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

MÔN HỌC TRÍ TUỆ NHÂN TẠO

Bài tập lớn 1

Giải trò chơi Bloxorz bằng các giải thuật Tìm kiếm (Searching)

GVHD: Vương Bá Thịnh
SV: Nguyễn Thanh Hùng - 1511358
Phạm Quí Luận - 1511899
Trần Mạnh Hoàng - 1511150
Nguyễn Duy Đạo - 1510654
Phan Việt Đức - 1510809

TP. HỒ CHÍ MINH, THÁNG 5/2018

Mục lục

Danh sách hình vẽ

1 Tóm tắt

Trò chơi bloxorz là một trò chơi giải trí liên quan đến dịch chuyển một khối có kích thước 2x1x1 từ vị trí của nó rơi vào hố đích, được đảm bảo luôn luôn tìm thấy đường đi từ vị trí ban đầu tới vị trí gốc. Bài tập lớn này, nhóm chúng em sẽ hiện thực một công cụ trí tuệ nhân tạo để tìm ra lời giải cho trò chơi bằng 2 giải thuật cổ điển là Breadth First Search và Depth First Search và 1 giải thuật Heuristic là Best-First-Search, với hàm lượng giá được hiện thực bằng cách tính khoảng cách. Trong report này nhóm chúng em sẽ mô tả cách tiếp cận với bài toán, các định nghĩa cần thiết, và các giải thuật được hiện thực. Các đoạn code trong report được hiện thực bằng python, hoặc là mã giả. Cuối cùng là phần kết quả, cho thấy mức độ tiêu tốn thời gian và bộ nhớ, cùng theo đó là mức độ tốt của lời giải.

2 Nhận diện bài toán

2.1 Giới thiệu về game Bloxorz

Bloxorz là một trò chơi đố vui được phát triển bởi Damien Clarke và ra đời vào ngày 21/6/2007. Mục tiêu của trò chơi là di chuyển 1 block (khối hình chữ nhật) đi đến đích, trên bản đồ chứa các viên gạch. Có tổng cộng 33 stages để hoàn thành trò chơi. Người chơi di chuyển khối block bằng cách sử dụng các phím mũi tên, và không để bị rơi ra khỏi map (bản đồ).

Có các bridge (cầu) và các switch (công tắc) được đặt trong nhiều levels. Switches được kích hoạt khi bị block đè lên. Có 2 loại switch, Heavy X-shaped và Soft O-shaped. Soft switch được kích hoạt khi bất cứ phần nào của block nằm trên nó. Heavy switch được kích hoạt khi cả block nằm trên nó. Khi được kích hoạt, các switch sẽ có các động thái khác nhau. Một số sẽ chỉ bật bridge, một số sẽ chỉ đóng bridge, một số khác sẽ làm cả hai việc.

Những viên gạch màu da cam sẽ mong manh hơn các viên gạch còn lại. Nếu block đứng trên viên gạch đó, viên gạch sẽ rơi ra và block sẽ rơi khỏi map.

Cuối cùng, dạng switch thứ ba là teleport: ( ), khi block đứng trên nó sẽ bị tách ra thành hai block nhỏ hơn và chuyển đến hai vị trí khác nhau. Cả hai đều có thể được điều khiển bằng cách ấn nút space để chuyển sang block khác. Chúng sẽ được gắn lại nếu như block này nằm kế tiếp block kia. Các block nhỏ vẫn có thể kích hoạt O-shaped switch, nhưng chúng không đủ 'heavy' để kích hoạt X-shaped switch. Các block nhỏ không thể đi qua ô đích, chỉ có khối hoàn thiện mới có thể làm điều này.

2.2 Định nghĩa cho bài toán

2.2.1 Không gian trạng thái

Là tập hợp các trạng thái có thể của block trên map, không có phần nào của block rơi ra khỏi map. Một trạng thái được định nghĩa bao gồm các thuộc tính: tọa độ của block, board, rotation, parent.

Trong đó tọa độ của block được lưu bằng tupple (x,y) là tọa độ của điểm trên cùng - bên trái của block ở bất cứ trạng thái nào. Board là biến lưu trữ trạng thái của map, bước lưu trạng thái này có một điểm yếu rất lớn là hao tốn bộ nhớ, nhưng nó giải quyết rất nhiều vấn đề sẽ được đề cập ở phần dưới. Rotation là trạng thái tồn tại của block, có 4 trạng thái là $STANDING$,

$LAYING\_X$, $LAYING\_Y$, $SPLIT$. Trạng thái $STANDING$ được gán khi hai phần của block có cùng tọa độ, trạng thái $LAYING\_X$ được gán khi hai phần của block nằm trên cùng hàng, trạng thái $LAYING\_Y$ được gán khi hai phần của block nằm trên cùng cột, trạng thái $SPLIT$ được xác định khi block bị tách thành hai block nhỏ hơn, lúc này tuple (x, y) sẽ lưu vị trí của một block nhỏ, và tuple (x1, y1) sẽ lưu vị trí của block kia, (x1, y1) được khởi tạo khi trạng thái được khởi tạo, nhưng chỉ được xem xét đến khi rotation là $SPLIT$, rất may mắn cho nhóm khi đã chọn cách lưu trữ này để xử lý trường hợp gặp teleport switch, mặc dù trước đó nhóm từng ra quyết định sẽ tạo thêm trạng thái mới cho block con, điều này sẽ gây phiền toái khi quản lý. Phần tử Parent là tham chiếu tới trạng thái cha để chúng ta biết được trạng thái nào đã dẫn đến trạng thái hiện tại, điều này giúp giải quyết vấn đề tìm ra con đường đi đến chiến thắng.

Ý tưởng định nghĩa trạng thái như trên được hiện thực trong code như sau:

```
class Block:

    def __init__(self, x, y, rot, parent, board, x1=None,y1=None):
        self.x      = x
        self.y      = y
        self.rot    = rot
        self.parent = parent
        self.board  = copy.deepcopy(board)
        self.x1     = x1
        self.y1     = y1
```

2.2.2 Giới thiệu map file

Map file là một file text cung cấp thông tin về trạng thái khởi đầu, trạng thái kết thúc, tọa độ của các viên gạch trên map, vị trí các switch và bridge cũng như thông tin các bridge mà switch quản lý. Tùy theo mỗi bản đồ mà có tọa độ ban đầu khác nhau, rotation ban đầu là $STANDING$, parent là $None$, board là python 2-dimension list thể hiện trạng thái của bản đồ. Có tổng cộng 33 map file từ map01.txt đến map33.txt được đặt trong thư mục map/. Ví dụ map05.txt sẽ có nội dung như sau.

```
10 15 13 1
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
0 1 1 1 1 1 1 1 5 1 1 1 1 1 1
0 1 1 1 1 0 0 0 0 0 0 0 1 1 1
0 1 1 6 1 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 4 1 1 1 1 1 1 0 0
0 0 0 0 0 0 0 0 0 0 1 1 1 1 5
1 1 1 0 0 0 0 0 0 0 1 1 1 1 1
1 9 1 1 1 1 1 1 1 1 1 1 1 0 0
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
8 1 2 1 6 1 5
3 3 2 8 5 8 6
6 5 2 8 5 8 6
14 6 2 8 5 8 6
```

Trong đó `10 15 13 1` cho biết map có 15 cột, 10 hàng, vị trí bắt đầu ở tọa độ (x, y) = (13, 1). Trục x được gán theo hàng ngang chiều dương sang phải, trục y được gán cho cột dọc chiều dương quay xuống, tọa độ điểm trên cùng bên trái là (0,0).

Mười hàng tiếp theo cho biết bản đồ của trạng thái ban đầu. Được quy định như sau:

- 0: Là vị trí không có viên gạch, nếu bất cứ phần nào của block rơi vào đây thì trò chơi sẽ kết thúc với trạng thái thua.
- 1: Là vị trí của các viên gạch bình thường.
- 2: Là vị trí của các viên gạch màu da cam, chỉ cho phép 1 phần của block nằm trên nó, không cho phép block đứng trên nó.
- 3: Là vị trí của X-shaped switch
- 4: Là vị trí của O-shaped switch (chỉ cho phép đóng bridge)
- 5: Là vị trí của O-shaped switch
- 6: Là vị trí của O-shaped switch (chỉ cho phép mở bridge)
- 7: Là vị trí của teleport switch
- 8: Là vị trí của X-shaped switch (chỉ cho phép mở bridge)
- 9: Là vị trí chiến thắng

Những hàng còn lại sẽ cho biết ManaBoa (là danh sách tọa độ các switch, và các tọa độ được switch đó quản lý), theo format sau:

`(x, y) num <iX, iY>`

Ví dụ:

- `8 1 2 1 6 1 5` cho biết switch có tọa độ (1,8) quản lý 2 vị trí (1,6) và (1,5).

- `3 3 2 8 5 8 6` cho biết switch có tọa độ (3,3) quản lý 2 vị trí (8,5) và (8,6).
- `6 5 2 8 5 8 6` cho biết switch có tọa độ (5,6) quản lý 2 vị trí (8,5) và (8,6).

Lúc đầu vì lối tiếp cận giải quyết từng stage, nhóm chỉ nghĩ có khoảng dưới 10 trường hợp cần mã hóa thành các con số, nên O-shaped được chia thành 3 trường hợp 4,5,6 như trên, nhưng ở các level sau số trường hợp cần giải quyết nhiều hơn thế. Nếu mã hóa trường hợp bằng số có 2 chữ số thì sẽ dẫn đến map file khó đọc được. Vì vậy, nhóm gộp một số trường hợp lại như sau:

Thay vì format lúc ban đầu là `(x, y) num <iX, iY>`, thì khi chương trình đọc tới vị trí O-shaped (số 5), và X-shaped (số 3) sẽ được đọc theo format sau:

`(x, y) numToggle <xT, yT> numClose <xC, yC> numOpen <xO, yO>`

Trong đó, `numToggle` là số lượng các bridge bị đảo trạng thái mỗi lần switch bị kích hoạt, `numClose` là số lượng các bridge chỉ bị đóng và `numOpen` là số lượng các bridge chỉ bị mở. Ví dụ:

- `13 0 4 7 2 7 3 1 4 1 5 0 0` cho biết switch có tọa độ (0,13) có số lượng bridge toggle là 4, đó là (7,2) (7,3) (1,4) (1,5), và `numClose` = 0, `numOpen` = 0, nên không có danh sách cái tọa độ này.

2.2.3 Trạng thái ban đầu

Trạng thái ban đầu là trạng thái đầu tiên của block khi stage bắt đầu, nó được xác định từ dòng đầu tiên của map file được truyền vào chương trình. Rotation ban đầu là $STANDING$, parent là None.

2.2.4 Trạng thái mục tiêu

Trạng thái mục tiêu là trạng thái mà block đứng trên điểm đích. Nghĩa là tọa độ (x, y) = (xGoal, yGoal), rotation là $STANDING$.

Trạng thái mục tiêu được xác định trong code như sau:

```
def isGoal(block):

    # local definition
    x = block.x
    y = block.y
    rot = block.rot
    board = block.board

    # statements
    if rot == "STANDING" and board[y][x] == 9:
        return True
    else:
        return False
```

2.2.5 Các bước chuyển hợp lệ

Từ một trạng thái chúng ta có thể sinh ra các bước chuyển bằng cách di chuyển block lên trên, xuống dưới, sang trái hoặc sang phải. Ở các rotation $STANDING$, $LAYING\_X$, $LAYING\_Y$ thì sẽ sinh ra 4 bước chuyển. Nhưng ở rotation $SPLIT$ thì sinh ra 8 bước chuyển bởi vì mỗi block con sẽ sinh ra 4 bước chuyển.

Bước chuyển đi lên và đi xuống cho rotation $STANDING$, $LAYING\_X$, $LAYING\_Y$ được định nghĩa như sau:

```
def move_up(self):

    newBlock = Block(self.x, self.y, self.rot, self, self.board)

    if self.rot == "STANDING":
        newBlock.y -= 2
        newBlock.rot = "LAYING_Y"

    elif newBlock.rot == "LAYING_X":
        newBlock.y -= 1

    elif newBlock.rot == "LAYING_Y":
        newBlock.y -= 1
        newBlock.rot = "STANDING"

    return newBlock
```

```
def move_down(self):

    newBlock = Block(self.x, self.y, self.rot, self, self.board)

    if newBlock.rot == "STANDING":
        newBlock.y += 1
        newBlock.rot = "LAYING_Y"

    elif newBlock.rot == "LAYING_X":
        newBlock.y += 1

    elif newBlock.rot == "LAYING_Y":
        newBlock.y += 2
        newBlock.rot = "STANDING"

    return newBlock
```

Bước chuyển đi lên cho rotation $SPLIT$ được định nghĩa như sau:

```
def split_move_up(self):
    newBlock = Block(self.x, self.y, self.rot, self, self.board,
                                                self.x1, self.y1)
    newBlock.y -= 1
    return newBlock
```

```
def split1_move_up(self):
    newBlock = Block(self.x, self.y, self.rot, self, self.board,
                                               self.x1, self.y1)
    newBlock.y1 -= 1
    return newBlock
```

3 Các chiến lược tìm kiếm để giải quyết bài toán

Có ba chiến lược được sử dụng để giải quyết bài toán, mặc dù có sự khác nhau đáng kể về thời gian, bộ nhớ tiêu tốn. Nhưng cơ bản vẫn đi theo một lối phát triển giống nhau. Ban đầu, nhóm hiện thực giải thuật Depth-First-Search để giải từng stage một, quá trình này mất khá lâu thời gian bởi vì sau mỗi stage lại có thêm sự kiện mới, và có lúc phải thay đổi cả cấu trúc dữ liệu để lưu trạng thái. Sau khi hoàn thành 33 stages bằng giải thuật Depth-First-Search thì việc chuyển sang giải thuật Breadth-First-Search chỉ tốn một thời gian rất ngắn. Sau cùng là giải thuật Heuristic, ban đầu nhóm chọn lựa giải thuật Hill Climbing để làm nhưng vì nhận thấy nó khó khăn hơn Best-First-Search ở chỗ không biết lựa chọn hàm evaluation như thế nào là hợp lý. Nên nhóm đã chuyển sang hiện thực giải thuật Best-First-Search với hàm evaluation là tính khoảng cách từ trạng thái hiện tại tới trạng thái đích. Sau khi hiện thực và xem kết quả, nhóm nhận thấy có một số ưu điểm và nhược điểm đáng kể, mà sẽ được nêu ra chi tiết trong phần hiện thực

3.1 Giải thuật Depth-First-Search

Vì nhóm đã dùng giải thuật Depth-First-Search để tiếp cận với bài toán ngay từ đầu, nên trong phần này, nhóm sẽ nêu ra lối tiếp cận để giải quyết theo từng stage, có đính kèm code trong file ai.py và giải thích.

Xem xét đoạn mã giả hiện thực giải thuật Depth-First-Search sau đây:

```
# solve DFS python pseudo code
def DFS(block):

    Stack.put(block)

    while Stack:
        current = Stack.pop()
        if isGoal(current):
            return SUCCESS
        else:
            Stack.put(current.all_possible_move())
    return False
```

Nhóm đã dùng vòng lặp while để hiện thực giải thuật thay vì dùng phương pháp đệ quy, ưu điểm code việc hiện thực bằng vòng lặp là dễ quản lý code và rủi ro, việc sửa lỗi lập trình cũng trở nên dễ dàng hơn rất nhiều. Giải thuật bắt đầu từ việc thêm trạng thái start vào Stack và chạy vòng lặp. Trong thân vòng lặp chúng ta lấy ra phần từ đầu tiên trong Stack và xem xét nó liệu có phải là trạng thái đích, chương trình sẽ dừng lại nếu điều kiện này đúng, nếu điều kiện này sai, chương trình sẽ duyệt lần lượt các bước chuyển hợp lý có thể của current, và đưa toàn bộ bước chuyển này vào Stack và lặp lần tiếp theo.

Dưới đây là đoạn mã hiện thực giải thuật bằng code python, có thêm vào một số thay đổi. Việc lưu các trạng thái đã đi qua vào biến `passState` giúp cho chúng ta tránh được việc chương trình sẽ lặp đi lặp lại các trạng thái đã duyệt. Mặc dù tiêu tốn bộ nhớ và thời gian để so sánh mỗi lần duyệt, nhưng nó tối ưu hơn về số bước chuyển hợp lý, tránh mắc bẫy di chuyển quá nhiều lần không hợp lý. Biến `virtualStep` là biến để lưu tổng số trạng thái mà chương trình thực sự phải duyệt qua để đi đến đích, mục tiêu là để tính toán bộ nhớ tiêu tốn dùng cho giải thuật, và độ phức tạp của giải thuật khi tìm ra lời giải, số lượng `virtualStep` được ghi lại ở cột `Time` ở Hình 1 Bảng số liệu về thời gian và sự tiêu tốn bộ nhớ của từng giải thuật đối với từng stage.

```
# solve DFS
def DFS(block):

    # local definitions
    board = block.board
    Stack = []
    Stack.append(block)
    passState.append(block)
    virtualStep = 0

    # statements
    while Stack:
        current = Stack.pop()

        if isGoal(current):
            printSuccessRoad(current)
            print("COMSUME", virtualStep, "VIRTUAL STEP")
            print("SUCCESS")
            return True
        else:
            if current.rot != "SPLIT":
                virtualStep += 4
                move(Stack,current.move_up(), "up")
                move(Stack,current.move_right(), "right")
                move(Stack,current.move_down(), "down")
                move(Stack,current.move_left(), "left")
            else:
                virtualStep += 8
                move(Stack,current.split_move_left(), "left0")
                move(Stack,current.split_move_right(), "right0")
                move(Stack,current.split_move_up(), "up0")
                move(Stack,current.split_move_down(), "down0")

                move(Stack,current.split1_move_left(), "left1")
                move(Stack,current.split1_move_right(), "right1")
                move(Stack,current.split1_move_up(), "up1")
                move(Stack,current.split1_move_down(), "down1")
    return False
```

Trong thân hàm move sẽ xem xét liệu bước chuyển có hợp lý (thông qua hàm `isValidBlock`) và xét xem nó đã được duyệt qua trước đó chưa (thông qua hàm `isVisited`). Nếu chưa thì sẽ tiến hành duyệt trạng thái này. Hàm move được định nghĩa như sau:

```
def move(Stack, block, flag):
    if isValidBlock(block):
        if isVisited(block):
            return None
        Stack.append(block)
        passState.append(block)
        return True
    return False
```

Các công đoạn xử lý các switches được hiện thực trong thân hàm `isValidBlock`. Mã giả của nó như sau (vì thân hàm của nó quá dài nên không được đưa vào report).

```
def isValidBlock(block):
    if isFloor(block):
            if isSwitch(block):
                #handle
        return True
    return False
```

Một điểm đáng chú ý nữa là hàm `isVisited` của nhóm. Ban đầu, một trạng thái chỉ lưu giữ tọa độ và rotation, để xem xét một trạng thái đã được duyệt qua chưa, thì chỉ cần so sánh các tọa độ và rotation. Nhưng ở các stage phía sau (stage 8), thì cần phải đi ngược lại *trạng thái đã duyệt qua rồi*. Bài toán lúc này không thể giải quyết được. Nhóm đã tìm ra nhiều cách giải quyết như là cho phép chương trình duyệt qua những node đã được duyệt, nhưng nó dẫn đến thất bại khi chương trình mắc bẫy vòng lặp vô tận, để giải quyết vòng lặp vô tận thì có thể ràng buộc chiều sâu của cây Depth-First-Search, nhưng như thế thì không còn ý nghĩa của giải thuật nữa. Một lối tiếp cận khác là lưu map vào trong từng stage, ưu điểm của cách này là giải quyết được vấn đề ở trên, và mỗi trạng thái của nó bây giờ là là unique và bắt buộc không được lặp lại. Nhược điểm lớn nhất và đáng kể tâm nhất ở cách giải quyết này là nó tiêu tốn một lượng lớn bộ nhớ. Cứ mỗi trạng thái, chúng ta phải lưu một ma trận map. Và tiêu tốn cả thời gian khi chúng ta kiểm tra xem trạng thái này đã duyệt qua chưa. Mặc dù vậy, nhược điểm này đã được chấp nhận để giải thuật hoàn thiện.

3.2 Giải thuật Breadth-First-Search

Sau khi hiện thực thành công giải thuật Depth-First-Search, thì việc chuyển qua giải thuật Breadth-First-Search hết sức dễ dàng bằng cách thay vì duyệt cây bằng Stack, thì chúng ta sẽ dùng Queue. Chúng ta xem xét đoạn mã giả sau:

```
# solve BFS python pseudo code
def BFS(block):

    Queue.enqueue(block)

    while Queue:
        current = Queue.dequeue()
        if isGoal(current):
            return SUCCESS
        else:
            Queue.enqueue(current.all_possible_move())
    return False
```

3.3 Giải thuật Best-First-Search

Chúng ta xem xét đoạn mã giả sau:

```
# solve Best-first-search python pseudo code
def BEST(block):

    BestQueue = PriorityQueue()
    startEval = evalFunction(block)
    BestQueue.enqueue((startEval, block))

    # until priority queue is empty
    while BestQueue.not_empty:
        item   = BestQueue.dequeue()  # item = (block, distance)
        iDista = item[0]
        iBlock = item[1]

        if isGoal(iBlock):
            return SUCCESS

        BestQueue.enqueue(iBlock.all_possible_state())
```

Trong đó hàm `evalFunction` là hàm lượng giá của giải thuật, một trạng thái bất kỳ được lượng giá bằng cách đo khoảng cách từ nó đến trạng thái đích, nhóm sử dụng công thức Pythagorean: $distance = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ để tính khoảng cách trong hàm lượng giá, vì để tránh tốn kém thời gian khi gọi hàm lấy căn nên nhóm đã cho trả về $distance^2$ mã giả như sau:

```
def evalFunction(block):
    # we read board to get goal
    goal = block.getGoal()
    curr = block.getCurr()
    return distance(goal,curr)
```

4 Số liệu về thời gian thực thi và sự tiêu tốn bộ nhớ

Mã nguồn được chạy để kiểm tra thời gian thực thi được thực hiện trên máy ảo VMWARE chạy hệ điều hành linux được cài đặt trên máy tính Windows 7 64-bit, Processor Intel(R) Core(TM) i3-2328M CPU 2.20GHz. RAM have 5.89GB usable.

4.1 Bảng số liệu cho các giải thuật

Bảng số liệu về thời gian và sự tiêu tốn bộ nhớ cho từng giải thuật được trình bày như sau:

Stage	Depth-First-Search			Breadth-First-Search			Best-First-Search		
	Time	Step	Memory	Time	Step	Memory	Time	Step	Memory
01	0.016s	30	156	0.049s	8	296	0.012s	10	296
02	0.017s	108	848	0.020s	18	1060	0.009s	32	1436
03	0.012s	55	340	0.017s	17	428	0.027s	26	452
04	0.012s	35	360	0.020s	29	276	0.011s	29	304
05	0.009s	92	444	0.005s	34	1264	0.009s	42	1228
06	0.005s	36	176	0.005s	36	448	0.004s	40	380
07	0.025s	55	472	0.016s	45	716	0.017s	51	708
08	0.017s	39	460	0.037s	11	4228	0.020s	14	9300
09	0.017s	269	2644	0.042s	25	5596	0.017s	53	4700
10	0.024s	954	8848	0.165s	58	45712	0.044s	199	29352
11	0.012s	48	668	0.008s	48	744	0.025s	48	632
12	0.017s	130	1144	0.028s	66	1296	0.041s	68	1220
13	0.008s	54	596	0.008s	47	532	0.016s	51	472
14	0.008s	114	1276	0.024s	68	1840	0.028s	72	1252
15	0.005s	176	1600	0.240s	58	33224	2.941s	71	119048
16	0.009s	57	3032	0.012s	25	684	0.053s	25	3240
17	0.021s	338	2892	0.040s	107	3932	0.185s	158	3960
18	0.009s	161	1528	0.036s	86	2392	0.070s	109	2356
19	0.009s	72	596	0.016s	68	892	0.089s	68	880
20	0.028s	260	5868	0.171s	57	32280	6.392s	111	158344
21	0.025s	73	612	0.024s	72	936	0.119s	73	1232
22	0.017s	150	1432	0.022s	66	1304	0.021s	81	1376
23	0.028s	413	12632	0.097s	76	29560	1.822s	100	28616
24	0.008s	59	636	0.036s	58	2184	0.042s	58	1712
25	0.012s	95	708	0.009s	56	1896	0.024s	56	1556
26	0.084s	1159	34908	0.317s	105	75188	0.482s	126	53124
27	0.016s	74	356	0.024s	72	1256	0.020s	74	936
28	0.084s	781	31812	0.345s	101	52520	0.076s	131	11280
29	0.008s	132	5960	0.025s	105	7248	0.040s	121	4448
30	0.032s	115	1112	0.024s	115	2224	0.012s	115	1448
31	0.008s	221	2256	0.024s	92	3504	0.045s	133	4372
32	0.016s	204	1220	0.028s	130	2520	0.025s	130	2108
33	0.012s	82	2784	0.057s	66	2952	0.020s	77	2768

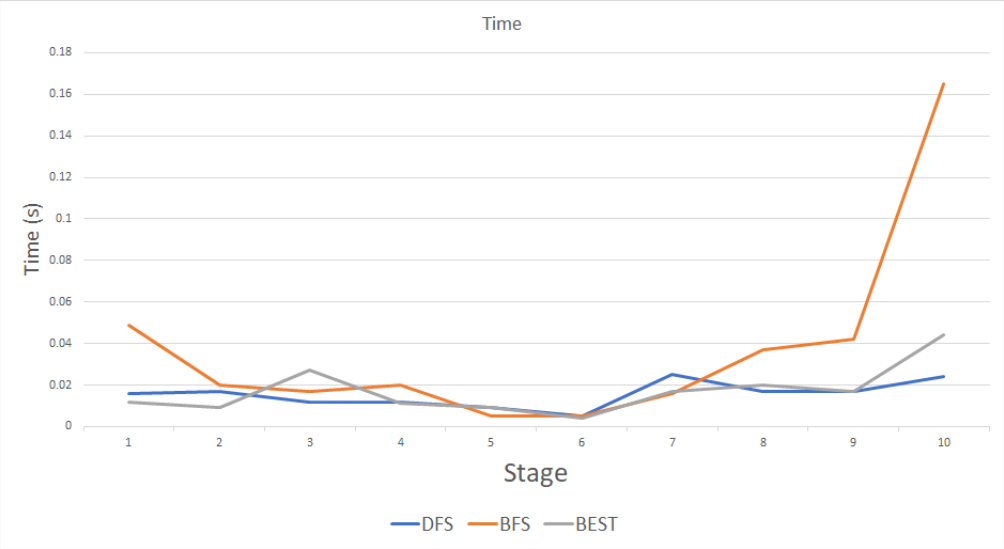
Hình 1: Bảng số liệu về thời gian và sự tiêu tốn bộ nhớ

Một số giải thích:

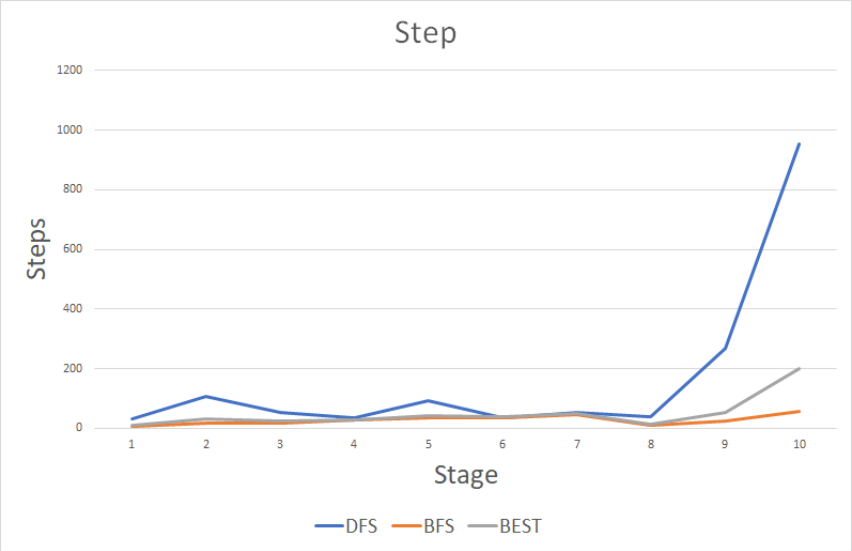
- Cột Time là thời gian CPU bỏ ra trong kernel cho tiến trình hiện tại, được trích xuất từ hàng sys trong lệnh time. Ví dụ: `"time python ai.py 01 DFS"` thì thời gian sẽ khoảng 0.016s.
- Step là số bước cần thiết do giải thuật tìm thấy để đi từ vị trí bắt đầu đến trạng thái đích.
- Memory là tổng số node mà giải thuật phải duyệt qua trước khi tìm được trạng thái đích, bộ nhớ cần dùng để sử dụng cho giải thuật được tính theo công thức: $AmountOfMem = Memory * 200bytes$. Trong đó 200 bytes là con số ước tính dung lượng để lưu trữ một trạng thái bao gồm tọa độ, rotation, map, và con trỏ với phần tử cha.

4.2 Các đồ thị trực quan so sánh các giải thuật

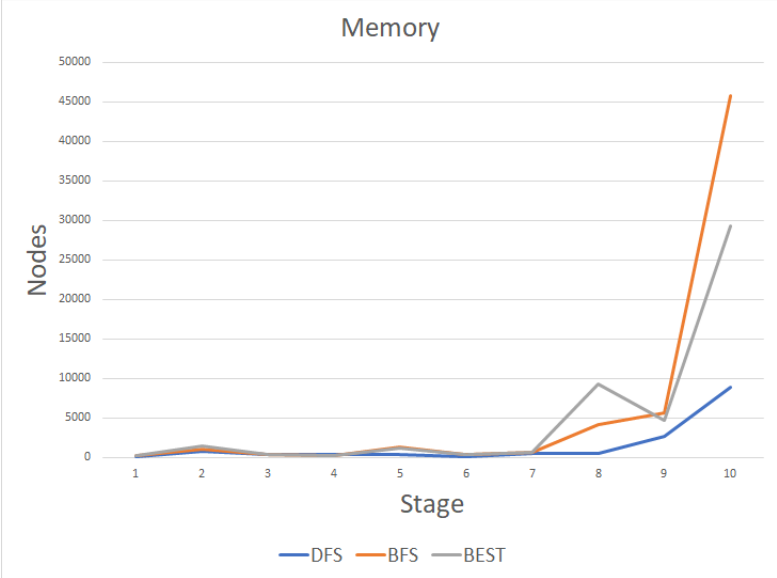
Dưới đây là một số đồ thị trực quan cho thấy sự khác nhau của các giải thuật trong 10 stage đầu tiên. Ở Hình 2 là đồ thị cho thấy sự khác nhau về sự tiêu tốn thời gian. Hình 3 so sánh số lượng bước chuyển tới trạng thái đích mà các giải thuật tìm thấy. Hình 4 so sánh sự tiêu thụ bộ nhớ của từng giải thuật.



Hình 2: So sánh thời gian chạy của 3 giải thuật trong 10 stages đầu tiên



Hình 3: So sánh số lượng step (mức độ thông minh) của 3 giải thuật trong 10 stages đầu tiên



Hình 4: So sánh bộ nhớ sử dụng của 3 giải thuật trong 10 stages đầu tiên

4.3 Đánh giá

Vì sự khác nhau giữa các stage nên phần đánh giá này không phải là tuyệt đối so với các stage.

	Thời gian tiêu tốn	Kết quả	Số trạng thái phải duyệt
Breadth-First-Search	nhiều nhất	tốt nhất	nhiều nhất
Best-First-Search	trung bình	trung bình	trung bình
Depth-First-Search	ít nhất	kém nhất	ít nhất

Hình 5: Bảng so sánh đánh giá các giải thuật

Trên lý thuyết chúng ta có thể thấy được BFS tìm kiếm ra kết quả gần giống với con người nhất, vì kết quả nó tìm thấy bằng cách duyệt theo chiều rộng luôn luôn có khoảng cách đến trạng thái ban đầu ngắn nhất. Ngược lại, giải thuật DFS tìm kiếm ra kết quả (gần như) là tệ nhất. Giải thuật Heuristic nằm ở mức độ trung bình ở giữa hai giải thuật này, nhưng có một số stage mà Heuristic không hiệu quả, ví dụ như stage 20, Heuristic phải duyệt qua gần 160 nghìn trạng thái mới tìm thấy kết quả, trong khi BFS chỉ cần duyệt khoảng 32 nghìn trạng thái và DFS chỉ cần duyệt 5868 trạng thái, tuy vậy số bước để đi đến thành công của nó là 111 bước, tốt hơn nhiều so với DFS là 260 bước.

5 Kết luận

Thông qua assignment này giúp chúng em hiểu rõ hơn về cách định nghĩa không gian trạng thái cho một bài toán, hiểu được cách thực hiện của từng giải thuật tìm kiếm như BFS, DFS, Best First Search và hiện thực chúng thông qua ngôn ngữ Python. Trong quá trình hiện thực, chúng em luyện tập được cách giải quyết vấn đề, nâng cao tư duy giải thuật, sử dụng nhiều cấu trúc giải thuật khác nhau giúp cho việc hiện thực trở nên dể dàng hơn.

Tài liệu

1. Solving Logic Puzzles using Model Checking in LTSmin Kamies, Bram University of Twente P.O. Box 217, 7500AE Enschede The Netherlands b.kamies@student.utwente.nl
2. SINGLE AGENT SHORTEST PATH DETERMINATION IN THE GAME OF BLOX-ORZ Nwulu E. A., Ndukwe I. G. Department of Computer Science, University of Jos Jos, Nigeria