ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

BÁO CÁO BÀI TẬP LỚN 1
NHẬP MÔN TRÍ TUỆ NHÂN TẠO (CO3061)

HIỆN THỰC GIẢI THUẬT TÌM KIẾM LOGIC PUZZLES

GV hướng dẫn: Vương Bá Thịnh
SV thực hiện: Nguyễn Quốc Duy 2120011
Nguyễn Tiến Phát 2011797
Võ Tấn Hưng 2113623
Lê Duy Anh 2112762

Ho Chi Minh City, July, 2023

Contents

1 Giới thiệu đề tài

1.1 Giới thiệu về bài toán Logic Puzzles

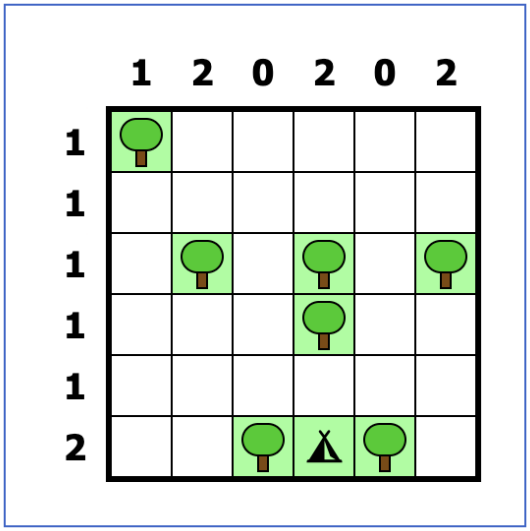
Logic Puzzles là thể loại game đòi hỏi người chơi suy luận, phân tích và giải quyết những câu đố thách thức. Những tựa game thuộc thể loại này đòi hỏi chúng ta phải "nghĩ logic", tức là sử dụng trí tuệ và logic để tìm ra các giải pháp đúng đắn. Thông thường, những trò chơi này có các quy tắc rõ ràng và mục tiêu cụ thể để giải quyết.

Một số tựa game nổi tiếng thuộc thể loại này có thể kể đến như: Sudoku, Nonograms, Pipes,... Trong đề tài lần này, nhóm sẽ sử dụng và hiện thực các giải thuật tìm kiếm trên 2 tựa game thuộc thể loại Logic Puzzles này là: **Tents** và **Light Up**.

1.1.1 Tents

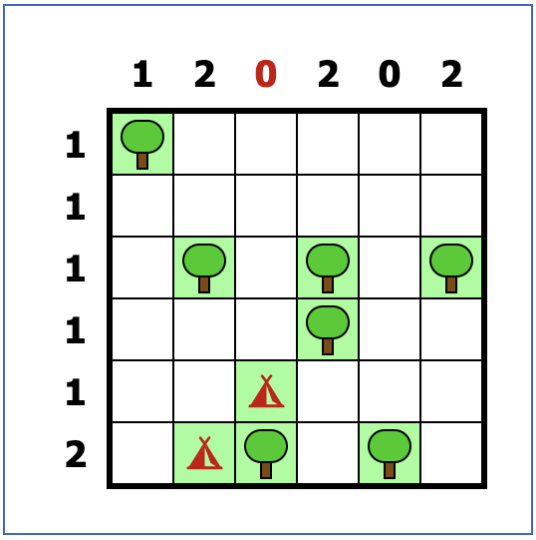
Tents là tựa game thuộc thể loại Logic Puzzles. Mục tiêu của trò chơi là tìm ra những vị trí thích hợp trên bản đồ để đặt lều vào mà không vi phạm luật chơi. Luật chơi của Tents như sau:

- Với mỗi cây trên bản đồ, chúng ta phải đặt lều sao cho luôn có một (và chỉ một) lều đứng cạnh nó theo chiều ngang và dọc, và các cây không thể sử dụng chung lều.

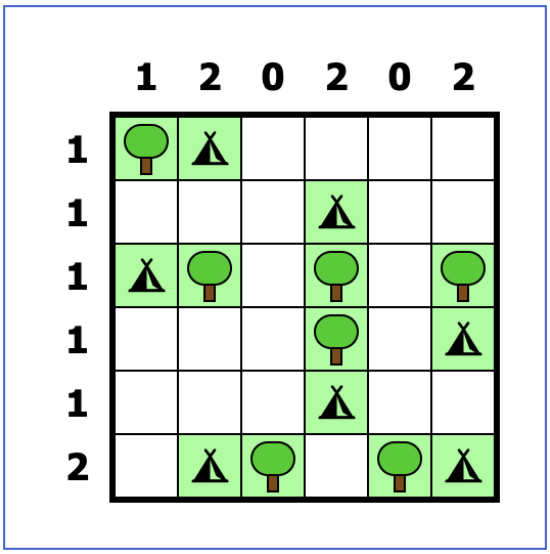


Hình 1: Vi phạm lỗi: Tồn tại một cây không có lều đứng bên cạnh

- Các lều không được đặt cạnh nhau trong phạm vi 1 ô, kể cả theo đường chéo.
- Mỗi hàng/cột trong ma trận sẽ được giới hạn số lều được đặt trong hàng/cột đó.



Hình 2: Vi phạm lỗi: Hai lều được đặt chéo nhau và ở cột thứ 3 vượt quá số lều cho phép

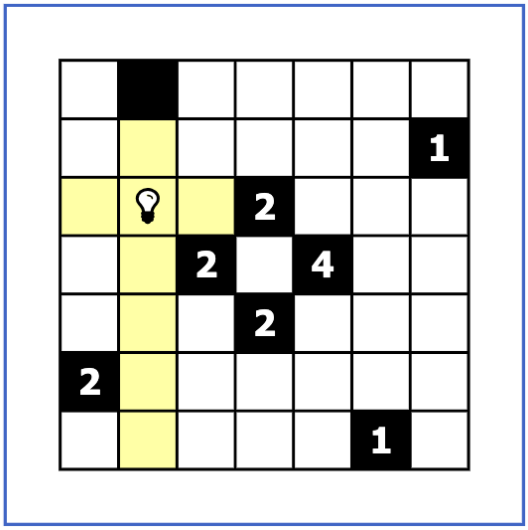


Hình 3: Một màn chơi Tents có kết quả hợp lệ

1.1.2 Light Up

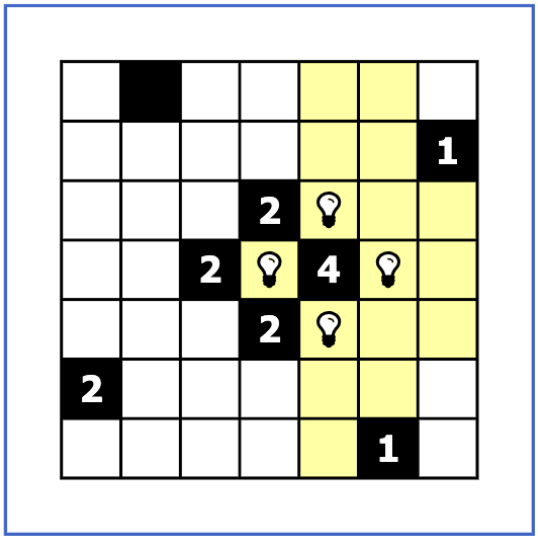
Mục tiêu của trò chơi **Light Up** là đặt các bóng đèn trên lưới ô vuông theo luật sao cho tất cả các ô vuông màu trắng đều được thắp sáng mà. Luật chơi cơ bản của Light Up như sau:

- Bóng đèn được đặt vào các ô vuông màu trắng và chiếu sáng theo 4 hướng (ngang và dọc) cho đến khi gặp cạnh bản đồ hoặc chướng ngại vật (ô vuông màu đen).



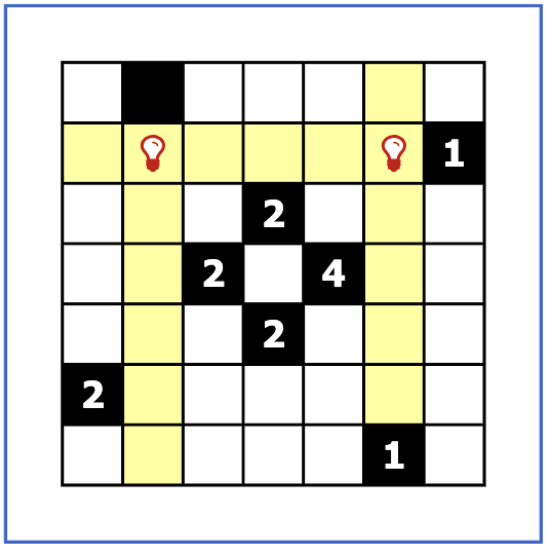
Hình 4: Bóng đèn sẽ chiếu sáng theo 4 hướng, cho đến khi gặp vật cản

- Số trong các ô vuông màu đen chỉ ra có chính xác bao nhiêu bóng đèn được đặt bên cạnh ô đó.

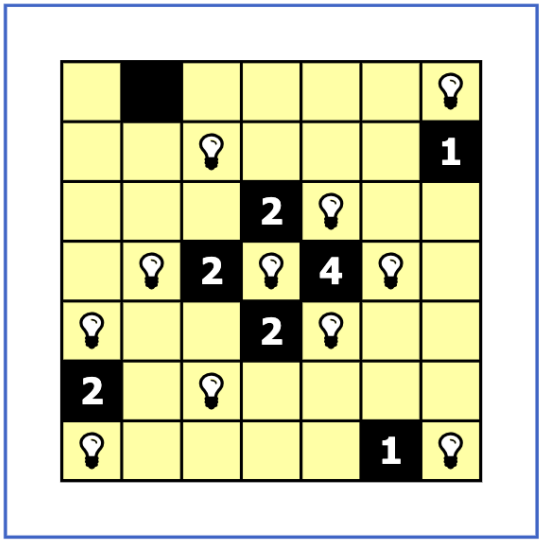


Hình 5: Phải có 4 bóng đèn được đặt bên cạnh chướng ngại vật có số 4

- Không được phép đặt bóng đèn vào ô đã được chiếu sáng trước đó, có nghĩa là trong cùng một hàng/cột, các bóng đèn không thể tồn tại cùng nhau trừ khi có chướng ngại vật ở giữa chúng.



Hình 6: Hai bóng đèn được đặt cùng một hàng mà không có chướng ngại vật giữa chúng



Hình 7: Mô tả một bàn chơi Light Up có kết quả hợp lệ

1.2 Phân tích yêu cầu

Sinh viên được yêu cầu hiện thực **2** bài toán Logic Puzzles dựa vào 2 giải thuật tìm kiếm:

- Blind Search: Depth-Fisrt-Search (DFS) hoặc Breadth-First-Search (BrFS)
- Heuristic Search: A* (Asterisk) Algorithm hoặc Hill Climbing...

Ngoài ra còn có các yêu cầu khác như sau:

- Sử dụng ngôn ngữ Python 3.

- Tạo ra các input phù hợp cho game.
- Hiện thực có tính năng demo lời giải trực quan (step-by-step) như game.
- Báo cáo nêu được quá trình tìm hiểu và hiện thực bài toán.
- Trình bày và giải thích được bảng số liệu về thời gian và sự tiêu tốn bộ nhớ của từng giải thuật đối từng input khác nhau.

2 Các thuật toán được sử dụng

2.1 Breadth First Search (BFS) - Tìm kiếm theo chiều rộng

Giải thuật tìm kiếm theo chiều rộng (Breadth First Search - viết tắt là BFS) duyệt qua một đồ thị theo chiều rộng và sử dụng hàng đợi (queue) ghi nhớ đỉnh liền kề để bắt đầu việc tìm kiếm khi không gặp được đỉnh liền kề trong bất kỳ vòng lặp nào.

Pseudocode:

```
% Input: G is the graph and s is the source node
BFS (G, s)
    let Q be queue
    Q.enqueue(s)
    % Inserting s in queue until all its neighbour vertices are marked.
    mark s as visited
    while (Q is not empty)
        % Removing that vertex from queue,whose neighbour will be visited now
        v = Q.dequeue()
        % Processing all the neighbours of v
        for all neighbours w of v in Graph G
            if w is not visited
                % Stores w in Q to further visit its neighbour
                Q.enqueue(w)
                mark w as visited
```

Giải thuật này tuân theo các bước sau:

1. Khởi tạo hàng đợi (queue) và danh sách đỉnh đã được thăm:
   - Đặt đỉnh bắt đầu vào hàng đợi.
   - Đánh dấu đỉnh bắt đầu là đã thăm.
2. Lặp lại cho đến khi hàng đợi queue trống: Lấy đỉnh hiện tại ở đầu hàng đợi ra.
3. Duyệt tất cả các đỉnh liền kề với đỉnh hiện tại, nếu đỉnh liền kề chưa được thăm:
   - Đánh dấu đỉnh liền kề là đã thăm.
   - Thêm đỉnh liền kề vào cuối hàng đợi.
4. Lặp lại bước 2 và 3 cho đến khi không còn đỉnh nào trong hàng đợi.

2.2 Depth First Search (DFS) - Tìm kiếm theo chiều sâu

Giải thuật tìm kiếm theo chiều sâu (Depth First Search – viết tắt là DFS), còn được gọi là giải thuật tìm kiếm ưu tiên chiều sâu, là giải thuật duyệt hoặc tìm kiếm trên một cây hoặc một đồ thị và sử dụng stack (ngăn xếp) để ghi nhớ đỉnh liền kề để bắt đầu việc tìm kiếm khi không gặp được đỉnh liền kề trong bất kỳ vòng lặp nào. Giải thuật tiếp tục cho tới khi gặp được đỉnh cần tìm hoặc tới một nút không có con. Khi đó giải thuật quay lui về đỉnh vừa mới tìm kiếm ở bước trước.

Pseudocode:

```
% Input: G is graph and s is source vertex
DFS(G, s)
    let S be stack
    % Inserting s in stack
    S.push(s)
    mark s as visited
    while (S is not empty):
        % Pop a vertex from stack to visit next
        v = S.top()
        S.pop()
        % Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
        if w is not visited :
            S.push(w)
            mark w as visited
```

Giải thuật này tuân theo các bước sau:

1. Khởi tạo ngăn xếp (stack) và danh sách đỉnh đã được thăm:
   - Đặt đỉnh bắt đầu vào ngăn xếp.
   - Đánh dấu đỉnh bắt đầu là đã thăm.
2. Lặp cho đến khi ngăn xếp (stack) trống: Lấy đỉnh hiện tại từ đỉnh trên cùng của ngăn xếp.
3. Duyệt tất cả các đỉnh kề với đỉnh hiện tại, nếu đỉnh kề chưa được thăm:
   - Đánh dấu đỉnh kề là đã thăm.
   - Đặt đỉnh kề vào đỉnh trên cùng của ngăn xếp.
4. Nếu không còn đỉnh kề chưa được thăm, lấy đỉnh khỏi ngăn xếp (điều này đồng nghĩa với việc di chuyển lên trên trong đồ thị).
5. Lặp lại các bước 2, 3 và 4 cho đến khi không còn đỉnh nào trong ngăn xếp.

2.3 A* (Asterisk) Algorithm

Thuật toán A* (A-Star) là một thuật toán tìm kiếm đường đi trong đồ thị hoặc trạng thái để tìm đường đi tối ưu từ điểm bắt đầu tới điểm đích, bằng việc kết hợp giữa 2 giải thuật Uniform Cost Search và Best First Search để tăng hiệu quả tìm kiếm.

A* sử dụng hàm f, là sự kết hợp giữa một hàm biểu diễn cho **chi phí đã di chuyển**

- g và một hàm h **để ước lượng chi phí còn lại từ vị trí (trạng thái) hiện tại tới điểm (trạng thái) kết thúc** ($f = g + h$), để tính giá trị f ở mỗi trạng thái và chọn trạng thái có giá trị hàm f nhỏ nhất để tiếp tục xử lý.

Các thành phần chính của thuật toán A* bao gồm:

- Danh sách Open (Open list): Là danh sách các trạng thái đang xem xét và chưa được xác định tốt nhất. Thường sử dụng **hàng đợi ưu tiên (priority queue)** để quản lý Open list, với thứ tự ưu tiên dựa trên giá trị $f = g + h$ (tổng chi phí đã di chuyển và ước lượng chi phí còn lại).
- Danh sách Closed (Closed list): Là danh sách các trạng thái đã được xem xét và được đánh giá tốt nhất (có giá trị f nhỏ nhất). Trạng thái được chuyển từ Open list sang Closed list sau khi đã xác định được giá trị tốt nhất cho nút đó.
- Hàm heuristic (h): Là một hàm ước lượng chi phí từ một trạng thái tới trạng thái kết thúc. Hàm heuristic không được đánh giá quá cao hoặc thấp quá mức, cần đảm bảo tính tối ưu và chắc chắn của thuật toán.

Thuật toán A* hoạt động như sau:

1. Khởi tạo Open list và Closed list. Đặt điểm bắt đầu vào Open list:
   - Khởi tạo g = 0 (chi phí đã di chuyển từ trạng thái bắt đầu tới trạng thái hiện tại).
   - Tính h (ước lượng chi phí còn lại từ trạng thái hiện tại tới trạng thái đích) cho điểm bắt đầu.
2. Lặp cho đến khi Open list trống hoặc đã tìm thấy đường đi tới trạng thái đích:
   a) Lấy trạng thái hiện tại từ Open list với giá trị f nhỏ nhất.
   b) Nếu trạng thái hiện tại là trạng thái đích, thuật toán kết thúc và trả về đường đi tối ưu.
   c) Di chuyển trạng thái hiện tại từ Open list sang Closed list.
   d) Duyệt các trạng thái kề với trạng thái hiện tại:
      - Nếu trạng thái kề không nằm trong Closed list, đánh giá và thêm nó vào Open list.
      - Nếu trạng thái kề đã nằm trong Open list, kiểm tra và cập nhật giá trị f nếu có giá trị f tốt hơn.
3. Danh sách Closed (Closed list): Là danh sách các trạng thái đã được xem xét và được đánh giá tốt nhất (có giá trị f nhỏ nhất). Trạng thái được chuyển từ Open list sang Closed list sau khi đã xác định được giá trị tốt nhất cho nút đó.
4. Hàm heuristic (h): Là một hàm ước lượng chi phí từ một trạng thái tới trạng thái kết thúc. Hàm heuristic không được đánh giá quá cao hoặc thấp quá mức, cần đảm bảo tính tối ưu và chắc chắn của thuật toán.

2.4 Simulated Annealing Algorithm

Thuật toán Simulated Annealing (SA) là một thuật toán tối ưu hóa toàn cục (global optimization) được lấy cảm hứng từ quá trình làm mát từng bước (annealing) trong metalurgy (kỹ thuật kim loại). Nó được sử dụng để tìm kiếm giải pháp tối ưu trong không gian tìm kiếm lớn.

Thuật toán bắt đầu với một giải pháp ngẫu nhiên và sau đó dựa vào một hàm mục tiêu (objective function) để đánh giá chất lượng của giải pháp hiện tại. Tiếp theo, nó thử nghiệm các giải pháp mới bằng cách thay thế, hoán đổi, hoặc điều chỉnh các yếu tố trong giải pháp hiện tại để tạo ra các giải pháp mới. SA quyết định có chấp nhận giải pháp mới hay không dựa trên hai yếu tố:

- **Sự khác biệt về giá trị mục tiêu giữa giải pháp mới và giải pháp hiện tại**: Nếu giá trị mục tiêu của giải pháp mới tốt hơn giải pháp hiện tại, giải pháp mới sẽ được chấp nhận.
- **Nhiệt độ của hệ thống**: Nhiệt độ giảm dần theo thời gian, giảm sự xác suất để chấp nhận một giải pháp mới có giá trị mục tiêu kém hơn giải pháp hiện tại. Điều này giúp thuật toán thoát khỏi các điểm tối ưu cục bộ và có cơ hội khám phá các giải pháp tối ưu mới.

Khi nhiệt độ giảm xuống, SA dần dần hội tụ về một giải pháp tối ưu hoặc gần tối ưu. Thuật toán thường được sử dụng để giải các vấn đề tối ưu hóa không liên tục và không lồi, trong đó không có phương pháp tối ưu nào có thể đảm bảo tìm được giải pháp toàn cục.

3 Hiện thực bài toán

3.1 Tents

3.1.1 Định nghĩa trạng thái (States) và các luật di chuyển hợp lệ (Legal Moves)

A. State:

Trạng thái bài toán (State) được mô tả trong file **state.py** bằng class **State**, các thành phần của trạng thái bao gồm:

- Kích thước *size* của bản đồ.
- Ma trận vuông *matrix*: có kích thước $size * size$, biểu diễn cho bản đồ trò chơi. Các phần tử của ma trận có thể là 0, 1 hoặc 2 (lần lượt đại diện cho ô trống, cây và lều)
- Hai danh sách *row* và *col*: biểu diễn cho số lều được đặt ở mỗi hàng/cột.
- State trước đó *prev*: dùng để lưu trạng thái ngay trước trạng thái được khởi tạo. Sau khi đạt được trạng thái kết thúc, *prev* được sử dụng để truy lại những trạng thái đã đi qua.

Và một số attribute khác được dùng trong thuật toán A* phục vụ cho hàm cost function f

- Số lều đã được đặt đúng vị trí: *tents_placed*.
- Số ô còn trống trên bản đồ: *cells_empty*.
- Số cây chưa được che phủ (chưa được đặt lều bên cạnh): *trees_uncovered*.

- Giá trị hàm f của trạng thái hiện tại: *cost*

```
class State:
def __init__(self, size: int, matrix: list[list], row: list, col: list, prev = None):
    self.size = size
    self.matrix = copy.deepcopy(matrix)
    self.row = row
    self.col = col
    self.prev = prev
    # For A-Star Algorithm
    self.tents_placed = len(getAllTentPosition(self))
    self.trees_uncovered = getUncoveredTrees(self)
    self.cells_empty = getEmptyCells(self)
    self.cost = cost_function(self)
```

B. Initial State:

Trạng thái khởi động: các phần tử của ma trận biểu diễn cho bản đồ sẽ chỉ chứa các phần tử 0 và 1.

C. Goal State:

Trạng thái kết thúc: ta kiểm tra trạng thái hiện tại có phải là trạng thái kết thúc không bằng cách truyền state hiện tại vào hàm **isBFSGoalState** (trong trường hợp sử dụng thuật toán BFS để tìm lời giải) hay **isAStarGoalState** (trong trường hợp sử dụng thuật toán A*). Trong ma trận của trạng thái kết thúc, sẽ xuất hiện các phần tử có giá trị 2, mỗi phần tử này được đặt cạnh một phần tử mang giá trị 1.

D. Legal Moves:

Trong file **state.py** còn định nghĩa một số hàm để sinh ra các trường hợp trạng thái hợp lệ tiếp theo từ trạng thái hiện tại (legal moves):

1. **genNextStatesList**: nhận vào 1 state (trạng thái hiện tại) và trả về 1 list chứa tất cả các state hợp lệ tiếp theo từ state đã truyền vào (thông qua hàm **canPutTent** để xác định vị trí thích hợp có thể đặt lều).

   ```
   def genNextStatesList(state: State):
       res = []
       for i in range(state.size):
           for j in range(state.size):
               if canPutTent([i,j], state):
                   temp = State(state.size, copy.deepcopy(state.matrix), state.row,
                       state.col, state)
                   temp.matrix[i][j] = 2
                   temp.tents_placed = state.tents_placed + 1
                   temp.trees_uncovered = state.trees_uncovered - 1
                   temp.cost = cost_function(temp)
                   res.append(temp)
       return res
   ```

2. **canPutTent**: nhận vào 1 state (trạng thái hiện tại) và toạ độ 1 vị trí (là list có dạng [x,y])

để kiểm tra vị trí có toạ độ (x, y) trên bản đồ của state hiện tại có thể đặt lều được không, dựa vào 2 hàm khác là **isValidPosition** và **isOverAvailableTents**.

```
def canPutTent(point: list, state: State):
    if isValidPostion(point, state) and isOverAvailableTents(point, state):
        return True
    else: return False
```

3. **isValidPosition**: nhận vào 1 state và toạ độ của 1 vị trí, kiểm tra xem vị trí đó thoả mãn 3 điều kiện: là ô trống, không tồn tại lều xung quanh và tồn tại ít nhất 1 cây đứng bên cạnh theo chiều ngang hoặc dọc. Nếu thoả mãn tất cả, hàm sẽ trả về $True$.

```
def isValidPostion(point: list, state: State):
    x = point[0]
    y = point[1]
    if (state.matrix[x][y]) == 0: # check if that position is empty
        # check if at least 1 tent exists around the position
        allMoves = getAllLegalMoves(point, state)
        for ele in allMoves:
            i = ele[0]
            j = ele[1]
            if state.matrix[i][j] == 2: return False
        # check if at least 1 tree is around the position
        allVerAndFor = getHorAndVerMoves(point, state)
        for cursor in allVerAndFor:
            i = cursor[0]
            j = cursor[1]
            if state.matrix[i][j] == 1: return True
        return False
    else: return False
```

4. **isOverAvailableTents**: nhận vào 1 state và toạ độ của 1 vị trí, kiểm tra xem số lều đã được đặt trên hàng và cột của vị trí đó trên bản đồ của trạng thái hiện tại đã đạt giới hạn hay chưa. Nếu chưa, tức là ta vẫn có thể đặt lều vào vị trí đó và hàm trả về $True$.

```
def isOverAvailableTents(point: list, state: State):
    if not(isValidPostion(point, state)): return False
    else:
        cnt_row = 0
        cnt_col = 0
        point_row = point[0]
        point_col = point[1]
        # check if the number of tents is greater or equal the allowed amount
        for i in range(state.size):
            if state.matrix[point_row][i] == 2: cnt_row += 1
        if cnt_row >= state.row[point_row]: return False
        for j in range(state.size):
            if state.matrix[j][point_col] == 2: cnt_col += 1
        if cnt_col >= state.col[point_col]: return False
        return True
```

Như vậy, bằng việc định nghĩa **State** như trên, từ **Initial State** được đọc từ input, ta sẽ dùng hàm **genNextStatesList** để sinh ra các trạng thái hợp lệ tiếp theo. Cùng với các thuật toán tìm kiếm, tiếp tục làm như vậy cho đến khi tìm được trạng thái thoả mãn các hàm kiểm tra **isGoalState** của từng loại thuật toán, ta thu được trạng thái kết thúc **Goal State**.

3.1.2 Giải quyết bài toán bằng phương pháp Blind Search

Đối với Blind Search, nhóm chọn giải thuật **Breadth First Search (BFS)**. Nhóm sẽ hiện thực thuật toán này thông qua hàm **BFS** được định nghĩa trong class **Search** trong file **search.py**.

Trong thuật toán này, hàm **isBFSGoalState** (trong class State) được dùng kiểm tra 1 trạng thái có phải là trạng thái kết thúc hay chưa, bằng cách so sánh số lều đã được đặt của trạng thái đó (*state.tents.placed*) với tổng số cây cần phải đặt (tổng các phần tử của *state.row* hoặc *state.col*).

```
def isBFSGoalState(state: State):
    sum = 0
    for i in state.row: sum += i
    if state.tents_placed == sum: return True
    else: return False
```

Class **Search** có 1 attribute là *init_state*, các thuật toán tìm kiếm sẽ sử dụng nó làm trạng thái bắt đầu (Initial State). Các Node trong trường hợp này là các State của game và ta sẽ dùng 2 list là *queue* và *visited* lần lượt tương ứng với hàng đợi và danh sách đỉnh đã thăm như trong Pseudocode ở trên.

```
class Search:
def __init__(self, state: State):
    self.init_state = state

def BFS(self):
    start_time = time.time()
    queue = [self.init_state]
    visited = []
    while len(queue) != 0:
        current_state = queue.pop(0)
        visited.append(current_state)
        if isBFSGoalState(current_state):
            execute_time = time.time() - start_time
            total_generated_states = len(queue) + len(visited)
            return current_state, execute_time, total_generated_states
        else:
            next_states_list = genNextStatesList(current_state)
            for s in next_states_list:
                if s not in visited and s not in queue:
                    queue.append(s)
```

Chi tiết về giải thuật như sau:

- **Bước 1**: Sử dụng 2 list *queue* và *visited* để khởi tạo hàng đợi và danh sách đỉnh đã được thăm. Sau đó, truyền Initial State (là *init_state* của class) vào *queue*.
- **Bước 2**: Kiểm tra hàng đợi *queue* có rỗng không. Trong trường hợp rỗng, đồng nghĩa với

việc bài toán không có lời giải, ta sẽ thoát khỏi hàm và dừng chương trình. Nếu không, tiếp tục qua Bước 3.

- **Bước 3**: Lấy ra từ hàng đợi trạng thái hiện tại (*current_state*) và truyền nó vào *visited* để đánh dấu state đó đã được viếng thăm.
- **Bước 4**: Kiểm tra *current_state* có phải là Goal State hay không (bằng cách truyền *current_state* vào hàm $isBFSGoalState$). Nếu đúng, hàm sẽ trả về *current_state*, thời gian thực thi (*execute_time*) và tổng số state đã đi qua (*total_generated_states*). Nếu không, tiếp tục Bước 5.
- **Bước 5**: Từ *current_state*, ta sẽ sinh ra các trạng thái kế tiếp hợp lệ bằng Legal Moves đã định nghĩa (kết quả được lưu vào danh sách *next_states_list*).
- **Bước 6**: Cuối cùng, từ những states liền kề vừa được sinh ra, ta sẽ kiểm tra từng state xem đã được viếng thăm hay chưa. Nếu chưa, đẩy nó vào hàng đợi và quay lại Bước 2.

3.1.3 Giải quyết bài toán bằng giải thuật Heuristic

Đối với Heuristic Search, nhóm chọn giải thuật **A*** với hàm chi phí $f(n)$ được tính bằng công thức:

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

Trong đó:

- n: state hiện tại đang được xét.
- $g(n)$: tổng số lều đã được đặt tại trạng thái đang xét.
- $h(n)$:

$$\sum_{0}^{k}(rowClue[k] - posCanPutTentOfRow[k] + failedRow[k]) + stateCountFailedTree$$

  Với
  - $rowClue[k]$: số lượng lều được phép đặt còn lại của hàng thứ k.
  - $posCanPutTentOfRow[k]$: số lượng các ô có thể đặt lều còn lại của hàng thứ k.
  - $countFailedTree$: số lượng cây chưa được lều che phủ, nhưng không còn nơi đặt lều hợp lệ trong state đang xét.
  - $failedRow[k]$: giá trị m nếu ở hàng thứ k, số lượng lều được phép đặt $>$ số lượng ô có thể đặt lều còn lại.

Chi tiết hơn về hàm chi phí $f(n)$, nhóm đã hiện thực hàm **cost_function** cùng một số hàm phụ khác giúp thuận tiện hơn trong việc tính giá trị *cost* của trạng thái đang xét trong file **state.py**.

Hầu hết các bước và việc hiện thực giải thuật A* cũng tương tự như BFS. Nhưng trong thuật toán này, ta sử dụng cấu trúc dữ liệu *hàng đợi ưu tiên (priority queue)* thay cho việc sử dụng hàng đợi thông thường, nhằm thuận tiện hơn trong việc lấy node (state) có giá trị *cost* nhỏ nhất trong hàng đợi ra để xét.

```
def A_Star(self):
    start_time = time.time()
    queue = PriorityQueue()
    queue.put(self.init_state)
    visited = []
    while (queue.qsize() != 0):
        current_state = queue.get()
        visited.append(current_state)
        if isAStarGoalState(current_state):
            execute_time = time.time() - start_time
            total_generated_states = queue.qsize() + len(visited)
            return current_state, execute_time, total_generated_states
        else:
            next_state_list = genNextStatesList(current_state)
            for s in next_state_list:
                if s not in visited:
                    queue.put(s)
```

3.2 Light Up

3.2.1 Định nghĩa trạng thái (States) và các luật di chuyển hợp lệ (Legal Moves)

A. State:

Đối với game Light Up, trạng thái của bài toán được nhóm định nghĩa là 1 ma trận, mỗi phần tử là một số nguyên với ý nghĩa như sau:

- **-1**: ô trắng.
- **-2**: ô được đánh dấu **X**.
- **0-4**: ô đen (chướng ngại vật), ứng với con số bên trong ô biểu thị số bóng đèn phải đặt cạnh.
- **5**: ô đen (chướng ngại vật) không chứa số.
- **8**: ô chứa đèn.

Ngoài ra còn một số phần tử trên ma trận có giá trị khác thoả mãn các biểu thức sau:

- $(a + 2)\%8 = 0$: ô được chiếu đèn và được đánh dấu **X**.
- $(a + 1)\%8 = 0$: ô trắng được chiếu đèn.

[23, 7, 15, 7, 15, 6, 8]

[15, 5, 8, 1, 6, 0, 6]

[15, -1, 7, 5, 8, 6, 15]

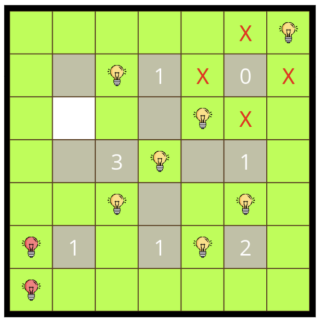
[15, 5, 3, 8, 5, 1, 7]

[23, 7, 8, 5, 15, 8, 15]

[8, 1, 7, 1, 8, 2, 7]

[8, 7, 15, 7, 15, 7, 15]

score: 36 - 1 - 1 = 34



Hình 8: Mô tả 1 trạng thái hợp lệ bất kỳ của Light Up

B. Initial State:

Trạng thái khởi động của bài toán (input): chưa có đèn được đặt. Tất cả các ô đều có giá trị không quá 5.

[[-1, -1, -1, -1, -1, -1, -1],

[-1, 5, -1, 1, -1, 0, -1],

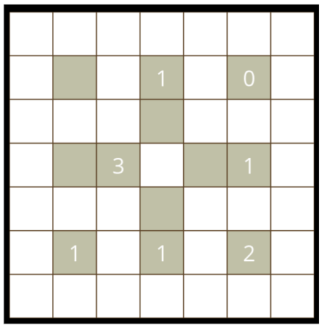
[-1, -1, -1, 5, -1, -1, -1],

[-1, 5, 3, -1, 5, 1, -1],

[-1, -1, -1, 5, -1, -1, -1],

[-1, 1, -1, 1, -1, 2, -1],

[-1, -1, -1, -1, -1, -1, -1]]



Hình 9: Mô tả 1 trạng thái khởi đầu

C. Goal State:

Trạng thái kết thúc của bài toán: tất cả các ô đều có giá trị trong đoạn $[0, 15]$.

[15, 7, 15, 8, 15, 6, 15]

[7, 5, 7, 1, 6, 0, 6]

[15, 7, 8, 5, 8, 6, 15]

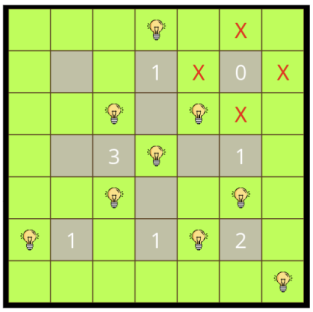
[7, 5, 3, 8, 5, 1, 7]

[15, 7, 8, 5, 15, 8, 15]

[8, 1, 7, 1, 8, 2, 7]

[15, 7, 15, 7, 15, 7, 8]

score: 37 - 0 - 0 = 37



Hình 10: Mô tả 1 trạng thái kết thúc

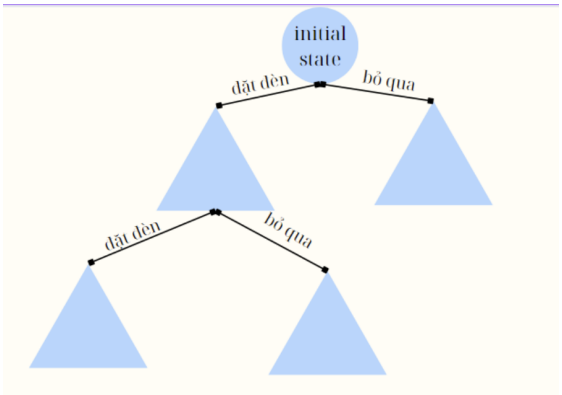
D. Legal Moves:

Mỗi trạng thái có thể có ba hành động: xóa một đèn, thêm một đèn, di chuyển một đèn từ ô này sang ô khác.

3.2.2 Giải quyết bài toán bằng phương pháp Blind Search

Để hiện thực Light Up bằng phương pháp Blind Search, nhóm chọn giải thuật **Depth First Search (DFS)**. Chi tiết về giải thuật như sau:

- **Bước 1**: Tìm kiếm các ô trắng ở trạng thái khởi động theo thứ tự từ trái sang phải, từ trên xuống dưới. Với mỗi ô trống (không chứa đèn hoặc không bị đánh dấu **X**), chúng ta có hai lựa chọn là đặt đèn hoặc bỏ qua.
- **Bước 2**: Thử đặt đèn vào ô đó và cập nhật lại trạng thái, dựa trên luật của trò chơi để xem trạng thái đó có hợp lệ hay không. Nếu không, quay lại Bước 1 và không đặt đèn. Sau đó đi đến ô trắng tiếp theo và lặp lại Bước 2.
  - Đèn này chiếu sáng những ô trên cùng một hàng và cột cho đến khi nó gặp ô đen hoặc biên.
  - Đèn không thể chiếu sáng đèn khác trên cùng một hàng hoặc cột.
  - Nếu như tất cả điều kiện của trò chơi được thỏa mãn (tất cả ô trắng đều được chiếu sáng và không có đèn nào vi phạm luật), thì trạng thái hiện tại là một lời giải hợp lệ. Dừng quá trình tìm kiếm lời giải và trả về kết quả.
- **Bước 3**: Tiếp tục quá trình này cho đến khi có lời giải. Nếu như tìm kiếm hết toàn bộ các trường hợp nhưng không tìm thấy lời giải hợp lệ, ta kết thúc quá trình và ghi nhận trò chơi với trạng thái ban đầu này không có lời giải.



Hình 11: Mô tả thuật toán DFS được áp dụng cho game Light Up

3.2.3 Giải quyết bài toán bằng giải thuật Heuristic

Simulated Annealing là một thuật toán tối ưu hóa xác suất có thể được sử dụng để giải trò chơi Light Up. Cách thuật toán hoạt động đối với game này như sau:

- Tạo ra các trạng thái lân cận thông qua việc thực hiện các "legal moves" từ một "state" cụ thể.
- Tính toán sự khác biệt: đánh giá điểm số của trạng thái hiện tại và trạng thái lân cận.
- Chấp nhận hoặc từ chối trạng thái mới: quyết định có lấy trạng thái mới làm trạng thái hiện tại không dựa trên việc xác định xem nó có tốt hơn không. Nếu tốt hơn, luôn chấp nhận nó, ngược lại, chấp nhận nó với một xác suất nhất định dựa trên nhiệt độ và khác biệt năng lượng. Xác suất này giảm dần khi nhiệt độ giảm.
- Giảm nhiệt độ sau mỗi lần lặp.
- Giải thuật kết thúc khi thỏa mãn "goal state" hoặc giải thuật đi hết số vòng lặp.

Lưu ý: Cả hai giải thuật DFS và Simulated Annealing, trước khi đi vào các bước của giải thuật, thực hiện quá trình tiền xử lý để làm tăng độ tốt của trạng thái khởi đầu:

- **DFS**: Xác định những nơi bắt buộc *phải* đặt đèn (ví dụ: ô có giá trị 2 nằm trong trong góc bản đồ, ô có giá trị 4) và tiến hành đặt cố định những đèn đó.
- **Simulated Annealing**: Xác định những nơi bắt buộc *không* được đặt đèn (xung quanh ô có giá trị 0) và đánh dấu **X** (giá trị -2) cho những ô này. Xác định những ô có giá trị $[0-4]$ và chọn ngẫu nhiên những ô xung quanh nó đặt đèn vào với số lượng đèn bằng với gía trị của ô đó. Cuối cùng, nếu số lượng đèn vừa đặt ít hơn *size* của ma trận (7, 10 hoặc 14), ta tiếp tục chọn ngẫu nhiên một số ô trắng để đặt đèn vào đến khi số lượng đèn bằng với size của trò chơi thì bắt đầu giải thuật.

4 Thử nghiệm, đánh giá và kết luận

4.1 Một số lưu ý về cài đặt

Để chạy được chương trình, sau đây là một số lưu ý:

1. Cài đặt môi trường ngôn ngữ **Python 3**.
2. Sau khi cài đặt môi trường thành công, ta sẽ thực hiện cài đặt framework **pygame** bằng câu lệnh: **pip install pygame**
3. Sau khi cài đặt thành công, chúng ta chạy chương trình bằng câu lệnh: **python main.py**

Lưu ý: Trong trường hợp sau khi chạy câu lệnh **python main.py** và tiến hành các bước nhập như yêu cầu, cửa sổ pygame hiện ra mà không có gì, chúng ta có thể submit folder chứa code lên trang web Replit, sử dụng môi trường Pygame và tiến hành như các bước trên để khắc phục lỗi này.

4.2 Tents

Link github: https://github.com/duyanhhh1811/CO3061-tents-puzzle-solver

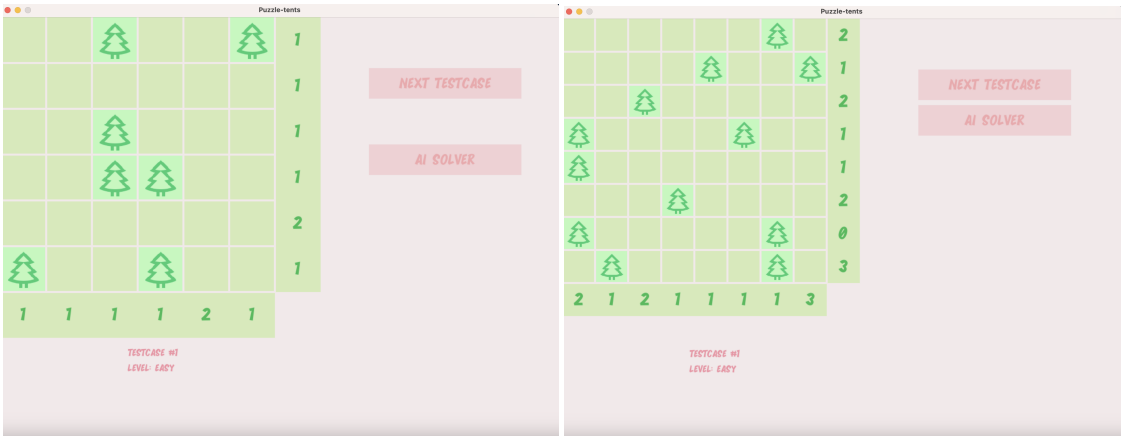
4.2.1 Giao diện game và thử nghiệm chương trình

Sau khi chạy câu lệnh **python main.py** thành công, trên terminal sẽ hiển thị như sau:

```
duyanhle@Duys-Air CO3061-assignment-1 % python3 main.py
pygame 2.4.0 (SDL 2.26.4, Python 3.11.4)
Hello from the pygame community. https://www.pygame.org/contribute.html
CHOOSE SIZE (6 or 8):
```

Hình 12: Terminal sau khi chạy thành công chương trình Tents

Đối với game Tents, nhóm có thực hiện tính năng demo lời giải bằng giao diện trực quan cho bản đồ có kích thước $6*6$ và $8*8$. Mỗi loại bản đồ có 10 testcases: 5 testcases có level dễ (Easy) và 5 testcases có level khó (Hard) được chọn lọc trên trang puzzle-tents.com. Sau khi chọn size bản đồ (6 hoặc 8), giao diện sẽ hiển thị như sau:

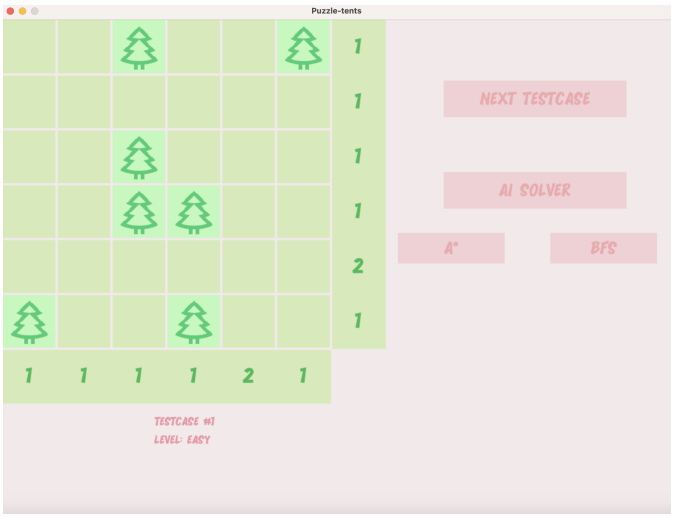


Hình 13: Giao diện game Tents ứng với từng kích thước bản đồ

Một số thông tin được hiển thị trên giao diện:

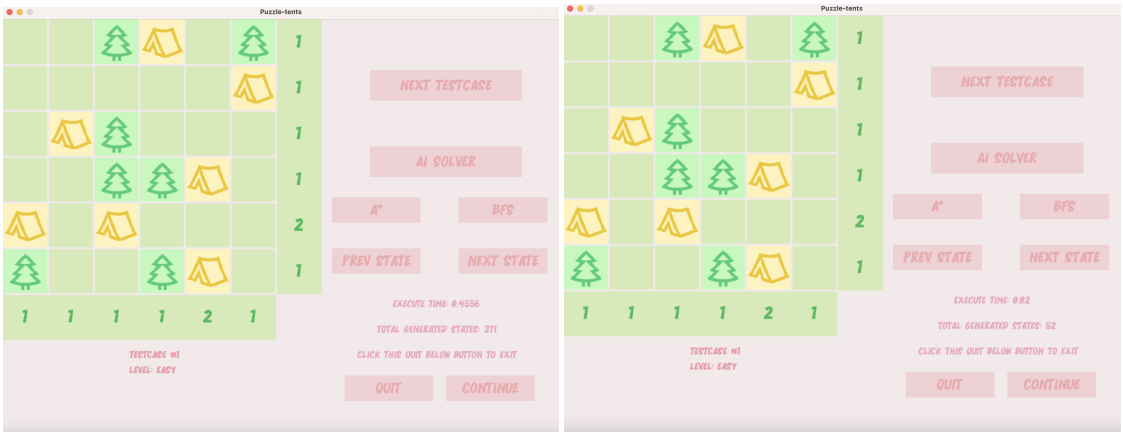
- Bản đồ trò chơi: Bản đồ được tạo ra dựa trên ma trận của trạng thái. Với initial state, các ô mang giá trị 1 có hình cây thông, còn các ô mang giá trị 0 không có hình gì. Ngay bên cạnh bản đồ là thông tin về số lều được đặt trong các hàng và cột.
- Thông tin về testcase: Ở dưới bản đồ là thông tin về testcase, bao gồm số hiệu và level (Easy/Hard).
- Button **Next Testcase**: Giúp người chơi chọn lựa testcase bằng cách hiển thị testcase tiếp theo.
- Button **AI Solver**: Giúp người chơi xem lời giải của bản đồ đã chọn.

Sau khi chọn được testcase muốn xem solution và nhấn nút **AI Solver**, ngay phía dưới sẽ có 2 nút mới hiện ra là **BFS** và **A*** để người chơi có thể lựa chọn giải thuật muốn xem.



Hình 14: Lựa chọn giải thuật để xem solution

Khi quá trình giải hoàn tất, 2 nút **Next State** và **Prev State** sẽ hiện ra. Người chơi có thể nhấn nút **Next State** để xem cách giải bài toán step-by-step, **Prev State** để quay lại bước trước đó. Sau khi hoàn tất, một số thông tin sẽ hiện ra như *Execute Time*, *Total Generated States* và 2 nút **Continue** và **Quit**. Người chơi có thể lựa chọn 1 trong 2 nút để tiếp tục hoặc thoát khỏi chương trình



Hình 15: Kết quả sau khi quá trình giải hoàn tất (BFS - trái, A* - phải)

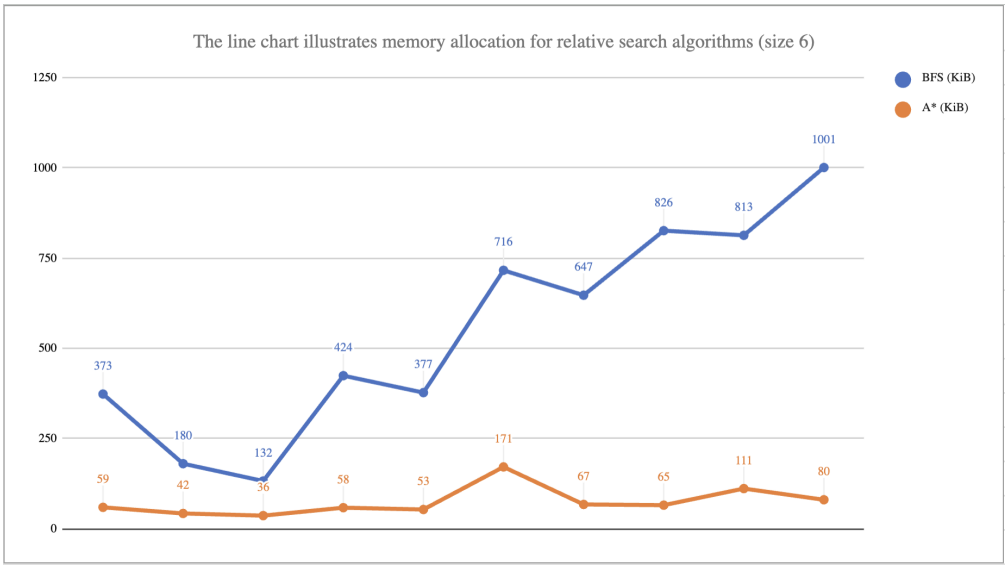
4.2.2 Đánh giá hiệu năng giải thuật

Thống kê sau khi chạy các testcases đối với game Tents của nhóm như sau:

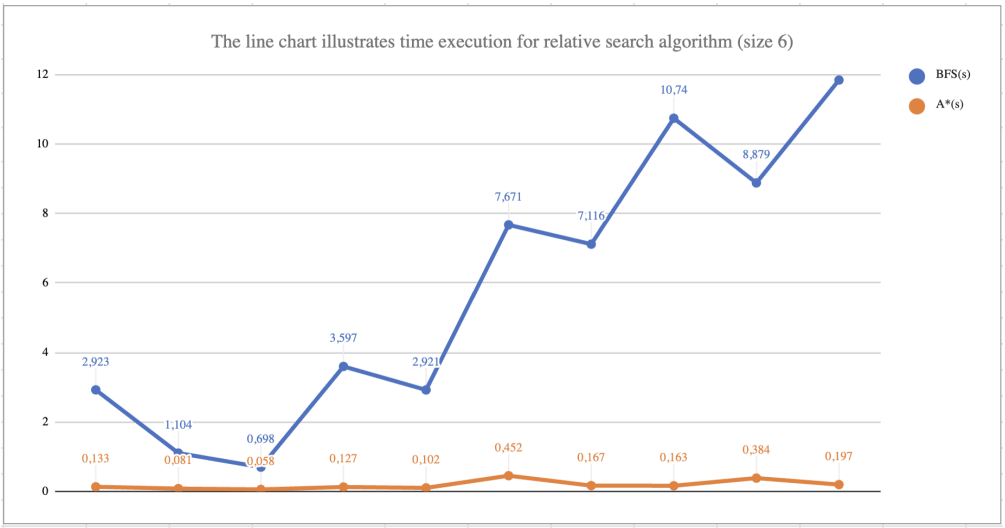
Testcase	Giải thuật	Thời gian thực thi(s)	Bộ nhớ cấp phát (KiB)
1	BFS	2,923	373
	A*	0,133	59
2	BFS	1,104	180
	A*	0,081	42
3	BFS	0,698	132
	A*	0,058	36
4	BFS	3,597	424
	A*	0,127	58
5	BFS	2,921	377
	A*	0,102	53
6	BFS	7,671	716
	A*	0,452	171
7	BFS	7,116	647
	A*	0,167	67
8	BFS	10,74	826
	A*	0,163	65
9	BFS	8,879	813
	A*	0,384	111
10	BFS	11,839	1001
	A*	0,197	80

Hình 16: Bảng số liệu hiệu năng game Tents khi áp dụng BFS và A* (Size 6)

Với kích thước bản đồ là $6 * 6$, nhóm đã thực hiện vẽ biểu đồ thống kê cho 2 thuộc tính **Thời gian thực thi** và **Bộ nhớ cấp pháp**



Hình 17: Biểu đồ đường thể hiện dung lượng bộ nhớ cấp phát cho từng testcase game Tents



Hình 18: Biểu đồ đường thể hiện thời gian thực thi của từng testcase game Tents

Ngoài ra, nhóm cũng đã thống kê số liệu cho các testcases của các bản đồ có kích thước $8*8$ và $10*10$, nhưng thời gian thực thi của giải thuật *BFS* quá lâu, khó có thể kiểm soát được thông tin. Vì vậy, trong game Tents, nhóm chỉ đưa ra kết quả khi thực thi với giải thuật $A*$

Testcase	Giải thuật	Thời gian thực thi (s)	Bộ nhớ cấp phát (KiB)
1	BFS	---	---
	A*	0,679	209
2	BFS	---	---
	A*	0,545	176
3	BFS	---	---
	A*	0,419	152
4	BFS	---	---
	A*	0,851	287
5	BFS	---	---
	A*	0,556	171
6	BFS	---	---
	A*	6,343	1601
7	BFS	---	---
	A*	1,481	411
8	BFS	---	---
	A*	2,434	693
9	BFS	---	---
	A*	20,41	4487
10	BFS	---	---
	A*	3,078	866

Testcase	Giải thuật	Thời gian thực thi(s)	Bộ nhớ cấp phát (KiB)
1	BFS	---	---
	A*	2,689	898
2	BFS	---	---
	A*	7,244	2316
3	BFS	---	---
	A*	2,318	786
4	BFS	---	---
	A*	3,813	1216
5	BFS	---	---
	A*	2,424	767

Hình 19: Bảng số liệu hiệu năng game Tents khi áp dụng BFS và A* (Size 8 và 10)

Dựa vào kết quả thu được, nhóm có một vài nhận xét như sau:

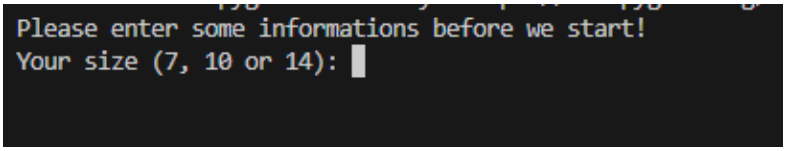
- Ở cả 2 thuật toán, thời gian tìm ra lời giải và dung lượng bộ nhớ cấp phát tăng dần theo độ khó (level) và kích thước ma trận (size). Đặc biệt, đối với size 6, có thể thấy dung lượng bộ nhớ cấp phát gần như tỉ lệ thuận với thời gian thực thi.
- Đối với những testcases phức tạp hơn (ma trận kích thước có kích thước lớn hơn 6), mặc dù đảm bảo có thể tìm được lời giải nhưng thời gian chạy của BFS rất lâu, trong khi với A* đa số các testcases chỉ cần khoảng 3 - 4 giây là đã tìm được lời giải. Nguyên nhân là do độ phức tạp của giải thuật BFS cao hơn rất nhiều so với A*, dẫn đến khi gặp những bài toán phức tạp, BFS sẽ mất thời gian cho việc traverse giữa các state lâu hơn rất nhiều.

4.3 Light Up

Link github: https://github.com/hungvo2003vn/Light-Up.git

4.3.1 Giao diện game và thử nghiệm chương trình

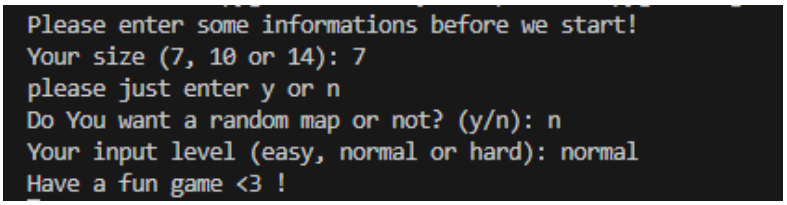
Sau khi chạy câu lệnh **python main.py** thành công, trên terminal sẽ hiển thị như sau:



Hình 20: Terminal sau khi chạy thành công chương trình game Light Up

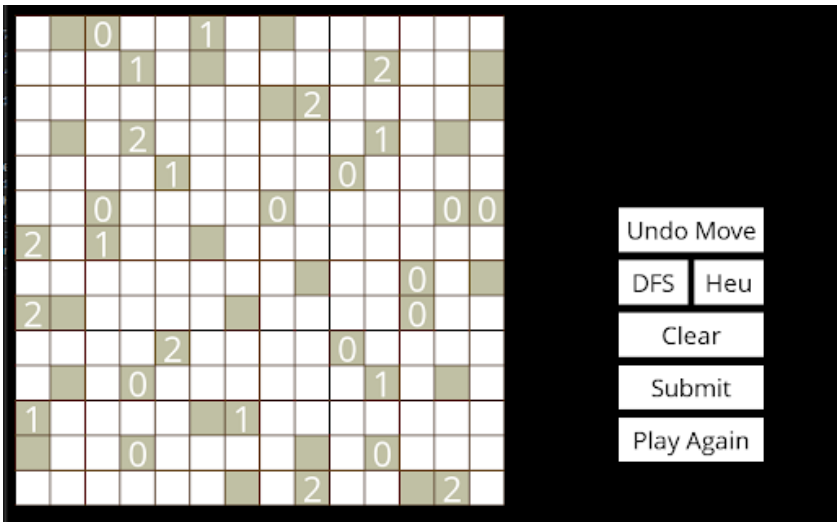
Tiến hành nhập các thông tin yêu cầu:

1. Chọn kích thước trò chơi: 7, 10, 14
2. Chọn chế độ chơi: **"n"** cho chế độ *random* và **"y"** cho chế độ *level*.
3. Khi nhập **"y"**, cần nhập level: *easy*, *normal*, *hard*. Còn với chế độ **random**, input sẽ được tạo ngẫu nhiên theo kích thước trò chơi đã nhập.



Hình 21: Nhập input theo yêu cầu

Sau khi nhập thành công các thông tin được yêu cầu, màn hình giao diện game như sau:

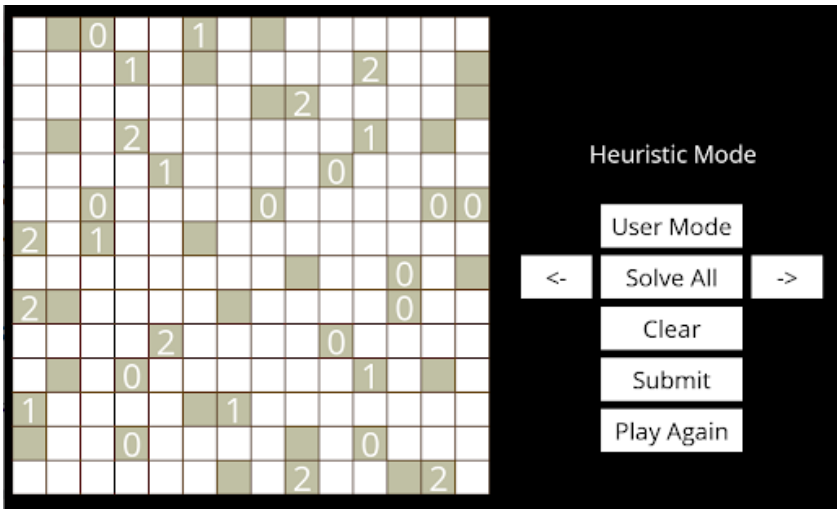


Hình 22: Giao diện game Light Up

Mặc định, người dùng đầu tiên sẽ ở "chế độ người chơi", người chơi có thể thử sức mình tự tay giải quyết bài toán này, bằng cách click chuột vào bản đồ. Một số chức năng được hỗ trợ trên giao diện ở chế độ người chơi như sau:

- **Undo Move**: lùi lại trạng thái trước đó của bảng.
- **Clear**: đưa bản đồ trở lại trạng thái ban đầu.
- **Submit**: kiểm tra trạng thái hiện tại có phải lời giải không.
- **Play Again**: chơi lại.

Đặc biệt, có 2 nút là **DFS** và **Heu** hỗ trợ người chơi tìm ra lời giải theo thuật toán đã chọn. Sau khi chọn 1 trong 2 thuật toán, giao diện sẽ hiển thị như sau:



Hình 23: Giao diện game Light Up sau khi chọn thuật toán

Người chơi có thể sử dụng các nút "->" hay "<-" để đi đến trạng thái tiếp theo hoặc quay về trạng thái trước đó. Ngoài ra, khi nhấn vào nút **Solve All**, toàn bộ lời giải sẽ được show lên màn hình. Người chơi cũng có thể chọn **User Mode** để quay lại chế độ người chơi.

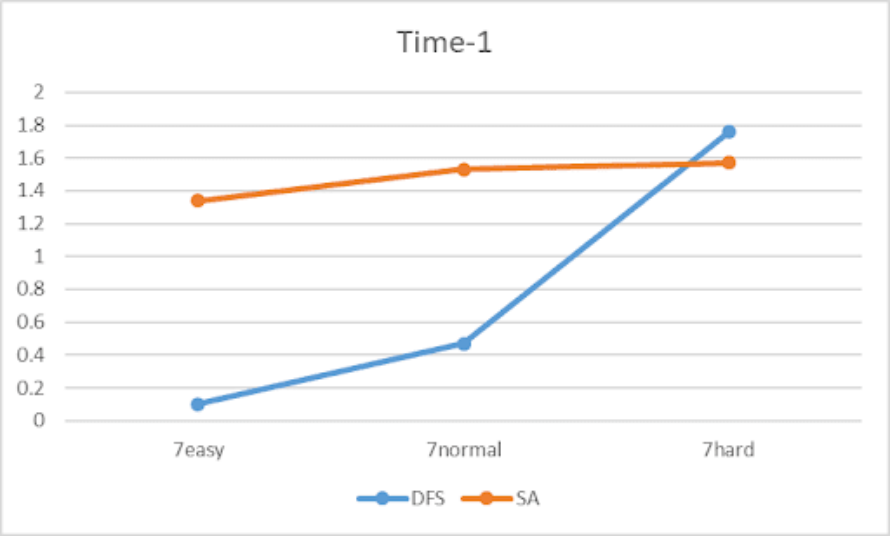
4.3.2 Đánh giá hiệu năng giải thuật

Thống kê sau khi chạy các testcases đối với game Light Up như sau:

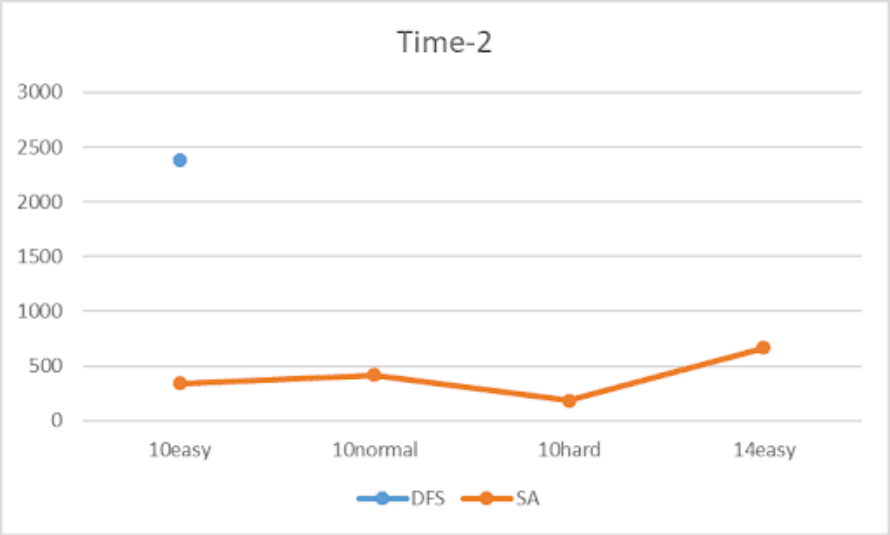
Kích thước	Level	Giải thuật	Thời gian thực thi (s)	Bộ nhớ cấp phát (KiB)
7 x 7	easy	DFS	0.1	26
		SA	1.34	5
	normal	DFS	0.47	114
		SA	1.53	4
	hard	DFS	1.76	402
		SA	1.57	18
10 x 10	easy	DFS	2382.41	68006
		SA	340.78	10
	normal	DFS	TLE (> 3g)	-----
		SA	416.00	185
	hard	DFS	TLE (> 3g)	-----
		SA	184.06	179
14 x 14	easy	DFS	TLE (> 3g)	-----
		SA	666.07	22
	normal	DFS	TLE (> 3g)	-----
		SA	-----	-----
	hard	DFS	TLE (> 3g)	-----
		SA	------	-----

Hình 24: Thống kê số liệu game Light Up

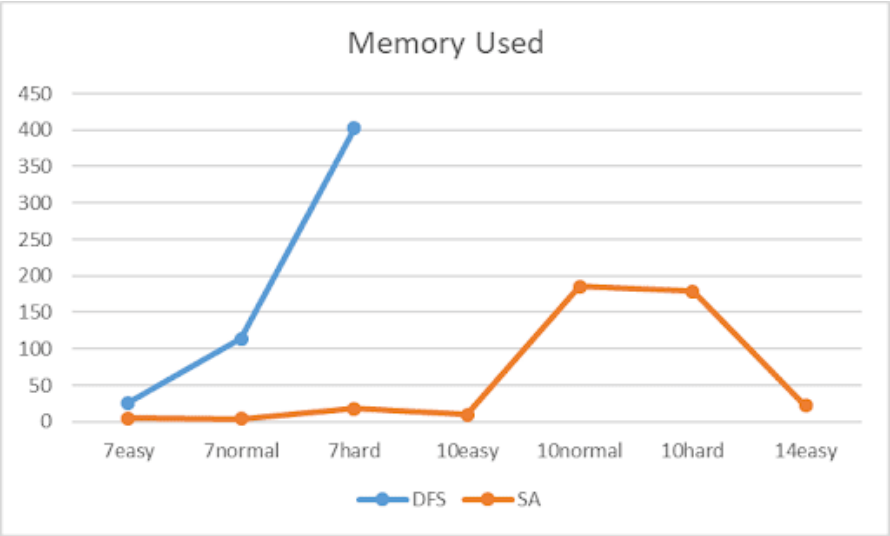
- **DFS**: Có độ phức tạp $O(2^n)$, với n là số ô trắng có thể đặt đèn. Khi kích thước bảng quá lớn (ví dụ với kích thước đầu vào $10 * 10$ và độ khó Easy, n = 36 - chạy mất 2382s) dẫn đến độ phức tạp lớn nên chạy rất lâu nhưng sẽ đảm bảo luôn tìm được lời giải.
- **SA**: Việc tìm kiếm nước đi tiếp theo và quyết định nước đi nào cho giá trị tốt nhất có sự ảnh hưởng lớn bởi xác suất và ngoài ra, nếu nước đi này làm cho ta kẹt vào trạng thái 'tối ưu cục bộ' sẽ dẫn đến việc không tìm ra được lời giải thỏa mãn và kẹt tại đó vô tận - điều này giải thích cho lí do tại sao một số testcase ở trên không được ghi nhận thời gian chạy với giải thuật này.



Hình 25: Biểu đồ đường thể hiện thời gian thực thi game Light Up (Size 7)



Hình 26: Biểu đồ đường thể hiện thời gian thực thi game Light Up (Size 10 và 14)



Hình 27: Biểu đồ đường thể hiện dung lượng bộ nhớ cấp phát game Light Up

Từ các kết quả trên, có thể thấy: đối với cả 2 thuật toán DFS và SA, thời gian tìm ra lời giải và dung lượng bộ nhớ cấp đều tăng lên theo độ khó màn chơi.

- Đối với **DFS**: Thời gian tỉ lệ thuận với dung lượng bộ nhớ tiêu tốn.
- Đối với **SA**: Thời gian chạy ở màn chơi cấp dễ nhất cao hơn DFS, nhưng ở những màn chơi khó hơn, kích thước đầu vào lớn hơn thì lại thấp hơn rất nhiều.

Nguyên nhân là ở những màn chơi khó hơn, DFS tốn rất nhiều thời gian để duyệt qua lần lượt từng đường đi để tìm ra lời giải hợp lệ. Độ phức tạp của thuật toán rơi vào $O(2^n)$ với n là

số ô trắng của màn chơi. Trong khi đó, giải thuật Simulated Annealing sẽ chỉ chọn những trường hợp tối ưu (với điểm số cao) để duyệt qua và chỉ cho phép những chấp nhận những trường hợp có điểm số thấp hơn với một xác suất nhất định.

5 Tài liệu tham khảo

1. https://en.wikipedia.org/wiki/Depth-first_search
2. https://en.wikipedia.org/wiki/Breadth-first_search
3. https://en.wikipedia.org/wiki/A*_search_algorithm
4. https://en.wikipedia.org/wiki/Simulated_annealing