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

NHẬP MÔN TRÍ TUỆ NHÂN TẠO (CO3061)

BÁO CÁO BÀI TẬP LỚN 1

BÀI TẬP LỚN 1: SEARCH

Giảng viên hướng dẫn: Vương Bá Thịnh

Sinh viên thực hiện: Nguyễn Trọng Vinh - 2015070
Đào Nguyễn Đức Duy - 2012811
Nguyễn Nhựt Nguyên - 2011706

Thành phố Hồ Chí Minh, Tháng 4 Năm 2024

Mục lục

1 MÔ TẢ CÔNG VIỆC

MEMBER	STUDENT ID	DESCRIPTION WORKS	EFFORT
Đào Nguyễn Đức Duy	2012811	Depth-first Search Pipes Puzzle	100%
Nguyễn Trọng Vinh	2015070	Giải thuật A* với trò chơi Pipes puzzle	%
Nguyễn Nhựt Nguyên	2011706	Sudoku, Breadth-first Search & Simulated Annealing	100%

2 CƠ SỞ LÝ THUYẾT CÁC GIẢI THUẬT SEARCH ÁP DỤNG ĐỂ GIẢI TRÒ CHƠI

2.1 Giải thuật Depth-first Search

- Tìm kiếm theo chiều sâu (DFS: Depth First Search hay Depth First Traversal) là một trong những bài toán phổ biến trong lĩnh vực của khoa học máy tính, đặc biệt là trong lĩnh vực đồ thị và tìm kiếm. Thuật toán này được sử dụng để khám phá và tìm kiếm thông tin trong các cấu trúc dữ liệu như đồ thị, cây, hoặc bất kỳ cấu trúc nào có thể được biểu diễn dưới dạng một tập hợp các nút và các liên kết giữa chúng. Thuật được sử dụng để duyệt qua tất cả các đỉnh của đồ thị bằng cách đi sâu vào một nhánh của đồ thị càng xa càng tốt. Điểm khởi đầu của thuật toán DFS là một đỉnh bất kỳ trong đồ thị, sau đó thuật toán di chuyển tới một đỉnh liền kề chưa được ghé thăm và lặp lại quá trình này cho đến khi không còn đỉnh nào để ghé thăm nữa.
- DFS là một thuật toán đơn giản và dễ hiểu, dễ thực hiện.DFS được ứng dụng trong tìm đường đi từ một nút đến một nút khác trong một đồ thị, mặc dù DFS không đảm bảo tìm ra đường đi ngắn nhất, DFS có thể được dùng để phát hiện và xác định các thành phần liên thông trong một đồ thị. Các thành phần liên thông là nhóm các đỉnh trong đồ thị mà mỗi đỉnh đều có thể truy cập được từ bất kỳ đỉnh nào trong nhóm đó, lập chỉ mục và duyệt web: DFS có thể dùng để duyệt qua các trang web và lập chỉ mục nội dung của chúng. DFS thường được sử dụng để tìm kiếm các liên kết các trang web liên quan.

2.2 Giải thuật tìm kiếm A*

- A* là giải thuật tìm kiếm trong đồ thị, cho phép người dùng tìm đường đi từ một đỉnh hiện tại đến đỉnh đích bằng việc sử dụng một hàm ước lượng khoảng cách hay còn gọi là **hàm Heuristic**.
- Từ trạng thái hiện tại A* xây dựng tất cả các đường đi có thể đi dùng hàm ước lược khoảng cách (hàm Heuristic) để đánh giá đường đi tốt nhất có thể đi, và thực tế A* luôn tìm được đường đi ngắn nhất nếu tồn tại đường đi như thế.
- A* lưu giữ một tập các đường đi qua đồ thị, từ đỉnh bắt đầu đến đỉnh kết thúc, tập các đỉnh có thể đi tiếp được lưu trong tập **OpenList(tương ứng với mã nguồn)**.
- Thứ tự ưu tiên cho một đường đi được quyết định bởi hàm Heuristic được đánh giá bởi hàm: **f(x) = g(x) + h(x)**, trong đó: **g(x)** là hàm xác định khoảng cách từ đỉnh xuất phát đến đỉnh hiện tại, và **h(x)** là hàm xác định khoảng cách từ đỉnh hiện tại đến đỉnh đích. Tại đây, **khi f(x) càng thấp thì đỉnh hiện tại sẽ có độ ưu tiên càng cao.**
- **Ưu điểm:** Đây là một thuật giải linh động, tổng quát, trong đó hàm chứa cả tìm kiếm chiều sâu, tìm kiếm chiều rộng và những nguyên lý Heuristic khác. Nhanh chóng tìm đến lời giải với sự định hướng của hàm Heuristic. Chính vì thế mà người ta thường nói A* chính là thuật giải tiêu biểu cho Heuristic.
- **Nhược điểm:** A* rất linh động nhưng vẫn gặp một khuyết điểm cơ bản - giống như chiến lược tìm kiếm chiều rộng - đó là tốn khá nhiều bộ nhớ để lưu lại những trạng thái đã đi qua.

2.3 Breadth-first Search (Tìm kiếm theo chiều rộng):

- Thuật toán Breadth-first Search (tìm kiếm theo chiều rộng) là một chiến lược tìm kiếm mù (blind search) - tức là chúng ta hoàn toàn không có thông tin để định hướng tìm kiếm.
- Thuật toán BFS sẽ duyệt trên một cây hoặc một đồ thị theo chiều ngang, với điểm khởi đầu tại gốc (trạng thái ban đầu) của cây hoặc đồ thị. Tức là thuật toán sẽ khám phá các đỉnh ở cùng mức (cùng tầng) trên cây trạng thái trước, sau đó mới đi sâu đến các đỉnh kề với chúng.
- Thuật toán BFS là một chiến lược tìm kiếm có tính tối ưu (vì duyệt theo chiều ngang nên sẽ tìm thấy trạng thái mục tiêu gần nhất trước) và tính đầy đủ (luôn tìm thấy trạng thái mục tiêu nếu tồn tại). Tuy nhiên, cũng giống như DFS (Depth-first Search), thuật toán BFS cũng vấp phải sự bùng nổ về số trạng thái phải duyệt (không gian tìm kiếm lớn) trong một bài toán/trò chơi, chẳng hạn như trò chơi Sudoku (ở mức độ cực khó).

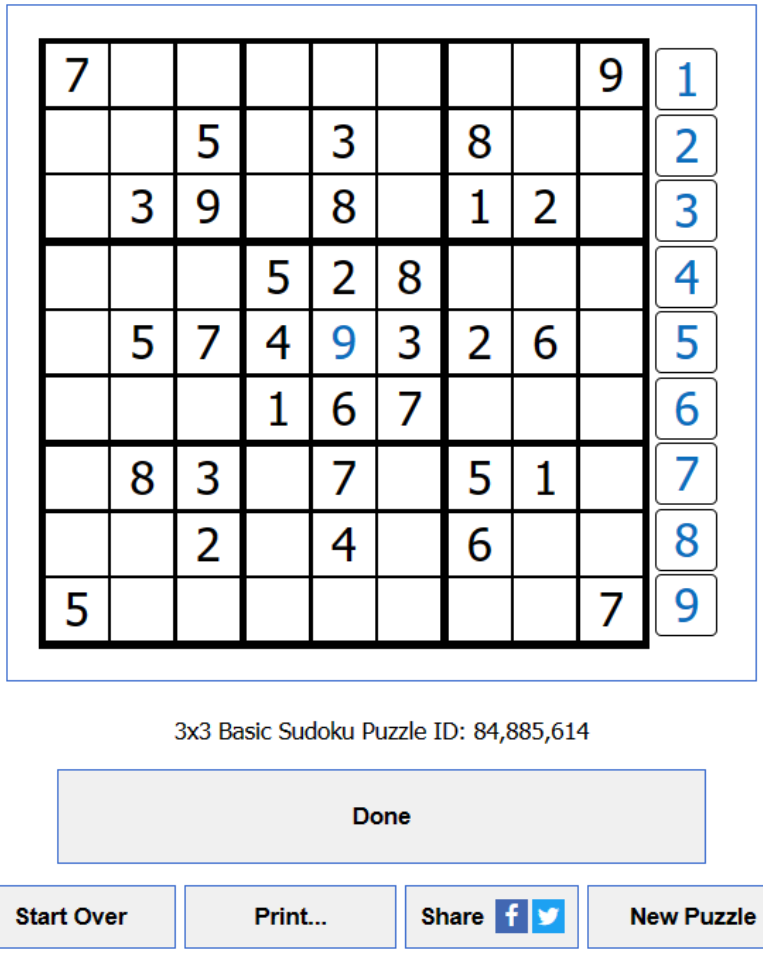
2.4 Simulated Annealing (Tôi luyện mô phỏng):

- Simulated Annealing (Tôi luyện mô phỏng) là một chiến lược tìm kiếm theo kinh nghiệm (heuristic search), mà chúng ta sẽ hy sinh tính tối ưu và tính đầy đủ trong việc tìm kiếm, để cho ra lời giải tối ưu nhất (và tất nhiên cũng có thể không tìm ra lời giải nào).
- Ý tưởng chính của Simulated Annealing là mô phỏng quá trình làm nguội (annealing) trong việc làm cho vật liệu từ trạng thái nóng đến trạng thái lạnh (hay tôi luyện kim loại). Tuy nhiên, thực tế thì quá trình thay đổi năng lượng từ trạng thái nóng (lỏng) sang trạng thái lạnh (rắn) không luôn đơn điệu giảm, mà đôi khi tăng lên. Xác suất để đột biến tăng xảy ra là rất nhỏ, bằng $e^{-\Delta E/kT}$, trong đó k là hằng số Boltzmann, ΔE là độ tăng năng lượng, và T là nhiệt độ tôi luyện ở thời điểm đang xét. Xác suất này sẽ giảm dần theo thời gian (theo nhiệt độ) theo một lịch trình được thiết lập trước.
- Simulated Annealing là một thuật toán cải tiến của thuật toán leo đồi (Hill Climbing). Tuy nhiên, khác với thuật toán leo đồi đơn giản, thuật toán này cho phép các bước chuyển trạng thái từ tốt xuống xấu với cách tính xác suất được lấy ý tưởng từ việc tôi luyện kim loại. Điều này cũng tương tự trong cuộc sống, không phải lúc nào một người cũng có thể đạt được kết quả của ngày hôm nay tốt hơn ngày hôm qua. Đôi khi, trên con đường đến thành công, cần có những bước lùi.

3 GAME 1: SUDOKU

3.1 Giới thiệu về game và luật chơi:

- **Sudoku** là một trò chơi câu đố sắp xếp chữ số dựa trên logic theo tổ hợp. Nguồn gốc của **Sudoku** được cho là xuất phát từ phát minh của nhà toán học **Leonhard Euler** - với cái tên **Hình vuông Latinh** - đó là sự sắp xếp các con số sao cho chúng không trùng lặp theo chiều ngang lẫn chiều dọc. Mặc dù vậy, trò chơi này chỉ thực sự được nhiều người biết đến thông qua một nhà phát hành trò chơi câu đó Nikoli của Nhật Bản, dưới cái tên **Sudoku** ("con số độc nhất" trong tiếng Nhật).
- Trò chơi **Sudoku** là một trò chơi giải đố logic phổ biến, đòi hỏi người chơi phải điền vào tất cả các ô trống nhưng con số mà không vi phạm các luật của Sudoku. Trò chơi này có rất nhiều biến thể, chẳng hạn như:
  - kích thước 9×9 ô, chia làm 3×3 vùng (dạng chuẩn)
  - Kích thước 4×4 ô chia làm 2×2 vùng.
  - Kích thước 6×6 ô chia làm 2×3 vùng.
  - Kích thước 5×5 ô chia vùng theo Pentomino (được phát hành với tên gọi Logi-5).
  - Kích thước 7×7 ô chia vùng theo Heptomino...
- Trò chơi **Sudoku** có các luật sau:
  - Mỗi hàng ngang, mỗi hàng dọc và mỗi khối con 3×3 đều chứa tất cả 9 chữ số (1 đến 9), không cần theo đúng thứ tự từ thấp đến cao.
  - Mỗi chữ số chỉ xuất hiện duy nhất 1 lần trong mỗi hàng, mỗi cột và mỗi khối con 3×3.
- Tuy nhiên, mỗi câu đố thường có nhiều cách giải và đáp án khác nhau. Khi vị trí các con số cho trước trên bảng 9x9 thay đổi, cách giải đố cũng sẽ thay đổi theo.
- Độ khó của trò chơi Sudoku chuẩn (kích thước 9×9, chia làm 3×3 vùng) được chia làm 5 mức độ, phụ thuộc vào **số lượng clues** (số lượng các con số đã được điền sẵn trong câu đố):
  - Cực dễ: có số lượng clues $>$ 46.
  - Dễ: số lượng clues từ 36 $\rightarrow$ 46.
  - Trung bình: số lượng clues từ 32 $\rightarrow$ 35.
  - Khó: số lượng clues từ 28 $\rightarrow$ 31.
  - Cực khó: số lượng clues từ 17 $\rightarrow$ 27.



Hình 1: *Trò chơi Sudoku trên website của Logic Puzzle.*

3.2 Các giải thuật được sử dụng giải trò chơi Sudoku:

3.2.1 Breadth-first Search (Tìm kiếm theo chiều rộng):

3.2.1.a Cách giải trò chơi:

- B1: Đầu tiên, ta khởi tạo queue[] chứa các Node chưa được duyệt và visited[] chứa các Node đã được duyệt. Đưa Node ban đầu (trạng thái ban đầu) vào trong queue[].
- B2: Bắt đầu vòng lặp trên queue[]. Vòng lặp sẽ chỉ được thực thi nếu số lượng phần tử trên queue[] $>$ 0. Lấy Node đầu tiên của queue[], kiểm tra xem Node đó đã được duyệt chưa, nếu đã được duyệt thì đẩy nó ra khỏi queue[], chuyển sang Node kế tiếp trong queue[].
- B3: Nếu Node đang xét chưa được duyệt, ta sẽ sinh các Node con hợp lệ của Node đang xét đó, đồng thời kiểm tra tình trạng của Node đang xét. Nếu Node đang xét là lời giải, ta sẽ thoát khỏi vòng lặp và trả về lời giải. Nếu Node đang xét không phải là lời giải, ta sẽ đưa Node đang xét đó ra khỏi queue[], và đưa vào trong visited[].

- B4: Tiếp tục xét Node tiếp theo trong queue[], cho đến khi tìm được lời giải hoặc không có lời giải khả thi (khi số lượng phần tử trên queue[] = 0).

3.2.1.b Các testcase mẫu:

- Mức độ dễ (số lượng clues = 36):

```
Easy Mode + BFS Algorithm:
==============================
Initial State:
------+-------+------
0 6 5 | 0 3 0 | 4 0 0
2 0 0 | 4 0 0 | 0 0 6
0 0 7 | 9 0 0 | 3 2 5
------+-------+------
4 0 0 | 3 9 5 | 7 0 8
5 0 8 | 6 2 0 | 9 0 0
0 9 0 | 0 7 4 | 0 1 0
------+-------+------
7 0 0 | 0 0 0 | 0 0 1
0 0 1 | 0 4 6 | 2 0 0
0 0 0 | 0 1 3 | 0 0 4
------+-------+------
```

Hình 2

```
Final State:
------+-------+------
8 6 5 | 1 3 2 | 4 7 9
2 3 9 | 4 5 7 | 1 8 6
1 4 7 | 9 6 8 | 3 2 5
------+-------+------
4 1 2 | 3 9 5 | 7 6 8
5 7 8 | 6 2 1 | 9 4 3
6 9 3 | 8 7 4 | 5 1 2
------+-------+------
7 5 4 | 2 8 9 | 6 3 1
3 8 1 | 5 4 6 | 2 9 7
9 2 6 | 7 1 3 | 8 5 4
------+-------+------
Count calls: 7209
Elapsed time: 0.01099705696105957 seconds
Memory used: 33.36 MB
```

Hình 3

- Mức độ trung bình (số lượng clues = 32):

```
Medium Mode + BFS Algorithm:
================================
Initial State:
------+-------+------
0 0 0 | 1 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 6 0
0 9 8 | 6 0 0 | 3 2 1
------+-------+------
0 0 7 | 0 0 0 | 9 0 0
0 0 5 | 0 1 0 | 0 8 2
0 0 2 | 3 0 7 | 1 5 0
------+-------+------
0 0 0 | 7 9 0 | 8 1 5
0 0 0 | 8 4 3 | 2 9 0
0 0 0 | 2 0 1 | 6 0 0
------+-------+------
```

Hình 4

```
Final State:
------+-------+------
2 3 6 | 1 8 5 | 4 7 9
7 1 4 | 9 3 2 | 5 6 8
5 9 8 | 6 7 4 | 3 2 1
------+-------+------
1 4 7 | 5 2 8 | 9 3 6
3 6 5 | 4 1 9 | 7 8 2
9 8 2 | 3 6 7 | 1 5 4
------+-------+------
4 2 3 | 7 9 6 | 8 1 5
6 5 1 | 8 4 3 | 2 9 7
8 7 9 | 2 5 1 | 6 4 3
------+-------+------
Count calls: 1111167
Elapsed time: 2.8971002101898193 seconds
Memory used: 206.44 MB
```

Hình 5

- Mức độ khó (số lượng clues = 28):

```
Hard Mode + BFS Algorithm:
================================
Initial State:
------+-------+------
3 0 5 | 0 0 2 | 8 7 0
4 0 7 | 0 9 0 | 0 0 0
1 0 0 | 6 7 0 | 0 0 0
------+-------+------
0 0 0 | 0 0 9 | 0 3 0
0 0 4 | 7 0 0 | 0 0 0
0 7 0 | 0 0 3 | 0 0 0
------+-------+------
7 0 6 | 9 2 0 | 1 8 0
0 0 3 | 0 0 0 | 0 0 0
9 0 0 | 0 0 7 | 5 0 6
------+-------+------
```

Hình 6

```
Final State:
------+-------+------
3 6 5 | 1 4 2 | 8 7 9
4 2 7 | 3 9 8 | 6 1 5
1 9 8 | 6 7 5 | 3 2 4
------+-------+------
5 8 1 | 2 6 9 | 4 3 7
2 3 4 | 7 5 1 | 9 6 8
6 7 9 | 4 8 3 | 2 5 1
------+-------+------
7 5 6 | 9 2 4 | 1 8 3
8 4 3 | 5 1 6 | 7 9 2
9 1 2 | 8 3 7 | 5 4 6
------+-------+------
Count calls: 958536
Elapsed time: 2.482635021209717 seconds
Memory used: 193.42 MB
```

Hình 7

- Mức độ cực khó (số lượng clues = 17): Thời gian cho ra kết quả quá lâu, kết hợp với việc bộ nhớ bị sử dụng quá nhiều → không thể giải mức độ này bằng thuật toán BFS.

3.2.2 Simulated Annealing (Tôi luyện mô phỏng):

3.2.2.a Cách giải trò chơi:

- B1: Khởi tạo và lượng giá trạng thái khởi đầu.
- B2: Bắt đầu vòng lặp cho đến khi đạt đến trạng thái mục tiêu hoặc không còn tác vụ để thử:
  - Thiết lập giá trị T theo cách điều chỉnh nhiệt độ trong quá trình tôi luyện kim loại.
  - Chọn một tác vụ để thử (ở đây là việc điền một cách ngẫu nhiên các con số từ 1 đến 9 vào những ô trống).
  - Lượng giá trạng thái mới do tác vụ vừa sinh ra. Có 3 trường hợp:
    - Nếu đó là trạng thái mục tiêu thì dừng vòng lặp, và trả về kết quả cuối cùng.
    - Nếu trạng thái mới tốt hơn trạng thái hiện tại thì ta sẽ chuyển sang trạng thái mới đó.
    - Nếu trạng thái mới xấu hơn trạng thái hiện tại, thì ta chuyển sang trạng thái mới với xác suất $e^{-\Delta E/kT}$.

3.2.2.b Các testcase mẫu:

- Mức độ dễ (số lượng clues = 36):

```
Easy Mode + SA Algorithm:
================================
Initial State:
------+-------+------
9 6 0 | 0 0 0 | 7 0 0
8 4 3 | 0 6 7 | 0 0 9
0 5 7 | 0 8 0 | 2 0 3
------+-------+------
2 0 0 | 0 0 0 | 0 0 0
0 0 8 | 0 0 0 | 0 2 0
5 0 6 | 0 0 2 | 3 8 0
------+-------+------
0 8 0 | 5 7 6 | 0 0 0
6 2 1 | 0 4 0 | 8 0 5
7 3 0 | 0 2 0 | 0 9 0
------+-------+------
```

Hình 8

```
Final State:
------+-------+------
9 6 2 | 1 5 3 | 7 4 8
8 4 3 | 2 6 7 | 5 1 9
1 5 7 | 9 8 4 | 2 6 3
------+-------+------
2 7 4 | 6 3 8 | 9 5 1
3 1 8 | 7 9 5 | 6 2 4
5 9 6 | 4 1 2 | 3 8 7
------+-------+------
4 8 9 | 5 7 6 | 1 3 2
6 2 1 | 3 4 9 | 8 7 5
7 3 5 | 8 2 1 | 4 9 6
------+-------+------
Elapsed time: 0.576331615447998 seconds
Memory used: 32.11 MB
```

Hình 9

- Mức độ trung bình (số lượng clues = 32):

```
Medium Mode + SA Algorithm:
==============================
Initial State:
------+-------+------
0 0 0 | 0 1 0 | 5 3 7
3 0 0 | 0 7 0 | 0 0 4
2 5 7 | 3 0 4 | 6 8 0
------+-------+------
0 0 0 | 8 4 0 | 0 0 0
0 7 0 | 9 0 0 | 0 0 0
5 0 0 | 0 6 0 | 0 0 0
------+-------+------
9 0 0 | 0 8 1 | 0 0 0
0 3 2 | 0 5 9 | 0 6 8
8 4 0 | 6 0 0 | 0 0 0
------+-------+------
```

Hình 10

```
Final State:
------+-------+------
4 9 6 | 2 1 8 | 5 3 7
3 1 8 | 5 7 6 | 2 9 4
2 5 7 | 3 9 4 | 6 8 1
------+-------+------
1 2 9 | 8 4 3 | 7 5 6
6 7 4 | 9 2 5 | 8 1 3
5 8 3 | 1 6 7 | 4 2 9
------+-------+------
9 6 5 | 7 8 1 | 3 4 2
7 3 2 | 4 5 9 | 1 6 8
8 4 1 | 6 3 2 | 9 7 5
------+-------+------
Elapsed time: 0.5259780883789062 seconds
Memory used: 32.08 MB
```

Hình 11

- Mức độ khó (số lượng clues = 28):

```
Hard Mode + SA Algorithm:
==============================
Initial State:
------+-------+------
0 0 0 | 0 0 0 | 0 5 0
4 0 0 | 5 0 6 | 0 0 3
3 2 5 | 0 7 0 | 0 0 0
------+-------+------
2 0 0 | 1 0 0 | 0 0 9
1 9 0 | 0 2 7 | 0 0 5
7 0 0 | 3 0 0 | 0 0 0
------+-------+------
0 4 0 | 0 0 2 | 0 7 0
9 0 0 | 0 0 8 | 5 0 4
5 0 0 | 0 4 0 | 0 0 0
------+-------+------
```

Hình 12

```
Final State:
------+-------+------
8 1 6 | 2 3 9 | 4 5 7
4 7 9 | 5 8 6 | 1 2 3
3 2 5 | 4 7 1 | 8 9 6
------+-------+------
2 5 3 | 1 6 4 | 7 8 9
1 9 4 | 8 2 7 | 6 3 5
7 6 8 | 3 9 5 | 2 4 1
------+-------+------
6 4 1 | 9 5 2 | 3 7 8
9 3 2 | 7 1 8 | 5 6 4
5 8 7 | 6 4 3 | 9 1 2
------+-------+------
Elapsed time: 1.1024796962738037 seconds
Memory used: 32.16 MB
```

Hình 13

- Mức độ cực khó (số lượng clues = 17):

```
Evil Mode + SA Algorithm:
==============================
Initial State:
------+-------+------
0 0 0 | 0 0 0 | 0 0 0
0 0 0 | 0 0 0 | 0 0 8
0 9 0 | 0 0 0 | 0 1 0
------+-------+------
0 0 0 | 8 0 0 | 0 0 4
7 0 0 | 2 0 0 | 0 0 1
9 0 0 | 0 0 0 | 6 0 0
------+-------+------
0 7 0 | 3 0 0 | 0 0 0
2 0 0 | 0 0 0 | 0 0 0
8 6 0 | 0 7 0 | 0 0 3
------+-------+------
```

Hình 14

```
Final State:
------+-------+------
1 8 4 | 5 2 7 | 3 9 6
3 2 5 | 6 1 9 | 7 4 8
6 9 7 | 4 3 8 | 5 1 2
------+-------+------
5 1 6 | 8 9 3 | 2 7 4
7 4 8 | 2 5 6 | 9 3 1
9 3 2 | 7 4 1 | 6 8 5
------+-------+------
4 7 1 | 3 6 5 | 8 2 9
2 5 3 | 9 8 4 | 1 6 7
8 6 9 | 1 7 2 | 4 5 3
------+-------+------
Elapsed time: 0.5093812942504883 seconds
Memory used: 31.97 MB
```

Hình 15

3.3 Đánh giá thời gian chạy và tiêu tốn bộ nhớ của 2 giải thuật:

3.3.1 Breadth-first Search:

- Mức độ dễ (số lượng clues = 36):

	Thời gian chạy (giây)	Bộ nhớ sử dụng (MB)	Số lần gọi đệ quy
1	0.010997	33.36	7209
2	0.116944	46.07	67779
3	0.030066	35.19	19026
4	0.002006	32.23	1053
5	0.008001	33.07	4419
6	0.019056	34.18	11070
7	0.102084	42.89	55926
8	0.009001	32.84	3969
9	0.017008	34.09	9729
10	0.043035	36.99	25389
Trung bình 10 lần thử	0.0358198	36.091	20556.9

Hình 16

- Mức độ trung bình (số lượng clues = 32):

	Thời gian chạy (giây)	Bộ nhớ sử dụng (MB)	Số lần gọi đệ quy
1	2.8971	206.44	1111167
2	0.014023	33.07	6957
3	0.35902	61.02	152064
4	0.097076	42.12	52137
5	3.397859	157.13	718173
6	0.704008	91.27	311625
7	0.575599	82.77	267705
8	0.129687	46.29	73575
9	0.072999	39.59	36648
10	0.511255	80.26	249543
Trung bình 10 lần thử	0.8758626	83.996	297959.4

Hình 17

- Mức độ khó (số lượng clues = 28):

	Thời gian chạy (giây)	Bộ nhớ sử dụng (MB)	Số lần gọi đệ quy
1	2.482635	193.42	958536
2	4.959878	245.11	1567872
3	0.131	46.43	72351
4	0.414044	68.83	206271
5	13.934561	303.1	3237867
6	12.417911	275.41	2797083
7	0.686742	80.95	287550
8	3.596394	212.43	1180341
9	1.88135	148.75	628749
10	9.47277	300.98	2384280
Trung bình 10 lần thử	4.9977285	187.541	1332090

Hình 18

3.3.2 Simulated Annealing:

- Mức độ dễ (số lượng clues = 36):

	Thời gian chạy (giây)	Bộ nhớ sử dụng (MB)
1	0.576331	32.11
2	0.051076	32.04
3	0.645955	32.15
4	0.058033	32
5	0.355088	32.02
6	0.138	31.99
7	0.067632	32.16
8	0.408781	31.99
9	0.096591	32.04
10	0.462709	31.93
Trung bình 10 lần thử	0.2860196	32.043

Hình 19

- Mức độ trung bình (số lượng clues = 32):

	Thời gian chạy (giây)	Bộ nhớ sử dụng (MB)
1	0.525978	32.08
2	0.329579	32
3	0.050005	32.04
4	0.401138	31.89
5	0.264003	31.99
6	0.19093	31.93
7	5.304871	31.98
8	0.267308	32.16
9	0.259004	31.99
10	0.395586	31.97
Trung bình 10 lần thử	0.7988402	32.003

Hình 20

- Mức độ khó (số lượng clues = 28):

	Thời gian chạy (giây)	Bộ nhớ sử dụng (MB)
1	1.10248	32.16
2	5.219087	32.07
3	0.632984	32.07
4	0.192307	32.04
5	1.359182	31.84
6	5.196761	32.14
7	0.248159	31.98
8	0.272572	32.01
9	0.382625	32.11
10	0.431	32.1
Trung bình 10 lần thử	1.5037157	32.052

Hình 21

- Mức độ cực khó (số lượng clues = 17):

	Thời gian chạy (giây)	Bộ nhớ sử dụng (MB)
1	0.509381	31.97
2	0.267087	32.12
3	0.355826	31.92
4	0.583001	32.05
5	0.110034	32.1
6	0.709308	32.07
7	0.2608	31.99
8	0.544159	32
9	0.846734	31.94
10	0.456731	32.09
Trung bình 10 lần thử	0.4643061	32.025

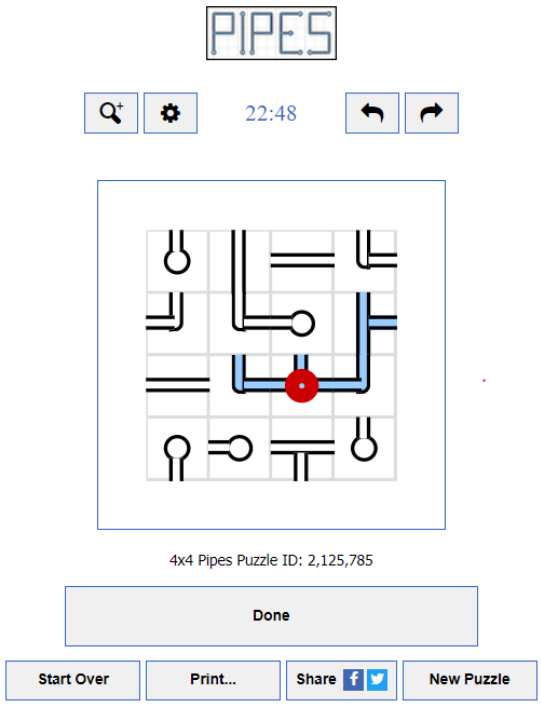
Hình 22

3.4 Kết luận:

- Dựa vào những phân tích trên, ta có thể kết luận: giải thuật Simulated Annealing là giải thuật hiệu quả hơn trong việc giải trò chơi Sudoku, đảm bảo được kết quả và thời gian giải (kể cả những màn chơi cực khó), trong khi giải thuật Breadth-first Search chỉ có thể giải được các màn chơi Sudoku ở mức độ dễ, trung bình và khó, ở mức độ cực khó (evil, #clues = 17) thì lại không thể giải được (hoặc giải rất lâu, vượt quá thời gian cho phép).
- Giải thuật Simulated Annealing cũng không tốn bộ nhớ nhiều so với giải thuật Breadth-first Search, do Breadth-first Search sử dụng đệ quy khi chạy, cho nên sẽ cần không gian bộ nhớ nhiều qua các lần đệ quy, còn Simulated Annealing thì không.

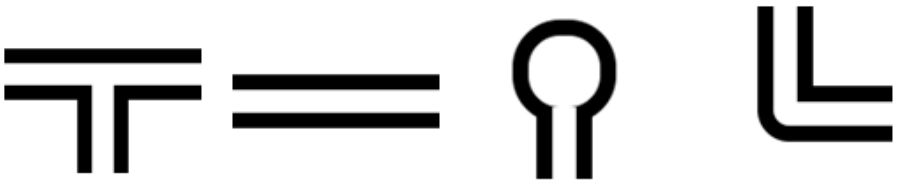
4 GAME 2: PIPES PUZZLE GAME

Trò chơi **Pipes Puzzle** được cho là có nguồn gốc từ trò chơi **Valve Puzzle**, được phát minh vào đầu những năm 1900. Trò chơi **Valve Puzzle** có thiết kế đơn giản hơn trò chơi **Pipes Puzzle** hiện đại, với các ô vuông chỉ có thể chứa các van hoặc đường ống. Trò chơi **Pipes Puzzle** hiện đại được phát triển vào những năm 1980, với sự ra mắt của các máy tính cá nhân.**Trò chơi Pipes Puzzle**, còn được gọi là trò chơi nối ống nước, trò chơi ghép đường ống, trò chơi nối đường ống, là một trò chơi giải đố logic phổ biến đòi hỏi người chơi sắp xếp các đường ống để tạo ra một đường dẫn nước từ nguồn đến đích. Trò chơi này thường được chơi trên bảng hình vuông hoặc hình chữ nhật với các ô vuông có thể chứa đường ống hoặc các chướng ngại vật. Mục tiêu của trò chơi là tạo ra một đường dẫn nước hoàn chỉnh từ nguồn đến đích mà không rò rỉ.



Hình 23: *Trò chơi pipes puzzle trên site Logic Puzzle.*

Trong Pipes Puzzle có sử dụng 4 loại ống:



(a) *Ống Chữ T* **(b)** *Ống Thẳng* **(c)** *Ống kết thúc* **(d)** *Ống Vuông*

4.1 Ứng dụng Depth-First Search trong Pipes Puzzle

4.1.1 Xác định vấn đề

- **Trạng thái của bài toán**: Mỗi trạng thái được nhóm biểu diễn bằng một bộ gồm 2 ma trận:
  - ma trận nxn biểu diễn các ống trên ô lưới với các giá trị nguyên dương đại diện cho các ống không chứa nước, và các giá trị nguyên âm đại diện cho các ống chứa nước.
  - ma trận 1x2 là ma trận tọa độ của vị trí đang xét trên ô lưới.
  - **Note** tọa độ của ống nguồn, nhóm chọn cố định là [2,2]
- **Trạng thái khởi đầu**: là bộ gồm ma trận nxn được cho từ các testcase, và tọa độ đầu tiên trong ô lưới [0,0].
- **Bước di chuyển hợp lệ**: Ở một thời điểm chỉ có thể xoay ống tại một vị trí với góc xoay thuộc một trong các góc (0, 90, 180, 270).
- **Trạng thái mục tiêu**: Khi các ống đều có nước (ma trận nxn chỉ chứa các giá trị âm) và thông với nhau (không tồn tại ống nào bị hở).

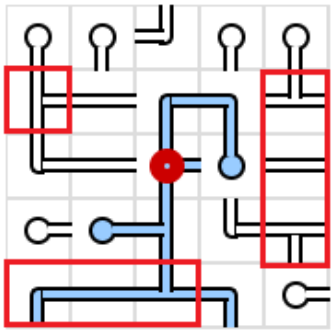
4.1.2 Sử dụng DFS để giải bài toán

Từ vấn đề của bài toán ta có thể xác định được rằng, không gian trạng thái của bài toán Pipes có thể biểu diễn bằng cấu trúc dữ liệu cây với:

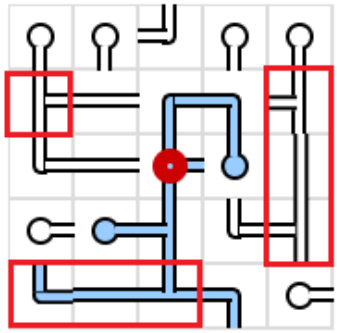
- Hệ số phân nhánh là 4 (tương ứng với 4 góc xoay)
- Độ cao tối đa của cây là nxn - 1 (root được chọn là vị trí đầu tiên [0,0] nên còn lại nxn - 1 ô lưới)
- Không tồn tại node nào có hai node cha trở lên.

Từ đó ta có thể giảm không gian tìm kiếm trước khi duyệt cây bằng cách:

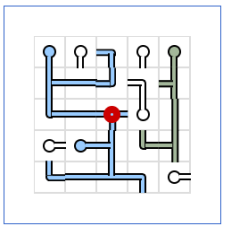
- Loại bỏ các vị trí cố định để giảm độ cao tối đa của cây (vị trí cố định là vị trí mà tại đó mỗi ống chỉ có một góc xoay duy nhất hợp lệ, vị trí cố định nếu có sẽ nằm trên rìa của Puzzle). Như hình bên dưới các vị trí được tô màu đỏ là các vị trí cố định: [1,0], [1,4], [2,4],...
- Thứ tự duyệt cây khi áp dụng DFS sẽ là từ trái sang phải, từ trên xuống dưới, do đó hệ số phân nhánh sẽ được giảm bằng việc loại bỏ các giá trị không thực tế khi xoay. Như ví dụ bên dưới, giả sử ta đang thực hiện xoay tại vị trí [1,2]. vì vị trí bên trên [0,2] và vị trí bên trái [1,1] đã được duyệt nên vị trí đang xét hiện tại phải xoay ống sao cho kết nối với cả hai vị trí [1,1] và [0,2] nếu có. Hình (a) hợp lệ vì kết nối với cả 2, hình (b), (c), (d) không hợp lệ vì chưa kết nối với một hoặc hai.



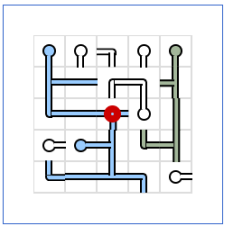
(a) *Xác định các vị trí*



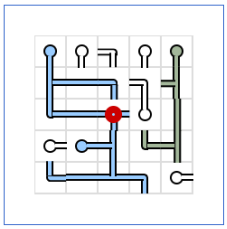
(b) *Điều chỉnh góc xoay*



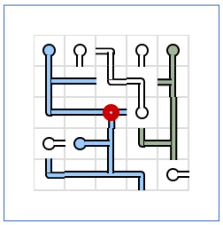
(a) *Hợp lệ*



(b) *Không hợp lệ*



(c) *Không hợp lệ*



(d) *Không hợp lệ*

Các bước thực hiện giải thuật:

- **Bước 1**: Tìm các vị trí cố định
- **Bước 2**: Đưa trạng thái sau khi tìm vị trí cố định vào stack
- **Bước 3**: Nếu stack không rỗng sẽ tiến hành:
  - **Bước 3.1**: Pop phần tử ra khỏi stack
  - **Bước 3.2**: Cập nhật lại danh sách đường đi
  - **Bước 3.3**: Tính toán các góc xoay hợp lệ
  - **Bước 3.4**: Kiểm tra có đạt trạng thái mục tiêu. Nếu có dừng và trả về kết quả.
  - **Bước 3.5**: Thêm các node con vào stack, quay lại **Bước 3**

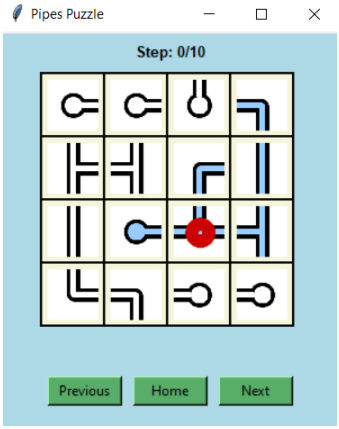
4.1.3 Kết quả

Chạy demo một vài testcase mẫu

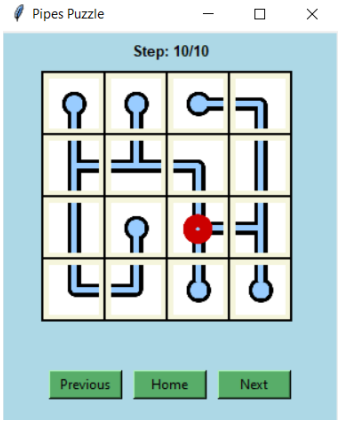
- testcase1

```
testcase1.txt
Result      Loops       Total Steps     Space (bytes)   Execution Time (hh:mm:ss)
-------------------------------------------------------------------------------
1           59          10              24552           0:00:00.013977
-------------------------------------------------------------------------------
```

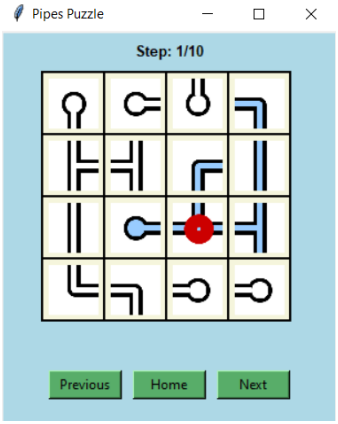
(a) *Kết quả chạy*



(a) *Ma trận ban đầu*



(b) *Bước 1*



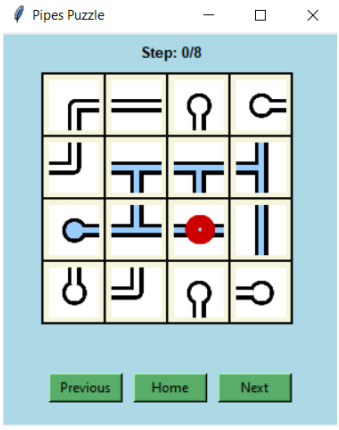
(c) *Bước cuối*

- testcase2

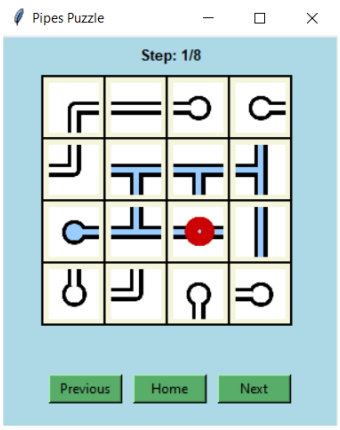
testcase2.txt

Result	Loops	Total Steps	Space (bytes)	Execution Time (hh:mm:ss)
1	33	8	22604	0:00:00.007547

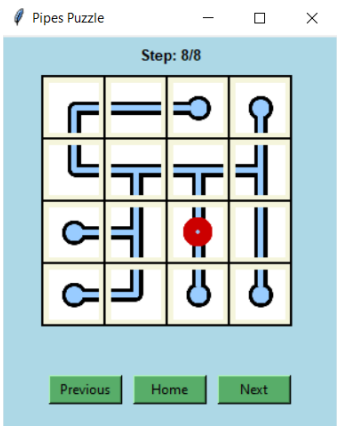
(a) *Kết quả chạy*



(a) *Ma trận ban đầu*



(b) *Bước 1*

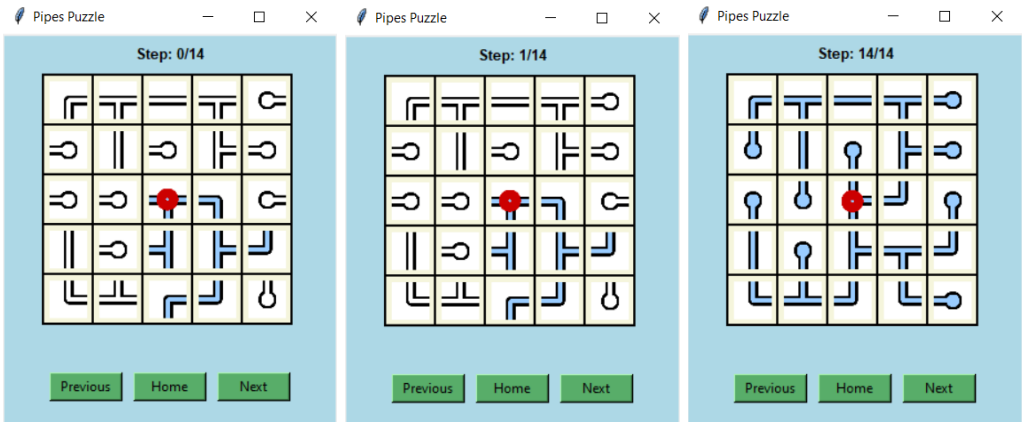


(c) *Bước cuối*

- testcase5

```
testcase5.txt
Result      Loops       Total Steps     Space (bytes)    Execution Time (hh:mm:ss)
-------------------------------------------------------------------------------------
1           75          14              21396            0:00:00.024026
-------------------------------------------------------------------------------------
```

(a) *Kết quả chạy*



(a) *Ma trận ban đầu* **(b)** *Bước 1* **(c)** *Bước cuối*

4.1.4 Đánh giá

- Tiêu tốn bộ nhớ và thời gian thực thi trong bảng thống kê bên dưới là giá trị trung bình của 100 lần chạy với mỗi input. Tiêu tốn bộ nhớ sẽ giảm đi kể từ lần chạy đầu tiên nếu thực hiện liên tiếp trên cùng một testcase nhờ vào cơ chế quản lý bộ nhớ của Python.
- các ma trận ở trường hợp 1 và 2 là ma trận 4x4, ma trận ở trường hợp còn lại là 5x5.
- **Item trong stack** là số lượng item tối đa mà stack lưu trữ khi chạy giải thuật.
- Thời gian thực thi bị ảnh hưởng bởi số lượng Node đã duyệt để đi đến node mục tiêu, nếu node mục tiêu nằm ở nhánh càng xa nhánh ngoài cùng bên trái, thì số lượng node phải duyệt tăng lên làm thời gian thực thi tăng theo bởi vì DFS là giải thuật duyệt theo chiều sâu, phải đi hết node cuối cùng của một nhánh mới duyệt sang nhánh tiếp theo. Vậy nên ở trường hợp 4 mặc dù là ma trận 5x5 nhưng lại có thời gian thực thi nhanh hơn so với trường hợp 1.
- Về tiêu tốn bộ nhớ, nhìn chung các ma trận 5x5 sẽ tốn nhiều bộ nhớ hơn các ma trận 4x4 vì kích thước cần cho các ma trận sẽ lớn hơn và các node duyệt qua sẽ nhiều hơn. Ở trường hợp 3 và 4 mặc dù số lượng node đã duyệt ở trường hợp 3 cao hơn nhưng số lượng item trong stack lại thấp hơn trường hợp 4. Nên bộ nhớ tiêu tốn ở trường hợp 3 sẽ thấp hơn, nguyên nhân vì có thể cơ chế dọn dẹp rác của Python.

ID	Tiêu tốn bộ nhớ (bytes)	Node đã duyệt	Items trong stack	Thời gian thực thi (giây)
testcase1	13151	59	5	0.011
testcase2	12004	33	5	0.006
testcase3	20784	67	4	0.016
testcase4	23574	29	7	0.009
testcase5	20652	75	5	0.017
testcase6	29540	167	10	0.052

Bảng 1: *Bảng thống kê*

4.2 Ứng dụng A* trong Pipes Puzzle

4.2.1 Ý tưởng giải trò chơi bằng giải thuật

- Ta khởi tạo một ma trận với kích thước là 5x5, trong đó mỗi ô trong ma trận sẽ được đại diện bởi 1 class Node, dùng để lưu trữ các trạng thái có thể thăm trong tương lai và lịch sử các trạng thái trước đó của ống nước trên ô. Và tất cả các ô sẽ được vào một List đại diện cho trạng thái hiện tại của trò chơi, và mỗi phần tử của List sẽ chứa 2 thông tin là: **loại ống và hướng quay**.

```
    class Node:
    def __init__(self,matrix: list, rotate: list, previous) -> None:
        self.state = State(matrix)
        self.state.triggledAt([2,2])
        # vi tri thay doi so voi node cu
        self.rotate = rotate
        if previous == None:
            self.previous = None
            self.step = 0
        else:
            self.previous = previous
            self.step = previous.step + 1

```

- Cốt lõi của giải thuật nằm ở việc hiện thực 3 hàm tính toán chi phí là: **f(x), g(x), h(x)**. Trong đó:
  - **h(x)**:là hàm heuristic đóng vai trò quan trọng nhất trong việc xác định node trạng thái của ống nước nào sẽ được push vào **OpenList** tiếp theo. cố gắng ước tính chi phí (hoặc khoảng cách) từ một trạng thái đã cho đến trạng thái mục tiêu với giá trị ánh xạ trả về dưới dạng một giá trị nguyên. Logic hiện tại trong hàm này dựa trên các quy tắc sau:
    - *Tăng phần giá trị đánh giá hx nếu số lượng bump (các phần cuối ống bị tắc) giảm xuống.*
    - *Tăng giá trị đánh giá hx nếu các ống ở các cạnh có góc phù hợp (dẫn đến tiềm năng tạo kết nối).*
    - *Giảm phần đánh giá nếu có đường đi không phù hợp, có thể tạo ra ngõ cụt.*
    - *Áp dụng penalty (+ 2000 vào hx cho trạng thái hiện tại) nếu phát hiện vòng lặp (cho thấy cấu hình không thể giải quyết được).*

- **g(x)**: Hàm này tính toán chi phí thực tế g(x) để đến trạng thái hiện tại. Trong triển khai này, chi phí được tính đơn giản dựa trên số bước (các vòng xoay ống) nhân với 2.
- **f(x)**: Hàm này kết hợp chi phí thực g(x) và giá trị heuristic h(x) để tính toán ước lượng tổng chi phí f(x) đối với một trạng thái nhất định trong thuật toán.

```
    def hx(self, current: Node)->int:
    ...{implement function h(x)}...
    def gx(self, current: Node):
        if current == None:
            return 0
        return current.step*2

    def fx(self, current: Node) -> int:
        return self.gx(current) + self.hx(current)

```

- Trong Class **solve_Astar** nơi hiện thực chính giải thuật A*, khởi tạo 2 List lần lượt là: **OpenList** và **CloseList** để lưu trữ các trạng thái đã sinh ra nhưng chưa thăm đến trên ma trận và các trạng thái đã được duyệt qua trên ma trận. Khởi tạo biến **current_state** để nhận trạng thái đầu tiên được lấy ra từ **OpenList**, việc gán này sẽ lặp lại cho đến khi List **OpenList** không còn giá trị nào hoặc nếu biến **current_state** đạt trạng thái kết thúc (khi 25 ống nước đều được đổ đầy nước) thì sẽ thông báo **"Kết thúc giải thuật!"**.
- Ngoài ra, thì initState của **solve_Astar** sẽ là nguồn nước với tọa độ là [2;2] và khônn tồn tại node trước đó (None), và hàm tính chi phí heuristic đi kèm với node khởi đầu cho nguồn nước này.

```
        def solve_Astar(self, init_state: list):
        head = Node(init_state, [2,2], None)
        head.heuristics = self.fx(head)
        head.step = 0
        openList = [[self.fx(head),head]]
        dataForPlot = {0:1}
        closeList = []
        while len(openList) != 0:
            current_state = openList.pop(0)
            if current_state[1].step not in dataForPlot:
                dataForPlot.update({current_state[1].step:1})
            else:
                dataForPlot[current_state[1].step] += 1
        ...

```

- Ngoài ra, thì các trạng thái được tạo ra mới từ **current_state** sẽ được thêm vào **OpenList** để đảm bảo tìm ra đường đi ngắn nhất. **OpenList** được sắp xếp lại sau mỗi lần lặp để ưu tiên các nút có hàm heuristic nhỏ hơn.
- Kết thúc thuật toán, thì hàm sẽ trả về tổng bộ nhớ mà giải thuật tìm kiếm sử dụng: bằng tổng **độ dài của danh sách OpenList và CloseList**, bên cạnh đó là data tạo ra để mô phỏng các bước tìm kiếm qua mỗi step trong quá trình tìm kiếm ứng với mỗi Testcase khác nhau.

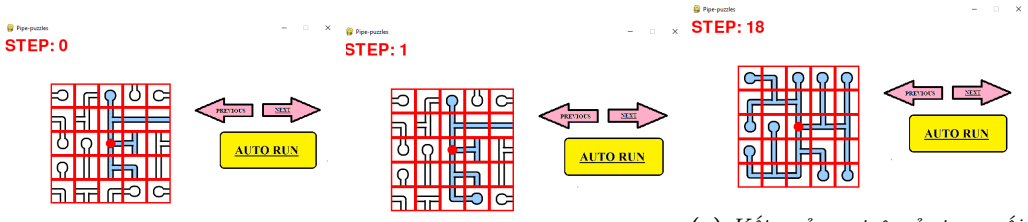
4.2.2 Testcase mẫu - solution từ giải thuật

Khởi tạo node chứa nguồn nước ở vị trí **[2;2]** trên ma trận kích cỡ là 5x5, với trạng thái node trước đó gán cho nguồn nước là sẽ **None**. Sau đó bằng giải thuật tìm kiếm A* với việc thăm các Node ở gần nhất

- Testcase số 1:

```
1
Doi vai giay!!!
Ket thuc giai thuat !
Tong bo nho su dung:  3382 bytes
Tong thoi gian thuc hien tim kiem:  3.263
```

Hình 33: *Kết quả về độ hiệu quả thời gian và bộ nhớ testcase số 1*

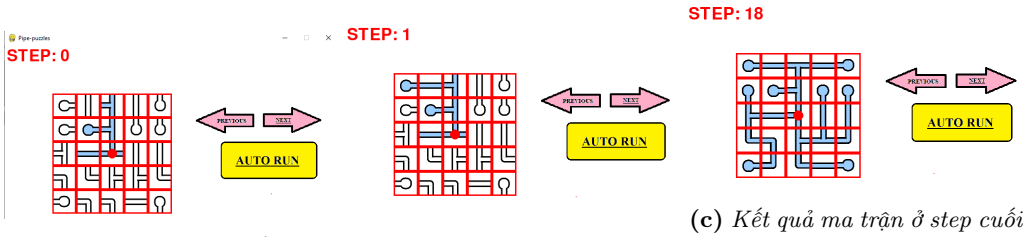


(a) *Ma trận ban đầu* **(b)** *Ma trận sau step 1* **(c)** *Kết quả ma trận ở step cuối cùng*

- Testcase số 2:

```
2
Doi vai giay!!!
Ket thuc giai thuat !
Tong bo nho su dung:  1408 bytes
Tong thoi gian thuc hien tim kiem:  0.9522
```

Hình 35: *Kết quả về độ hiệu quả thời gian và bộ nhớ testcase số 2*

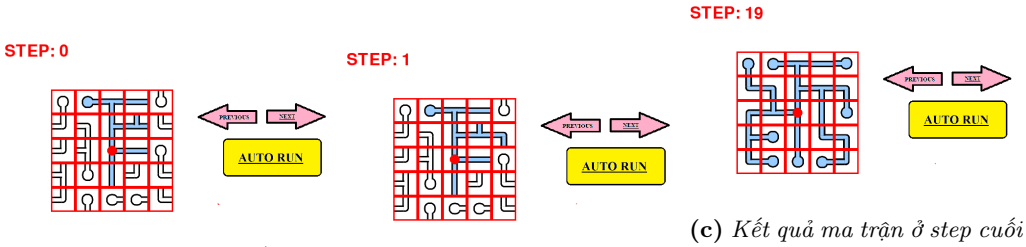


(a) *Ma trận ban đầu* **(b)** *Ma trận sau step 1* **(c)** *Kết quả ma trận ở step cuối cùng*

- Testcase số 3:

```
3
Doi vai giay!!!
Ket thuc giai thuat !
Tong bo nho su dung:  3095 bytes
Tong thoi gian thuc hien tim kiem:  2.8485
```

Hình 37: *Kết quả về độ hiệu quả thời gian và bộ nhớ testcase số 3*



(a) *Ma trận ban đầu* **(b)** *Ma trận sau step 1* **(c)** *Kết quả ma trận ở step cuối cùng*

4.2.3 Đánh giá thời gian chạy và tiêu tốn bộ nhớ của giải thuật

- Thời gian thực thi việc tìm kiếm và giải quyết trò chơi sẽ tăng dần lên với độ phức tạp của từng testcase. Trong điều kiện tốt nhất, độ phức tạp thời gian của thuật toán A* trong trò chơi Pipes Puzzle có thể gần như tuyến tính theo số lượng ô trong lưới, được biểu diễn bằng **O(N)(bestCase)**, với N là số lượng ô trong lưới. Trong testcase mà ta sử dụng thì **kích thước của ma trận là: 5x5**. Ngoài ra thì về mặt lý thuyết, độ phức tạp về thời gian trong trường hợp xấu nhất của A* sẽ là **O(N^d)(WorstCase)**, với d là độ sâu tối đa của cây tìm kiếm.

- Độ phức tạp thực tế thấp hơn nhiều, nhưng để phân tích chính xác còn phụ thuộc vào cấu hình ban đầu của ma trận khởi tạo ban đầu cho các ống dẫn, bên cạnh đó là vị trí đặt các ống dẫn có tạo nên các vòng lặp vô hạn không thoát ra được hay không, và các trạng thái gây bất lợi lân cận khác đối với vị trí ống nước hiện tại.

- Còn với mức độ tiêu tốn bộ nhớ thì xét về mặt lý thuyết, các ma trận với kích cỡ 5x5 như đang được demo hiện tại sẽ tiêu tốn một lượng bộ nhớ lớn hơn nhiều so với việc sử dụng ma trận kích cỡ 4x4. Bởi lúc này số lượng node, trạng thái đã tăng lên so với 4x4 kéo theo đó thì quá trình tìm kiếm và lưu trữ các node đã thăm cũng như các node mới tìm ra ở trạng thái tiếp theo vào 2 list được khởi tạo sẽ lớn hơn, dẫn đến yêu cầu về bộ nhớ để lưu trữ trạng thái của ống nước sẽ tăng lên nhiều so với ma trận cấp 4.

5 TÀI LIỆU THAM KHẢO

[1] Introduce DFS. Được truy cập từ: http://en.wikipedia.org/wiki/Depth-first_search

[2] Sudoku. Được truy cập từ https://en.wikipedia.org/wiki/Sudoku

[3] Sudoku: Number of clues for each difficulty level. Được truy cập từ https://www.researchgate.net/figure/Number-of-clues-for-each-difficulty-level_tbl1_259525699

[4] Simulated Annealing. Được truy cập từ https://en.wikipedia.org/wiki/Simulated_annealing

[5] Khái quát về thuật toán A*. Được truy cập từ: https://www.iostream.co/article/thuat-giai-a-DVnHj