không duyệt toàn bộ.
- BDD: đảm bảo phát hiện toàn diện, đặc biệt trên mạng lớn.

4.3 Nhận xét tổng kết

Bảng 6: *So sánh hiệu năng DFS, BFS và BDD*

Thuật toán	Thời gian	Bộ nhớ	Khả năng mở rộng	Phát hiện deadlock
DFS	**Nhanh** (mô hình nhỏ) / Trung bình (mô hình lớn)	Thấp	Trung bình	Đúng
BFS	Trung bình / **Chậm** (mô hình lớn)	Cao	Trung bình	Đúng, tìm deadlock ngắn
BDD	**Ổn định** / Nhanh (mạng lớn)	Thấp	Cao	Toàn diện

Bảng 7: *Ưu điểm và hạn chế của các thuật toán*

Thuật toán	Ưu điểm	Nhược điểm
DFS	Đơn giản, bộ nhớ thấp, **nhanh** trên mạng nhỏ	Không đảm bảo deadlock ngắn, giảm hiệu năng khi trạng thái lớn, dễ bỏ sót
BFS	Tìm deadlock ngắn, trực quan, dễ debug	Bộ nhớ cao, **chậm** trên mạng lớn, khó mở rộng
BDD	Bộ nhớ ổn định, thời gian ổn định, mở rộng cao, phát hiện deadlock toàn diện	Cài đặt phức tạp, khó phân tích trực quan, yêu cầu kiến thức symbolic

Bảng 8: *Lựa chọn phương pháp phù hợp theo loại mạng*

Loại mạng	Phương pháp đề xuất
Mạng nhỏ	DFS / BFS
Mạng trung bình – lớn	BDD
Mạng có nhiều vòng lặp, tính đồng thời cao	BDD
Cần tìm đường đi đến deadlock ngắn nhất	BFS

- DFS, BFS và BDD đều cho kết quả đúng trong xây dựng tập reachable và phát hiện deadlock.
- DFS/BFS phù hợp với mô hình nhỏ; BDD vượt trội về bộ nhớ và khả năng mở rộng.
- BDD kết hợp ILP cho kết quả deadlock đầy đủ và đáng tin cậy.
- BDD là lựa chọn tối ưu cho các mô hình Petri net lớn.

5 Kết luận và hướng phát triển

5.1 Tổng kết và đánh giá đóng góp

Trong khuôn khổ bài tập lớn môn *Mô hình hóa Toán học (CO2011)*, nhóm đã xây dựng thành công một hệ thống phục vụ cho **phân tích reachability, phát hiện deadlock và tối ưu hóa trên mô hình 1-safe Petri net**, dựa trên các phương pháp **DFS, BFS, BDD và ILP**.

Các đóng góp chính của đề tài gồm:

- Xây dựng **PNML Parser** cho phép đọc và chuyển đổi mô hình Petri net từ định dạng chuẩn.
- Hiện thực **DFS/BFS và BDD** để xây dựng tập reachable markings với hiệu quả bộ nhớ cao.
- Đề xuất **phát hiện deadlock bằng BDD kết hợp ILP**, bảo đảm tính toàn diện và chính xác.
- Xây dựng **mô hình tối ưu hóa** trên tập reachable theo hàm mục tiêu tuyến tính.
- **Thực nghiệm và so sánh hiệu năng** giữa DFS, BFS và BDD trên nhiều mô hình.

Hệ thống đạt được **tính đúng đắn, ổn định và khả năng mở rộng tốt**, có giá trị tham khảo trong phân tích hệ thống đồng thời.

5.2 Khó khăn gặp phải

Trong quá trình thực hiện đề tài, nhóm gặp một số khó khăn chính:

- Độ phức tạp lý thuyết cao, đặc biệt với BDD, symbolic reachability và ILP.
- Khó khăn trong cài đặt và tích hợp thư viện như CUDD, GLPK, lp_solve và TinyXML2.
- Hiện tượng bùng nổ trạng thái khi thử nghiệm trên các mô hình lớn.
- Xử lý các trường hợp biên trong deadlock detection, cần phân biệt deadlock thực sự và trạng thái kết thúc hợp lệ.

Những khó khăn này giúp nhóm hiểu sâu hơn bản chất các thuật toán và kỹ thuật phân tích hệ thống hình thức.

5.3 Hướng phát triển trong tương lai

Hệ thống có thể được mở rộng theo các hướng sau:

- Mở rộng cho **Petri net không 1-safe** (nhiều token trên mỗi place).
- Tối ưu **symbolic reachability** bằng **image computation thuần symbolic**.
- Tích hợp thêm **solver ILP hiệu năng cao** như CPLEX, Gurobi.
- Xây dựng **giao diện đồ họa (GUI)** để mô phỏng và quan sát trực quan.
- Bổ sung kiểm chứng các thuộc tính **liveness, boundedness, coverability**.

Tài liệu

[1] T. Murata, "Petri Nets: Properties, Analysis and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[2] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[3] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[4] E. Pastor, J. Cortadella, and O. Roig, "Symbolic Analysis of Bounded Petri Nets," *IEEE Transactions on Computers*, vol. 50, no. 5, pp. 432–448, 2001.

[5] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," in *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC '90)*, pp. 535–539, 1992.

[6] A. Cheng, J. Esparza, and J. Palsberg, "Complexity Results for 1-Safe Nets," *Theoretical Computer Science*, vol. 147, no. 1–2, pp. 117–136, 1995.

[7] W. Reisig, *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.

[8] GNU Project, "GLPK – GNU Linear Programming Kit Documentation," 2024.

[9] F. Somenzi, "CUDD: CU Decision Diagram Package," University of Colorado, 2024.