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

NHẬP MÔN TRÍ TUỆ NHÂN TẠO

Bài tập lớn số 1

Trò chơi Bloxorz

Giảng viên hướng dẫn: Vương Bá Thịnh
Sinh viên: Lê Tấn Lộc 2011572
Hoàng Tiến Hải 2011152
Đặng Diễm Quỳnh 2011956
Trương Hoàng Nguyên Vũ 2112673

TP. HỒ CHÍ MINH, THÁNG 3/2023

Mục lục

1 Kiến thức chuẩn bị

1.1 Một số thư viện tiêu biểu được sử dụng

Các thư viện được sử dụng trong chương trình chủ yếu được dùng để xử lí input và hiển thị kết quả trực quan, đồng thời dùng để đo thời gian và sự tiêu tốn bộ nhớ của giải thuật. Cụ thể:

- queue: Chức sẵn các loại hàng đợi dùng trong giải thuật tìm kiếm.
- numpy: thư viện chuyên về các phép toán trên ma trận, được dùng trong giải thuật Monte-Carlo Tree Search.
- pygame: hiển thị trực quan các giải thuật cũng như tạo sự sinh động cho bài toán.
- time: tính thời gian giải thuật chạy.

1.2 Giải thuật tìm kiếm

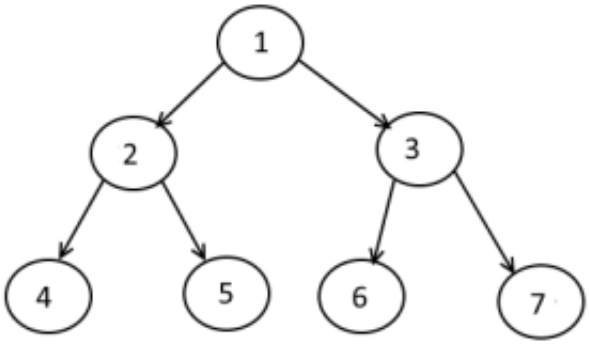
1.2.1 Depth-First Search

Thuật toán DFS khởi đầu từ gốc (hoặc một đỉnh nào đó coi như gốc) và tiến hành tìm kiếm theo hướng xa nhất ở mỗi nhánh. Thuật toán thường được sử dụng để tìm kiếm phần tử trong cây hoặc đồ thị.

Để hình dung rõ hơn, ta hãy tìm hiểu bài toán duyệt phần tử trong một cây nhị phân dùng thuật toán DFS.

Bắt đầu từ nút 1, thuật toán sẽ duyệt qua nhánh ngoài cùng bên trái, nếu nút đang duyệt không có nút con hoặc tất cả nút con đã được duyệt thì quay ngược trở lại và duyệt qua phần tử bên phải. Kết quả, thứ tự duyệt cây con lần lượt là : 1-2-4-5-3-6-7.

Đối với bài toán duyệt cây có chiều sâu là n, yếu tố phân nhánh là m thì giải thuật DFS có độ phức tạp thời gian là $O(m^n)$, độ phức tạp không gian là $O(mn)$.



Duyệt cây bằng DFS. Nguồn: algorithms.tutorialhorizon.com

1.2.2 A* Search

Giải thuật tìm kiếm A* tìm một đường đi từ một nút khởi đầu tới một nút đích cho trước. Thuật toán này sử dụng một hàm đánh giá heuristic để xếp loại từng nút và duyệt chúng theo thứ tự của đánh giá này. Do đó, giải thuật này là một trường hợp của tìm kiếm theo lựa chọn tốt nhất (Best-first Search).

Hàm đánh giá mà A* sử dụng là:

$$f(n) = h(n) + g(n)$$

Trong đó $h(n)$ là chi phí đường đi ngắn nhất từ nút n đến trạng thái đích; $g(n)$ là chi phí đường đi ngắn nhất từ trạng thái ban đầu đến nút n.

Tuy nhiên trong thực tế, rất khó để tìm ra giá trị chính xác của hàm $h(n)$ và $g(n)$ vì chúng ta chưa biết trạng thái đích của bài toán. Do đó, trong giải quyết bài toán thực tế người ta thường sử dụng hàm ước lượng $f^*(n) = h^*(n) + g^*(n)$, trong đó $h^*(n)$ là ước lượng giá trị của $h(n)$; $g^*(n)$ được tính đến thời điểm đang xét bởi giải thuật và là xấp xỉ của giá trị $g(n)$.

1.2.3 Monte-Carlo Tree Search

Monte Carlo Tree Search (MCTS) là một giải thuật lặp được dùng để tìm kiếm trong không gian trạng thái và xây dựng các số liệu thống kê cho các hành động có sẵn ở mỗi trạng thái cụ thể. Nói một cách đơn giản hơn, mỗi khi đến một trạng thái cụ thể với các hành động khác nhau, phương pháp sẽ cho chạy mô phỏng như một lượt chơi bình thường, kết quả mỗi lần chạy sau đó sẽ được thống kê lại và hành động tốt nhất ở trạng thái đó có thể được xác định thông qua các giá trị kết quả ấy.

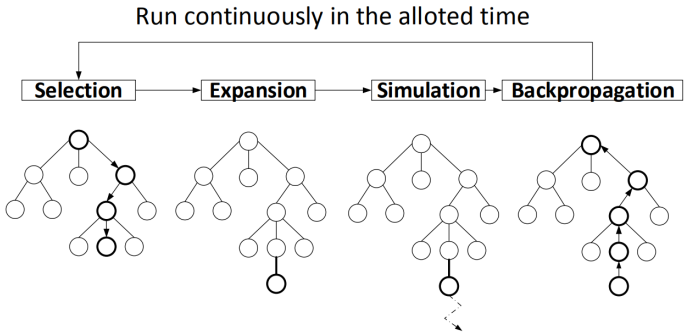
Trong một bài toán không tầm thường (non-trivial), không gian trạng thái chỉ đơn giản là không thể được tìm kiếm cách đầy đủ. Trong các ứng dụng thực tế của phương

pháp, MCTS chỉ được thực hiện trong một số lần lặp cụ thể hoặc bị giới hạn bởi thời gian đưa ra hành động tốt nhất. Bằng cách thực hiện mô phỏng theo những giới hạn nêu trên, phương pháp có thể đưa ra được hành động tốt nhất bằng cách áp dụng phương trình sau:

$$a^* = \underset{a \in A(s)}{ARGMAX}\ Q(s,a)$$

Trong đó, $A(s)$ là tập các hành động có sẵn ở trạng thái s, trong các hành động được thực thi thì $Q(s,a)$ là giá trị kinh nghiệm (kết quả) trung bình của trạng thái s khi thực hiện hành động a. Theo lẽ thông thường, số lượng mô phỏng càng nhiều, độ tin cậy của các giá trị kinh nghiệm càng tăng từ đó các giá trị có thể cho ta hành động nào là tốt nhất với độ tối ưu cao nhất.

Ở mỗi lần lặp của giải thuật sẽ có bốn giai đoạn khác nhau. Các giai đoạn đó bao gồm: Lựa chọn (Selection), Mở rộng (Expansion), Mô phỏng (Simulation) và Lan truyền ngược (Backpropagation). Hình ảnh minh họa của từng giai đoạn được cho ở bên dưới:



Hình 1: Các trạng thái của Monte Carlo Tree Search

Hình ảnh trên là mô tả rất tốt cho hoạt động của từng giai đoạn trong từng lần lặp của MCTS, trong đó các giai đoạn được hiểu như sau:

- **Lựa chọn (Selection):** Ở giai đoạn này, giải thuật tìm kiếm trong một phần các trạng thái đã có sẵn trong bộ nhớ của MCTS. Lựa chọn luôn bắt đầu ở node gốc và ở mỗi tầng (level), lựa chọn node tiếp theo dựa trên phương án lựa chọn của cây (tree policy). Giai đoạn này dừng lại khi đã đến được node kết thúc (node terminal) chứa trạng thái kết thúc của vấn đề (terminal state) hoặc là node lá của MCTS và không có trạng thái tiếp theo nào đến đươc từ node đó tồn tại trong bộ nhớ của MCTS.
- **Mở rộng (Expansion):** Giai đoạn mở rộng sẽ thêm vào bộ nhớ của MCTS ít nhất một node con mới, nếu trong giai đoạn lựa chọn tìm ra một node kết thúc thì không thực hiện giai đoạn này. Node con được thêm vào này chứa trạng thái đạt được sau khi thực hiện hành động cuối cùng vào giai đoạn lựa chọn. Khi giai đoạn mở rộng đạt được node kết thúc (mặc dù điều này rất khó xảy ra), giải thuật lập tức nhảy qua giai đoạn cuối cùng.

- **Mô phỏng (Simulation):** Ở giai đoạn này, thực hiện mô phỏng một cách ngẫu nhiên hoàn toàn cho đến khi đến được trạng thái kết thúc và có được kết quả của lần mô phỏng đó. Đây chính là phần "Monte Carlo" trong MCTS.
- **Lan truyền ngược (Backpropagation):** Kết quả được thấy ở giai đoạn mô phỏng sẽ được lan truyền lên các node nằm trên con đường dẫn từ node kết thúc của giai đoạn mô phỏng đến node gốc. Từ giá trị kết quả này, các giá trị thống kê được cập nhật.

Phương án lựa chọn của cây (tree policy) có mục đích là làm sao để cân bằng được giữa sự khám phá (exploration) các node hay trạng thái chưa được xem xét kĩ và sự khai phá (exploitation) các node được đánh giá là tốt nhất theo các thông số hiện tại. Phương pháp phổ biến nhất hiện giờ chính là UCT (Upper Confidence Bounds applied for Tree). Ta có công thức cho phương pháp này như sau:

$$a^* = \underset{a \in A(s)}{ARGMAX} \left\{ \; Q(s,a) + C\sqrt{\frac{ln[N(s)]}{N(s,a)}} \; \right\}$$

Công thức này được dùng để kiểm tra từng hành động một, từ đó đưa ra hành động nên được lựa chọn. Trong đó, $A(s)$ là tập hợp các hành động có sẵn ở trạng thái s, $Q(s,a)$ chính lá giá trị kết quả trung bình khi thực hiện hành động a ở trạng thái s trong quá trình mô phỏng và được thống kê từ suốt quá trình đến giờ, $N(s)$ là số lần trạng thái s đã được đi qua trong các lần lặp trước và $N(s,a)$ chính là số lần hành động a được làm mẫu ở trạng thái s. Hằng số C điều chỉnh sự cân bằng giữa sự khám phá và khai phá của cả quá trình. Thông thường giá trị C sẽ được đặt là $\sqrt{2}$ với điều kiện rằng các giá trị Q nằm trong khoảng $[0,1]$, nhưng nói chung đây là một tham số phụ thuộc vào tính chất bài toán, có thể điều chỉnh cho phù hợp với các bài toán cụ thể.

Bằng cách sử dụng công thức của UCT như trên và mô phỏng ngẫu nhiên (random rollout policy), MCTS dựng nên cây trò chơi không đối xứng khi so sánh với các giải thuật truyền thống khác. Ở trong cây tìm kiếm MCTS, các node có giá trị kết quả cao hơn sẽ được mô phỏng và chạy thử nhiều hơn.

2 Định nghĩa bài toán

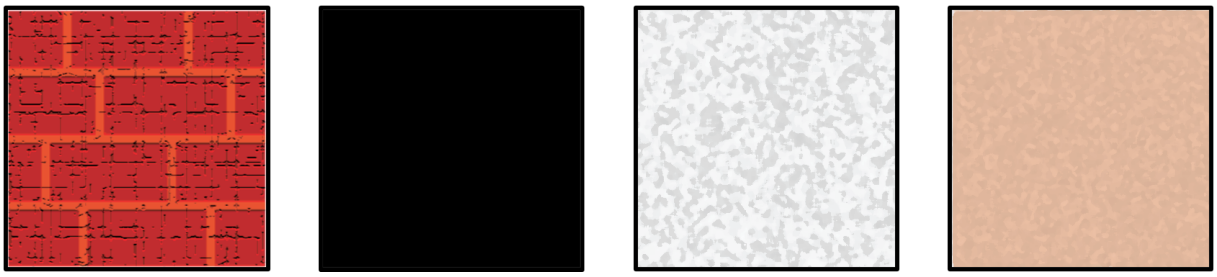
2.1 Giới thiệu về trò chơi Bloxorz

Bloxorz là một trò chơi giải đố 3D, được phát triển bởi Damien Clarke vào năm 2007.

Về cơ bản, người chơi sẽ di chuyển một khối hình hộp chữ nhật (2a) có kích thước 1x2 theo bốn hướng sao cho nó rơi xuống lỗ thoát (2b) ở cuối mỗi màn chơi. Trò chơi kết thúc khi người chơi di chuyển khối rớt khỏi platform (phần nền) của màn chơi.

Platform là tập các ô vuông có kích thước 1x1 được phân loại như sau:

- Ô trắng (2c): loại ô bình thường cho phép khối chữ nhật tự do di chuyển.
- Ô cam (2d): loại ô mỏng yếu và sẽ bị sập khi khối chữ nhật đứng thẳng trên ô này.

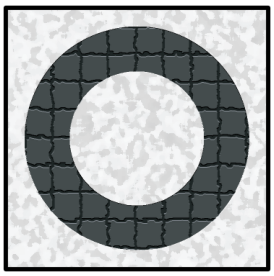


(a) Khối gạch (b) Lỗ thoát (c) Ô trắng (d) Ô cam

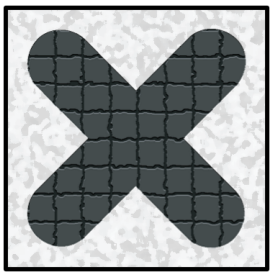
Hình 2: Các ô trên màn chơi

Ngoài ra, mỗi màn chơi có thể thêm một số công tắc nhất định hỗ trợ người chơi vượt qua như:

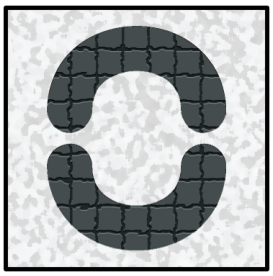
- Công tắc mềm (3a): được kích hoạt khi bất kỳ phần nào của khối chữ nhật đè lên nó. Có khả năng xây cầu.
- Công tắc cứng (3b): được kích hoạt chỉ khi khối chữ nhật đứng thẳng và đè lên nó. Có khả năng xây cầu.
- Công tắc tách (3c): được kích hoạt khi bất kỳ phần nào của khối chữ nhật đè lên nó. Có khả năng phân tách khối chữ nhật thành 2 khối lập phương, đồng thời dịch chuyển chúng đến vị trí khác nhau theo từng màn chơi. Người chơi có thể điều khiển hai khối này một cách độc lập và chúng có thể nối lại khi được đặt cạnh nhau để trở về như ban đầu.



(a) Công tắc mềm



(b) Công tắc cứng



(c) Công tắc tách

Hình 3: Các công tắc đặc biệt

Cầu là tập hợp các ô trắng được kích hoạt khi đè lên công tắc tương ứng. Trong đó, một vài loại cầu có thể được mở và đóng bởi công tắc, số còn lại thì sẽ vĩnh viễn được kích hoạt.

2.2 Định nghĩa không gian trạng thái

Một trạng thái của trò chơi Bloxorz sẽ gồm hai thành phần: Bảng (Board) và Khối chữ nhật (Box). Trong đó:

1. **Bảng (Board)**: Sẽ có dạng mảng 2 chiều. Các vị trí trên bảng sẽ có tọa độ, gốc của tọa độ $(0, 0)$ được đặt ở góc trên cùng bên trái, các thành phần của bảng được biểu diễn bằng các ký tự đại diện cho các đối tượng trong trò chơi, cụ thể:
   - Ô trắng: biểu diễn bằng ký tự '■'.
   - Ô cam: biểu diễn bằng ký tự '▩'.
   - Ô đích: biểu diễn bằng ký tự '#'.
   - Công tắc mềm: biểu diễn bằng ký tự 'O'.
   - Công tắc cứng: biểu diễn bằng ký tự 'X'.
   - Công tắc tách: biểu diễn bằng ký tự 'C'.
   - Vực thẳm: biểu diễn bằng ký tự khoảng trắng ' '.
2. **Khối (Box)**: được biểu diễn bằng một cặp tọa độ $\{(x_1, y_1), (x_2, y_2)\}$. Khi nằm, nửa thứ nhất (x_1, y_1) sẽ luôn này về phía trên, bên trái so với nửa còn lại (x_2, y_2). Khi đứng, hai tọa độ này sẽ trùng nhau.

Mỗi màn chơi sẽ có các công tắc, các công tắc này sẽ được sử dụng trong quá trình chuyển trạng thái.

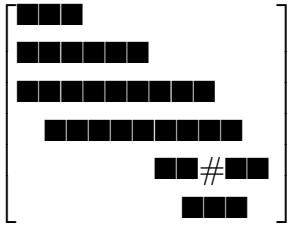
Ví dụ trạng thái của màn 1:

$$\text{Board} = \begin{bmatrix} ■■■ & & & & & & \\ ■■■■■■■ & & & & & & \\ ■■■■■■■■■■ & & & & & & \\ & ■■■■■■■■■■ & & & & & \\ & & & & ■■\#■■ & & \\ & & & & & ■■■ & \end{bmatrix}$$

$$\text{Box} = \{(1, 1), (1, 1)\}$$

- **Trạng thái mục tiêu:** Trò chơi chiến thắng khi người chơi điều khiển khối chữ nhật đứng trên ô đích. Vậy trạng thái mục tiêu đạt được khi cả hai tọa độ của Box đều trùng với tọa độ của Ô đích trên Board.

  Ví dụ: trạng thái mục tiêu của màn 1:

  Board = 

  Box = $\{(4, 7), (4, 7)\}$

- **Bước chuyển trạng thái:** mỗi bước thay đổi trạng thái sẽ thay đổi tọa độ của Box

  – *Box*: các bước chuyển trạng thái là LEFT, RIGHT, UP, DOWN, SWITCH. Trong đó, bước SWITCH chỉ hợp lệ khi Box đang ở trạng thái phân tách. Tức:

  $$(x_1 \neq x_2 \wedge y_1 \neq y_2) \vee (x_1 = x_2 \wedge |y_1 - y_2| > 1) \vee (y_1 = y_2 \wedge |x_1 - x_2| > 1)$$

  Các bước chuyển LEFT, RIGHT, UP, DOWN sẽ dựa vào trạng thái hiện tại của Box (đứng, nằm ngang, nằm dọc) để thay đổi các tọa độ sao cho tọa độ đầu tiên luôn ở phía trên-bên trái tọa độ còn lại.

  – *Board*: đôi khi việc điều khiển Box sẽ làm kích hoạt các công tắc trên Board, dẫn đến một số giá trị trên Board có thể bị thay đổi. Các công tắc này sẽ được lưu như một đối tượng dùng để chuyển trạng thái cho Board.

- **Bước chuyển trạng thái hợp lệ**: Một bước chuyển trạng thái hợp lệ là bước chuyển không làm cho trò chơi thua, tức tọa độ của Box phải nằm trong khoảng xác định của Board và không nằm trên Vực thẳm (các vị trí có giá trị là khoảng trắng).

3 Giải quyết bài toán

3.1 Depth-First Search

Chiến lược tìm kiếm ban đầu của nhóm là Tree search. Giải thuật được thực hiện như sau:

1. Khởi tạo một stack rỗng và đẩy trạng thái khởi đầu vào trong stack
2. Lấy trạng thái ở trên đỉnh stack ra.
3. Nếu trạng thái này là trạng thái mục tiêu thì trả về đường đi từ trạng thái khởi đầu đến trạng thái này và kết thúc giải thuật.
4. Sinh ra các trạng thái con của trạng thái này dựa vào các bước chuyển trạng thái.
5. Đẩy các trạng thái con vào trong stack.
6. Nếu stack không rỗng thì trở lại bước 2. Ngược lại, báo lỗi không tìm thấy trạng thái mục tiêu.

Giải thuật Tree-DFS tương đối hiệu quả với các màn có kích thước Board nhỏ. Tuy nhiên, khi kích thước Board lớn hơn giải thuật thực thi rất chậm. Ngoài ra, trong một số trường hợp, giải thuật còn bị rơi vào vòng lặp vô tận. Lý do là vì giải thuật Tree Search không có bước kiểm tra lại những trạng thái đã duyệt qua, dẫn đến việc tốn nhiều thời gian để duyệt lại các nhánh dư thừa, cũng như dễ bị rơi vào vòng lặp. Do đó, nhóm đã chuyển sang chiến lược Graph Search để tránh trường hợp trên. Giải thuật được thực hiện tương tự Tree Search. Tuy nhiên, trước khi đẩy các trạng thái con vào trong stack, ta sẽ kiểm tra xem trạng thái này đã được duyệt qua chưa, nếu chưa thì sẽ đẩy vào trong stack và thêm trạng thái này vào tập đã duyệt. Ngược lại nếu trạng thái đã được duyệt thì bỏ qua và kiểm tra trạng thái con tiếp theo.

3.2 A* Search

Nhóm chọn giải thuật tìm kiếm A* với chiến lược Graph Search. Hàm lượng giá của giải thuật được định nghĩa là:

$$f(n) = g(n) + h(n)$$

Trong đó: $g(n)$ là số nước đi đã thực hiện kể từ trạng thái khởi đầu.

$h(n)$ là tổng khoảng cách từ vị trí hai nửa của Box đến Ô đích.

Hàm $g(n)$ sẽ giúp A* chọn những nhánh có số nước đi ít nhất, nhằm đưa ra lời giải tối ưu hơn so với DFS.

Hàm $h(n)$ sẽ giúp A* chọn những nước đi giúp Box đến gần Ô đích hơn. Việc sử dụng tổng khoảng cách từ cả 2 phần của Box giúp tăng trọng số của hàm $h(n)$ so với $g(n)$ khiến A* tập trung vào việc tiến gần đến Ô đích hơn. Ngoài ra, nó còn giúp A* vẫn hoạt động tốt trong trường hợp công tắc tách được kích hoạt, khiến cho 2 nửa của Box ở 2 vị trí cách xa nhau.

3.3 Monte Carlo Tree Search (MCTS)

Như đã được đề cập từ trước về giải thuật, MCTS là một giải thuật tìm kiếm ngẫu nhiên và từ thống kê các nước chơi ngẫu nhiên đó đưa ra một nước đi hợp lý nhất. Tuy vậy điều này chỉ hiệu quả với trờ chơi hai người chơi, vốn dĩ từ khi được đề xuất giải thuật tìm kiếm này chủ yếu nhắm vào các trò chơi có nhiều người chơi chứ không phải cho việc giải puzzle (trò chơi một người chơi). Do đó, cần có một số biến đổi, thay đổi phù hợp để có thể áp dụng tốt nhất giải thuật cho bài toán Bloxorz đề bài đưa ra.

Sau khi đánh giá và phân tích về trò chơi Bloxorz cũng như chơi thử trò chơi, nhóm chúng em có một số nhận xét như sau:

- Đây là một trò chơi một người chơi, và đây là trò chơi có thể hoàn lại nước đi (trừ một số trường hợp đi vào nút có chức năng cố định không thể hoàn lại).
- Số nước đi để dẫn đến đích đến (goal) là rất ít nếu so với các nước đi có thể đi trong khi chơi trò chơi.
- Số nước đi rác dẫn đến trạng thái kết thúc nhưng không thắng là vô cùng lớn và rất dễ xảy ra.
- Rất nhiều nút chức năng xuyên suốt các màn của trò chơi, tạo nên sự đa dạng về số nước đi khả dĩ trong mỗi màn chơi.

Giải thuật MCTS truyền thống sẽ được dùng trong việc giải quyết các màn trong trò chơi Bloxorz, các giá trị thưởng ban đầu sẽ bao gồm 1 (nếu trạng thái đích được tìm thấy) và 0 (nếu không phải là trạng thái đích). Tuy vậy sẽ có một số điều chỉnh ở các giai đoạn được áp dụng để giải quyết tốt nhất trò chơi, các điều chỉnh bao gồm:

- **Ở giai đoạn lựa chọn (Selection):** Công thức UCT được dùng trong giai đoạn này sẽ được điều chỉnh. Đối với trò chơi một người chơi, việc các nước đi có lặp lại được hay không không phụ thuộc vào người chơi khác; do đó việc lặp lại các nước đi là dễ dàng thực hiện và giá trị trung bình giá trị các nước chơi cũng không còn cần thiết. Vì thế ta thay đổi công thức để nó dùng giá trị $MAX(s,a)$ thay vì giá trị $Q(s,a)$:

$$a^* = \underset{a \in A(s)}{ARGMAX} \left\{ \; MAX(s,a) + C\sqrt{\frac{ln[N(s)]}{N(s,a)}} \; \right\}$$

  Bên cạnh đó, một ngưỡng được cung cấp để chặn đi hành vi khám phá (exploration) hay cho hằng số $C = 0$, ngưỡng này có thể được kích hoạt khi một sự kiện bất lợi hay thuận lợi được tìm thấy (có thể là đường dẫn đến chiến thắng hay đường đến ngõ cụt được tìm thấy).

- **Ở giai đoạn mở rộng (Expansion):** Việc mở rộng ngẫu nhiên được sử dụng. Bên cạnh đó cây tìm kiếm có thể chuyển qua trạng thái độc nhất nghĩa là không có node nào trên cây tìm kiếm được lặp lại. Điều này giúp giảm bớt trường hợp bùng nổ trạng thái khi đây là trò chơi có thể hoàn lại nước đi, tuy vậy khi đó cây tìm kiếm sẽ suy biến thành cây tìm kiếm bình thường nhưng với hiệu suất tệ hơn so với các giải thuật DFS hay $A*$. Về vấn đề tỉa nhánh cây tìm kiếm, việc tỉa nhánh rất khó để thực hiện do thách thức về mặt cài đặt, logic và thời gian thực thi.

- **Giai đoạn mô phỏng (Simulation):** Với việc có thể hoàn lại nước đi, một ngưỡng độ sâu tối đa được cung cấp để quá trình mô phỏng không dính vào vòng lặp vô tận. Vì trong việc giải một puzzle, mục đích đưa ra là tìm ra trạng thái tiếp theo mới và có thể đưa đến kết quả tốt hơn; vì vậy việc mô phỏng nên được thực hiện theo hướng tìm kiếm những trạng thái mà khác biệt nhất so với các trạng thái đã đi trước đó, tập số các nước đi đã đi trước được lưu lại để tìm kiếm nước đi tiếp theo, nếu không có nước đi khác mô phỏng ngẫu nhiên sẽ được thực hiện, khi số các nước đi trước được cho bằng 0, quá trình mô phỏng trở về mô phỏng ngẫu nhiên hoàn toàn. Bên cạnh đó, trong quá trình mô phỏng nếu tìm thấy trạng thái đích, lời giải sẽ được trả về ngay lập tức và kết thúc thuật toán tìm kiếm.
- **Thay đổi về các nước đi khả dĩ:** Từ nhận xét các nước đi rác là rất lớn, nhóm chúng em quyết định trong quá trình chọn nước đi để mô phỏng hay mở rộng cây chỉ chọn những nước đi sẽ không đi đến trạng thái kết thúc; từ đó giảm bớt các số lượng nước đi trong quá trình tìm kiếm và cải thiện được hiệu suất tìm kiếm.
- **Thay đổi về mặt thiên kiến (bias):** Ở một số màn chơi nhóm chúng em nhận thấy các nút chức năng rất quan trọng trong việc cải thiện quá trình chơi, do đó đã đưa ra một số trọng số cho các sự kiện quan trọng được diễn ra trong trò chơi bao gồm: Chia đôi khối và dịch chuyển, tăng số gạch xám trên bàn chơi và giảm số gạch xám trên bàn chơi. Trong đó hai sự kiện đầu tiên thường được đánh trọng số dương không cao hơn 1 (do 1 là giá trị thưởng khi đến đích) và sự kiện cuối sẽ được đánh số âm. Tuy vậy phương pháp này không áp dụng được với tất cả các màn chơi.

4 Trực quan hóa

Nhóm đã hiện thực các giải thuật tìm kiếm Depth-First Search, A* Search, Monte-Carlo Tree Search và được trực quan hoá lời giải bằng giao diện window thông qua ngôn ngữ python.

Giao diện game được nhóm viết trên thư viện **pygame** và trực quan ở dạng các hình ảnh 2D nhưng vẫn hiển thị đầy đủ các trạng thái của lời giải trong quá trình thực hiện các giải thuật tìm kiếm trong không gian trạng thái.

4.1 Cài đặt môi trường và thư viện hỗ trợ

Yêu cầu môi trường: Python 3.9 trở lên, Pip Package version được cập nhật mới nhất. Để cập nhật pip package mới nhất:

```
$ pip install --upgrade pip
```

Cài đặt các thư viện cần thiết: **pygame**

```
$ pip install pygame
```

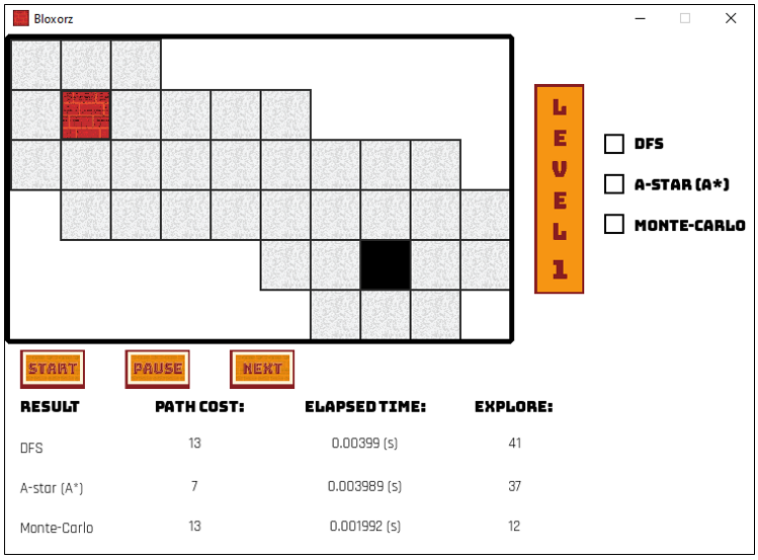
4.2 Thực thi chương trình

Để thực thi chương trình và hiện thị chế độ xem trực quan bằng giao diện, vào thư mục **Bloxorz** nằm trong source code mà nhóm đã cung cấp, thực thi lệnh sau trên terminal:

```
$ python run.py
```

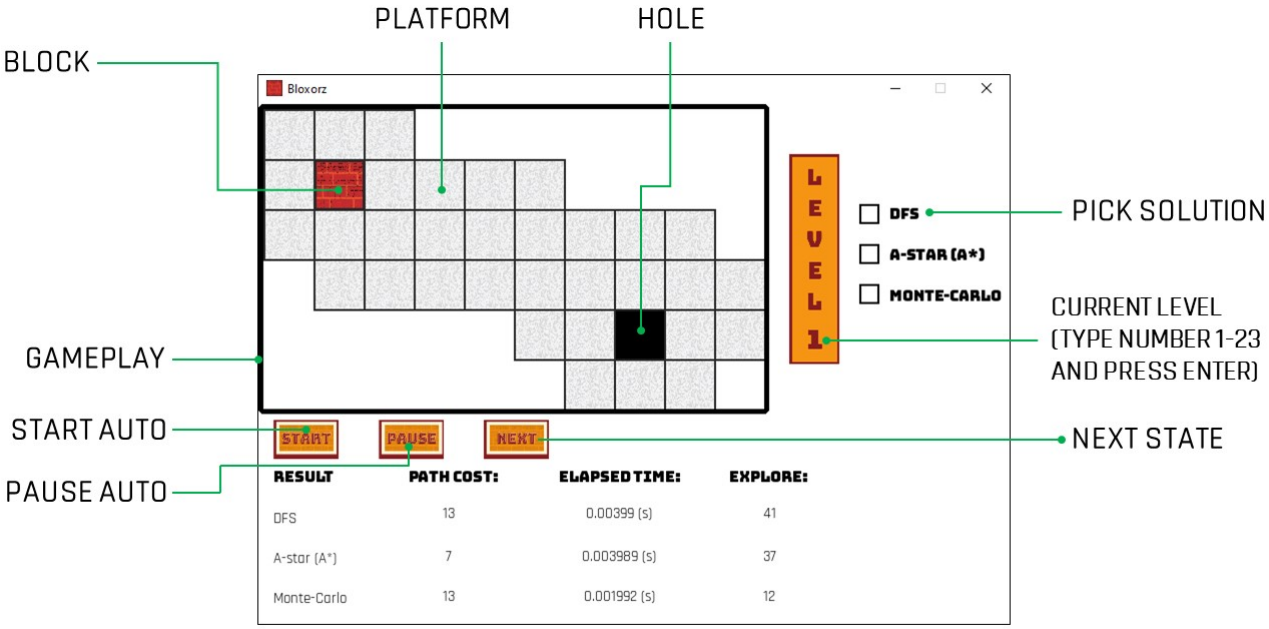
4.3 Giao diện của chương trình

Sau khi tiến hành chạy chương trình thành công, của sổ game với kích thước 750 x 520 sẽ được hiển thị dưới dạng cửa sổ như sau:



Hình 4: Cửa sổ chính của trò chơi

Ngoài ra, người dùng có thể tương tác với giao diện trò chơi thông qua các nút được chỉ dẫn trong hướng dẫn sử dụng dưới đây:



Hình 5: Cửa sổ chính của trò chơi (hướng dẫn)

Trong đó:

- **Gameplay**: Là ô được khoanh vùng màu đen bằng một hình chữ nhật nằm ở góc trái của trò chơi. Các trạng thái của trò chơi sẽ được hiển thị trong khung này.
- **Block**: Là khối gồm hai hình hộp chữ nhật xếp chồng lên di chuyển qua các trạng thái.

- **Platform**: Phần nền của trò chơi mà trên đó khối **block** có thể di chuyển được. Ngoài ra còn có nhiều loại platform được đặt trong trò chơi.
- **Hole**: Đích đến của khối **block** sau khi di chuyển trong không gian trạng thái để hoàn thành màn chơi.
- **Pick solution**: Người chơi có thể lựa chọn 1 trong 3 giải thuật (DFS, A*, Monte-Carlo) để xem giao diện trực quan của lời giải đối với màn chơi.
- **Current level**: Hiển thị màn chơi hiện tại đang được hiện thị trong khung **gameplay**. Người dùng có thể thay đổi màn chơi bằng cách nhập chuỗi số từ 1 - 23 và phím BACKSPACE để thay đổi màn chơi, sau đó nhấn phím ENTER để xác nhận thay đổi.
- **Nút Start**: Thực hiện hiển thị lời giải trực quan trong khung **gameplay** một cách tự động với lời giải được chọn bên khung **Pick solution** (mặc định được sử dụng là Depth-First Search).
- **Nút Pause**: Dừng việc tự động hiển thị trạng thái trò chơi trung ô **gameplay**.
- **Nút Next**: Tiếp tục trạng thái tiếp theo của giải thuật trong ô **gameplay**.

5 Đánh giá các giải thuật

5.1 Các giải thuật tìm kiếm thông thường

Nhóm đánh giá hiệu suất của các giải thuật dựa trên hai tiêu chí là thời gian thực thi, số node đã khai phá và số nước đi của lời giải. Nhìn chung, giải thuật A* có hiệu suất tốt hơn Graph-DFS trong hầu hết các trường hợp. Do đó, nhóm chỉ trình bày một số trường hợp tiêu biểu.

5.1.1 Các màn đầu tiên

Màn	Giải thuật	Thời gian (s)	Số nước đi	Số node đã khai phá
1	DFS	0.001	13	41
	A*	0.002	7	37
2	DFS	0.022	57	528
	A*	0.007	19	191
3	DFS	0.009	32	247
	A*	0.004	19	103

Có thể nhận thấy, đối với các màn đơn giản, kích thước Board nhỏ, không có quá nhiều công tắc thì giải thuật A* thể hiện tốt hơn so với DFS. Lời giải tìm ra ngắn hơn nhờ vào hàm $g(n)$ và số lượng nút được khai phá cũng ít hơn nhờ vào hàm $h(n)$.

5.1.2 Các màn có nhiều công tắc điều khiển cầu

Màn	Giải thuật	Thời gian (s)	Số nước đi	Số node đã khai phá
5	DFS	0.017	100	426
	A*	0.022	41	535
14	DFS	0.030	106	761
	A*	0.050	67	1121

Với việc số công tắc cầu tăng lên, việc tiến trực tiếp tới Ô đích không còn hiệu quả, do đó hàm $h(n)$ không còn nhiều giá trị, giải thuật A* sẽ phụ thuộc vào hàm $g(n)$, dẫn đến việc nó có hiệu suất tương tự như Breath-First Search. Mặt khác, giải thuật DFS không có hàm lượng giá, nên hiệu năng của nó không có quá nhiều biến động so với các màn đầu tiên.

5.1.3 Các màn có công tắc tách

Khi công tắc tách được kích hoạt, hai nửa của Box sẽ bị phân tách thành các khối 1x1, khiến cho việc điều khiển Box khó rơi vào vực thẳm hơn. Điều này làm cho DFS có nhiều lựa chọn để chuyển trạng thái hơn, do đó, lời giải mà DFS đưa ra có số lượng nước đi rất lớn so với lời giải tối ưu.

Màn	Giải thuật	Thời gian (s)	Số nước đi	Số node đã khai phá
9	DFS	0.095	476	2319
	A*	0.065	25	950
15	DFS	0.185	812	4518
	A*	0.628	69	11030

Ngược lại, với hàm heuristic là tổng khoảng cách giữa hai phần của Box đến ô đích, A* cho một lời giải với chi phí thấp và số node cần khai phá cũng ít hơn so với DFS (màn 9). Tuy nhiên, ở màn 15, ngoài công tắc tách còn có các công tắc điều khiển cầu khác, nên giải thuật A* không hiệu quả về mặt thời gian và số node khai phá ở màn này.

5.2 Đánh giá giải thuật Monte Carlo Tree Search

Ở đây nhóm chúng em đánh giá giải thuật qua các lượt chơi với các màn chơi đại diện để đánh giá mức độ tốt của giải thuật đáp ứng cho từng màn chơi cụ thể cũng như đánh giá mức độ tốt của các cải tiến được đề xuất. Để việc đánh giá được dễ dàng, một số thông tin được cung cấp về tên gọi các tham số, kí hiệu và giới hạn được dùng trong đánh giá như sau:

- **simulation_time:** Tham số chỉ thời gian tối đa giải thuật có thể thực hiện để tìm ra lời giải.
- **max_simulation_depth:** Độ sâu tối đa được áp dụng trong giai đoạn mô phỏng.
- **max_previous_nodes:** Số lượng node đã đi tối đa được lưu dùng để chọn nước đi mới khác nhất trong giai đoạn mô phỏng.
- **points_list:** Danh sách trọng số các sự kiện mặc định là None, ba giá trị lần lượt là trọng số khi phân tách, khi tăng số gạch xám và khi giảm số gạch xám.
- **is_unique_node:** Cho phép lặp lại các node trong cây tìm kiếm hay không.
- **c:** Giá trị hằng số cho phần khám phá trong công thức UCT.
- Các kí hiệu $PATH$ chỉ giá trị đường đi trả về, $TIME$ chỉ thời gian giải xong màn chơi và $MEMORY$ chỉ bộ nhớ sử dụng trong quá trình giải màn chơi.
- Giới hạn được đưa ra trong quá trình đánh giá bao gồm thời gian thực thi tối đa là $1000s$, bộ nhớ sử dụng tối đa là $1GB$. Khi vượt qua các giới hạn này, các giá trị tính toán sẽ không có do đó kí hiệu # sẽ được dùng thay thế đại diện cho việc không có giá trị.

5.2.1 Các màn chơi giải thuật tỏ ra hiệu quả

Lấy các màn chơi 1 và màn chơi 4 làm đại diện, trong hai màn chơi này nhóm chúng em áp dụng các thông số phù hợp để giúp giải thuật giải được màn chơi tốt nhất. Đối với màn 1 các thông số bao gồm: simulation_time=1000, max_simulation_depth=30, max_previous_nodes=2, is_unique_node=False, c=1.4, đối với màn chơi 4 các thông số bao gồm: simulation_time=1000, max_simulation_depth=30, max_previous_nodes=2,

is_unique_node=True, c=1.4. Cho chạy giải thuật 20 lần ta được bảng giá trị như sau:

	Level 1	**Level 4**
$TIME_{min}(s)$	0.0011	0.4368
$TIME_{max}(s)$	0.0549	1.8558
$TIME_{ave}(s)$	0.0139	1.2149
$PATH_{min}(steps)$	8	33
$PATH_{max}(steps)$	33	41
$PATH_{ave}(steps)$	19.8	38.2
$MEMORY_{min}(B)$	19628	718208
$MEMORY_{max}(B)$	247076	959334
$MEMORY_{ave}(B)$	96969.8	818210.1

Ở đây ta cũng có thể thấy được sự không ổn định của giải thuật khi một vài thông số min max tỏ ra rất khác biệt. Vì số lượng các trạng thái được duyệt và thêm vào trong cây tìm kiếm là nhỏ (do ít trường hợp phát sinh) nên ta có thể thấy rằng bộ nhớ được dùng là rất ít; cũng vì lý do đó việc đưa ra được lời giải trong khoảng thời gian ngắn là dễ dàng xảy ra.

5.2.2 Sử dụng công thức UCT chỉnh sửa trong giai đoạn lựa chọn

Nhóm chúng em đánh giá việc đưa ra lời giải chính xác khi áp dụng hai công thức là tính trung bình và tính giá trị lớn nhất có ngưỡng chặn vào màn chơi 8 với các tham số như sau: simulation_time=1000, max_simulation_depth=20, max_previous_nodes=0, is_unique_node=False, c=1.4. Khi này giải thuật sẽ trở về việc mô phỏng hoàn toàn ngẫu nhiên, cho chạy 20 lần ta được bảng kết quả sau:

	AVE UCT	**MAX UCT**
$TIME_{min}(s)$	0.0105	0.1566
$TIME_{max}(s)$	9.3462	3.1999
$TIME_{ave}(s)$	2.03263	1.3834
$PATH_{min}(steps)$	14	17
$PATH_{max}(steps)$	27	29
$PATH_{ave}(steps)$	22.1	25
$MEMORY_{min}(B)$	182837	1394294
$MEMORY_{max}(B)$	24696158	9523591
$MEMORY_{ave}(B)$	6592760	5087790

Theo các thông số ở trên, dễ thấy rằng thời gian chạy của công thức lấy giá trị lớn nhất có chút tốt hơn so với công thức tính trung bình. Có thể giải thích rằng với việc lấy giá trị lớn nhất có ngưỡng, quá trình mô phỏng sẽ tỏ ra đi đúng hướng và dễ dàng đến với lời giải hơn so với tính giá trị trung bình (do ảnh hưởng từ giá trị khám phá trong công thức UCT). Ở màn chơi 8 này, mặc dù có nút phân tách, tuy vậy khi đã kích hoạt nút phân tách thì màn chơi trở nên vô cùng đơn giản để tìm ra lời giải đến đích do không còn bất kì nút chức năng hay các ô gạch cản trở nào; do đó các giá trị thời gian và bộ

nhớ sử dụng vẫn ở mức tốt. Giá trị đường đi đến đích vẫn tỏ ra rất không ổn định, một lần nữa thể hiện sự ngẫu nhiên của giải thuật.

5.2.3 Cho phép lặp lại node ở cây tìm kiếm trong giai đoạn mở rộng

Nhóm chúng em đánh giá hai trường hợp khi cho phép và không cho phép việc lặp lại node ở cây tìm kiếm vào màn chơi 11. Ở màn chơi 11, các tham số được cho như sau: simulation_time=1000, max_simulation_depth=20, max_previous_nodes=2, c=1.4. Cho chạy thử 20 lần ta được bảng số liệu như sau:

	Unique node	**Non-unique node**
$TIME_{min}(s)$	0.7935	#
$TIME_{max}(s)$	2.9568	#
$TIME_{ave}(s)$	1.74749	#
$PATH_{min}(steps)$	50	#
$PATH_{max}(steps)$	58	#
$PATH_{ave}(steps)$	54.4	#
$MEMORY_{min}(B)$	1050900	#
$MEMORY_{max}(B)$	1424328	#
$MEMORY_{ave}(B)$	1268689.8	#

Theo các thông số ở trên, việc không cho phép lặp lại đã đưa ra kết quả rất tốt về mặt thời gian cũng như mặt tài nguyên so với việc cho phép lặp lại các node trên cây tìm kiếm. Bên cạnh đó cần nhận xét rằng các giá trị thời gian và tài nguyên bộ nhớ sử dụng khi áp dụng việc không cho phép lặp lại ở màn 11 là tốt. Có thể giải thích việc thời gian tốt bằng việc khi đã giới hạn số node trong cây tìm kiếm, việc lặp lại do nước đi hoàn lại sẽ không diễn ra do đó giúp cây tìm kiếm định hướng tốt hơn vào lời giải đến đích giúp tìm ra lời giải nhanh hơn; việc tài nguyên bộ nhớ sử dụng ít có thể được giải thích bằng việc khi đã giới hạn số lượng node có thể có ở trong cây tìm kiếm, bộ nhớ sử dụng sẽ ít đi và có giới hạn; cũng cần chú ý rằng giá trị nước đi đến đích ở màn chơi này đã ổn định hơn do lượng lời giải khả dĩ ở màn chơi này không còn nhiều như các màn chơi đã trình bày.

Tuy vậy như đã trình bày ở trong phần cải tiến, việc không cho lặp lại node sẽ biến cây tìm kiếm thành một cây tìm kiếm bình thường khác như của giải thuật DFS hay $A*$ nhưng hiệu suất tệ hơn, khi đó việc cho phép lặp lại và mở rộng ngẫu nhiên lại tỏ ra hiệu quả. Ta có thể thấy rõ điều đó ở màn chơi 14 với các tham số: simulation_time=1000, max_simulation_depth=30, max_previous_nodes=4, points_list=[0.0, 0.5, -0.5], c=1.4. Cho chạy thử 20 lần, ta được bảng số liệu sau:

	Unique node	Non-unique node
$TIME_{min}(s)$	#	15.6225
$TIME_{max}(s)$	#	112.5952
$TIME_{ave}(s)$	#	42.9702
$PATH_{min}(steps)$	#	285
$PATH_{max}(steps)$	#	421
$PATH_{ave}(steps)$	#	341.3
$MEMORY_{min}(B)$	#	7558573
$MEMORY_{max}(B)$	#	28051723
$MEMORY_{ave}(B)$	#	14026270

Các thông số ở trên chứng minh rằng không cho phép lặp lại các node đôi khi không phải là một ý hay tuy phương pháp vẫn có các lợi ích nhất định. Để giải thích cho việc này, khi một node được chọn trong giai đoạn lựa chọn và không được mở rộng thêm do các node con đã có trong cây, việc này sẽ hạn chế đi khả năng tiếp tục khám phá các node đó khi giai đoạn lựa chọn sẽ có khả năng cao rơi vào ngõ cụt như vậy một lần nữa (cho đến khi các giá trị khám phá làm thay đổi quyết định trong giai đoạn lựa chọn) dẫn đến mắc kẹt, trong khi đó việc đi ngẫu nhiên sẽ có khả năng vô tình tìm thấy một lời giải tốt hơn. Giá trị thời gian cao và có sự biến động lớn do khi cho phép lặp lại node trong cây, cây tìm kiếm sẽ ngẫu nhiên các nước đi đến khi tìm ra lời giải đến đích. Giá trị bộ nhớ sử dụng có giá trị biến động do khi không giới hạn số node, việc cây mở rộng sẽ theo kiểu bùng nổ và bộ nhớ sử dụng sẽ tỉ lệ thuận với độ lớn của cây tìm kiếm được dựng nên. Giá trị đường đi tới đích cũng phản ánh đúng sự ngẫu nhiên trong giải thuật này.

5.2.4 Ưu tiên đi theo các nước khác so với các nước đã đi trong giai đoạn mô phỏng

Nhóm chúng em đánh giá hai trường hợp khi cho giá trị các nước đã đi là 0 và các nước đã đi là 2 vào màn chơi 4; trong đó các tham số được cho như sau: simulation_time=1000, max_simulation_depth=30, is_unique_node=False, c=1.4. Cho chạy thử 20 lần, ta được bảng số liệu sau:

	Pre-node = 0	Pre-node = 2
$TIME_{min}(s)$	#	5.3399
$TIME_{max}(s)$	#	170.5771
$TIME_{ave}(s)$	#	56.6561
$PATH_{min}(steps)$	#	31
$PATH_{max}(steps)$	#	35
$PATH_{ave}(steps)$	#	32.8
$MEMORY_{min}(B)$	#	10026306
$MEMORY_{max}(B)$	#	253078824
$MEMORY_{ave}(B)$	#	89458762

Các thông số ở trên chứng minh được rằng việc cho mô phỏng đi theo ưu tiên các nước đi khác nhất mang lại kết quả tốt hơn hẳn so với việc đi ngẫu nhiên, do việc đi ngẫu nhiên dễ dẫn đến những nước đi đã đi qua do đó làm giảm hiệu suất chương trình mà

không mang lại kết quả thực sự tốt hơn. Về giá trị thời gian không ổn định, vì ở đây ta cho việc mở rộng node là không lặp lại và ngẫu nhiên nên việc mở rộng có thể tạo ra các nhánh không mong muốn và tăng thời gian xử lý lên. Đó cũng là lý do cho việc bộ nhớ sử dụng không ổn định và có giá trị lớn, vì khi kích thước cây tìm kiếm tăng do cho phép lặp lại node và mở rộng ngẫu nhiên, bộ nhớ sử dụng cũng sẽ tăng theo.

5.2.5 Đánh trọng số cho các sự kiện ở giai đoạn mô phỏng

Nhóm chúng em đánh giá việc đánh trọng số vào các sự kiện ở màn chơi 2 bao gồm việc không đánh và đánh trọng số sự kiện chia tách là 0.0, sự kiện tăng số gạch là 0.5, sự kiện giảm số gạch là -0.5. Bên cạnh đó các tham số ban đầu bao gồm: simulation_time=1000, max_simulation_depth=30, max_previous_nodes=0, is_unique_node=False, c=1.4. Cho chạy thử 20 lần ta được bảng số liệu sau:

	No weight	**Has weight**
$TIME_{min}(s)$	35.8199	0.3564
$TIME_{max}(s)$	270.8328	1.9414
$TIME_{ave}(s)$	165.0855	1.09559
$PATH_{min}(steps)$	21	32
$PATH_{max}(steps)$	35	67
$PATH_{ave}(steps)$	30.4	51.4
$MEMORY_{min}(B)$	61291237	1104626
$MEMORY_{max}(B)$	427266970	3006299
$MEMORY_{ave}(B)$	263136635	2010448

Qua bảng số liệu trên, ta thấy rằng việc đánh trọng số cho các sự kiện đã mang lại hiệu quả tốt với màn chơi 2 cho cả thời gian thực thi và bộ nhớ sử dụng. Về thời gian thực thi khi không đánh trọng số, lượt chơi ngẫu nhiên sẽ được tiến hành khi nào tìm ra lời giải thì thôi, trong khi với trọng số các sự kiện sẽ được tìm thấy trước và các sự kiện đó cũng là bàn đạp để đến với đích nên thời gian thực thi sẽ nhanh hơn. Đối với bộ nhớ sử dụng, khi thời gian thực thi nhanh hơn thì số lượng node được thêm vào cây tìm kiếm cũng sẽ ít hơn dẫn đến bộ nhớ sử dụng ít đi theo. Dễ nhận thấy các giá trị thời gian, bộ nhớ sử dụng hay đường đi đến đích đều không ổn định, đó là hệ quả tất yếu của việc đi ngẫu nhiên trong giải thuật này.

Tuy nhiên không phải lúc nào việc đánh trọng số cũng mang lại hiệu quả, trong một số màn khi các sự kiện tốt phụ thuộc vào các sự kiện tồi để đi đến kết quả tốt hơn, phương pháp này tỏ ra vô cùng kém hiệu quả.

6 Kết luận

Thông qua trò chơi giải đố Bloxorz, nhóm đã tìm hiểu về các phương pháp tìm lời giải cho bài toán, phân tích hiệu suất của giải thuật tìm kiếm theo chiều sâu (DFS) và giải thuật heuristic (A*) theo độ phức tạp của bài toán và mô phỏng trực quan cách hai giải thuật hoạt động với một số màn tiêu biểu. Có thể tóm tắt như sau:

Về cách thức hoạt động, giải thuật tìm kiếm theo chiều sâu sẽ đi từ phần tử hiện tại đến phần tử lân cận chưa được duyệt; trong khi đó, giải thuật tìm kiếm A* lại chọn đi đến phần tử chưa duyệt có hàm chi phí thấp nhất. Cả hai giải thuật sau đó sẽ thực hiện thay đổi và kiểm tra trạng thái hiện tại đã là trạng thái mục tiêu chưa, quá trình này được lặp cho đến khi đạt được trạng thái đích.

Về mặt hiệu suất, giải thuật A* đưa ra lời giải với chi phí thấp (số nước đi ít), nhưng sẽ đánh đổi với thời gian và số lượng node cần khai phá khi trạng thái trò chơi trở nên phức tạp và hàm heuristic không thể hoạt động hiệu quả. Ngược lại, giải thuật DFS tuy không đảm bảo đưa ra lời giải tối ưu, nhưng lại khá ổn định về mặt thời gian và số node cần khai phá.

Như vậy, chúng ta cần phải xác định được tính chất cũng như độ phức tạp của bài toán, từ đó lựa chọn được thuật toán giúp tối ưu thời gian tìm kiếm cũng như bộ nhớ tiêu tốn.

Đối với giải thuật Monte Carlo Tree Search nói riêng, đây là giải thuật theo nhóm chúng em là không phù hợp để giải quyết các bài toán một người chơi như Bloxorz. Sau khi cho chạy thử, đưa ra số liệu và phân tích cũng như mô phỏng trực quan cách giải thuật hoạt động với một số màn tiêu biểu, nhóm đưa ra một số nhận xét có thể tóm tắt như sau:

Về các giá trị trả về sau khi giải, vì đây là một giải thuật thuần ngẫu nhiên tìm kiếm, do đó các giá trị như đường đi tốt nhất, thời gian hoàn thành hay bộ nhớ sử dụng sẽ có biến động không hể nhỏ trong các giá trị ở các lượt chơi khác nhau.

Một số cải biến được thêm vào giải thuật bước đầu cho thấy sự cải thiện đáng kể ở hiệu suất, tuy vậy không cải biến nào cũng là cải biến ”toàn năng” có thể dùng để giải quyết hết tất cả các vấn đề trong việc giải quyết bài toán Bloxorz, một số ngoại lệ có thể dễ dàng được tìm thấy chỉ cho ta việc cải biến thậm chí còn làm tệ hơn hiệu năng của giải thuật. Bên cạnh đó cần nói thêm rằng đôi khi các phương pháp truyền thống lại tỏ ra hiệu quả hơn đối với các màn chơi cụ thể.

Cuối cùng, đối với mỗi màn chơi ta cần tìm kiếm các tham số cụ thể, áp dụng chính xác và hợp lý các cải biến cho giải thuật Monte Carlo Tree Search, từ đó đưa ra được giải thuật tốt nhất sao cho việc giải màn chơi được dễ dàng và hiệu quả nhất.

References

[1] Tahani Q. Alhassana, Shefaa S. Omara, Lamiaa A. Elrefaei. (2019). "Game of Bloxorz Solving Agent Using Informed and Uninformed Search Strategies". In *16th International Learning & Technology Conference 2019.*

[2] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, Jacek Mańdziuk. (2022). "Monte Carlo Tree Search: A Review of Recent Modifications and Applications".

[3] Maciej Świechowski, Jacek Mańdziuk, Yew Soon Ong. (2016). "Specialization of a UCT-Based General Game Playing Program to Single-Player Games". In *IEEE transactions on Computational Intelligence and AI in Games, Vol. 8, No. 3, September 2016.*