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

NHẬP MÔN TRÍ TUỆ NHÂN TẠO (CO3061)

BÀI TẬP LỚN
BLOXORZ AND WATER SORT PROBLEMS

Giảng viên hướng dẫn: **Vương Bá Thịnh**

Sinh viên thực hiện: **Võ Phạm Tuấn Dũng - 2010013**
Phạm Hữu Phú - 2010516
Phạm Cảnh Hưng - 2010029
Nguyễn Đình Tuấn - 2010754

TP. Hồ Chí Minh, Tháng 11/2022

Mục lục

Danh sách thành viên và phân chia công việc

STT	Họ và tên	MSSV	Công việc	Mức độ hoàn thành
1	Võ Phạm Tuấn Dũng	2010013	Nhóm trưởng - phân việc, quản lí tiến độ nhóm Phụ trách chính ý tưởng, hiện thực phần A* Water Sort Phụ trách chính ý tưởng, hiện thực phần GA Bloxorz Hỗ trợ fix bug, sinh testcase cho thành viên	100%
2	Phạm Hữu Phú	2010516	Phụ trách chính ý tưởng, hiện thực phần DFS Water Sort Phụ trách chính ý tưởng, hiện thực phần DFS Bloxorz Phụ trách phần hiện thực GUI demo 2 game Kiểm tra coding style và refactor source code	100%
3	Nguyễn Đình Tuấn	2010754	Phụ trách phần báo cáo, đo đạc thời gian và bộ nhớ Phụ trách phần thống kê và phân tích kết quả Tham gia hiện thực phần BFS/DFS 2 game Tham gia hiện thực GUI cả 2 game	100%
4	Phạm Cảnh Hưng	2010029	Tham gia viết báo cáo Kiểm tra chính tả báo cáo và source code	100%

Phần I
Water Sort Problem

1 Mô tả luật trò chơi và input file

1.1 Luật trò chơi

- Trò chơi có X ống thủy tinh có kích cỡ y hệt nhau. Một số ống thủy tinh có chứa các cột chất lỏng đồng màu khác nhau, tổng chiều dài các cột đồng màu trong mỗi cốc không được vượt khỏi chiều dài của ống.
- Độ dài mỗi cốc được chia thành n đơn vị độ dài. Mỗi cột chất lỏng đồng màu có chiều dài là một số nguyên đơn vị độ dài.
- Ứng với mỗi màu, tổng các cột màu tương ứng đúng bằng kích thước của **chỉ một** cột thủy tinh. Điều này rất quan trọng vì nó sẽ ảnh hưởng tới phần hiện thực giải thuật A*.
- Ở mỗi bước, người chơi chọn hai ống A và B sao cho có hai cột chất lỏng đồng màu ở đỉnh mỗi ống có màu giống nhau, hoặc ống B đang rỗng. Người chơi sẽ chọn cột màu ở 1 ống rồi đổ qua cột bên kia. Hai cột chất lỏng giống nhau về màu khi tiếp xúc sẽ hòa tan trở thành một cột màu duy nhất.
- Chú ý khi đổ một cột chất lỏng từ một ống sang ống khác phải xem xét sức chứa hiện tại của ống sẽ nhận chất lỏng. Nếu trong quá trình đang đổ ống này chạm đến sức chứa tối đa thì phải dừng lại ngay.
- Điều kiện thắng: Người chơi phải thực hiện các bước sao cho kết quả cuối cùng, trong mỗi cốc chỉ chứa một cột chất lỏng đồng màu ở chiều cao tối đa hoặc không chứa cột chất lỏng nào cả.

1.2 Mô tả input file

Test file là một test case dưới dạng file text cung cấp thông tin về số lượng cốc, số lượng màu, trạng thái khởi đầu của bài toán. Tùy theo mỗi test case dễ hay khó mà trạng thái khởi đầu của bài toán sẽ khác nhau. Một số test file đã được nhóm chuẩn bị sẵn được đặt trong thư mục testcase/. Ví dụ testcase0.txt sẽ có nội dung như sau.

```
7
4


C L X H
H D C D
L L D X
L C X H
C H D X
```

- Dòng đầu tiên mô tả số cốc có trong bài toán là 7
- Dòng thứ 2 có số lượng màu là 4.
- Từ dòng 3 đến dòng 9 tương ứng với trạng thái ban đầu. Được quy định như sau:

- Dòng thứ 3, 4 thể hiện cốc 1 và 2 rỗng
- Các dòng còn lại lần lượt thể hiện cho các cốc 3,4,5,6,7 có chứa các màu nước.

Trong đó:

- Ứng với mỗi chữ cái tương đương với 1 màu nước
- Chữ cái đầu tiên ở mỗi dòng tương ứng với màu ở đáy của mỗi cốc

Ví dụ:

- C L X H cho biết cốc thứ 3 có các màu nước theo thứ tự [C,L,X,H] với C là màu ở đáy cốc và H là màu ở đỉnh cốc
- L L D X cho biết cốc thứ 5 có các màu nước theo thứ tự [L,L,D,X] với L là màu ở đáy cốc và X là màu ở đỉnh cốc

Ban đầu, nhóm đã có ý tưởng về những test case bằng cách định nghĩa mỗi màu bằng 1 từ có nghĩa như XANH DO DO VANG, tuy nhiên nó khó khăn trong việc tạo ra 1 test case ngẫu nhiên với những từ dài dòng, nhóm đã có ý tưởng tiếp theo là sử dụng những con số để định nghĩa cho từng màu nhưng lại bị giới hạn số màu quá lớn (chỉ tối đa được 10 màu từ 0-9). Cuối cùng nhóm đã nghĩ ra giải pháp tối ưu nhất là định nghĩa mỗi màu bằng 1 kí tự trong bộ 81 kí tự đã chọn chọn sẵn. Từ đó nhóm chúng em đã tạo ra 1 file testcase_generation.py nhằm sinh ra các testcase ngẫu nhiên với mức độ khó hơn, chi tiết 81 kí tự có để trong hàm generate đặt trong file này.

Ta có thể lấy ví dụ về testcase (vì test case quá dài nên rút gọn lại bằng dấu 3 chấm, chi tiết testcase có trong thư mục testcase/testcase17.txt)

```
92
4
p = v i
? L Q x
^ S - =
# ^ e I
O [ n $
q A B L
C K } B
g ^ * H
U Y ? G
J m y ‘
s s p h
f n I w
{ * k r
J y [ b
z = + d
M W Q +
W ‘ ; b
{ f K k
q , $ g
Z R ~ K
p + ‘ #
^ F j B
G < h -
...
```

2 Ý tưởng hiện thực

2.1 Thiết kế không gian trạng thái cho bài toán

2.1.1 Không gian trạng thái

Mỗi trạng thái là một danh sách các cốc, mỗi cốc là một stack các cột chất lỏng đồng màu. Để dễ làm việc, ta hiện thực mỗi cốc bằng một chuỗi, ví dụ 'XXV', các phép biến đổi trên stack quy về phép biến đổi trên chuỗi.

Không gian trạng thái là tập hợp các trạng thái có thể có của các cốc, không tồn tại trường hợp tất cả các đỉnh cốc đều có màu khác nhau. Mỗi node trong cây tìm kiếm, tương ứng với 1 trạng thái sẽ bao gồm các thuộc tính state, parent, action, trong đó action của node là hành động để từ node parent của nó sinh ra nó.

Trong đó:

- *state* dùng để lưu thông tin của trạng thái, được mô tả bằng danh sách các stack - ứng với mỗi stack là một cốc, bao gồm các phần tử là các màu nước tương ứng chứa trong cốc đó
- *action* là hành động đổ nước từ cốc này sang cốc khác
- *parent* là trạng thái cha của node đó, hay là node mà chúng ta đã sử dụng để dẫn đến trạng thái hiện tại, điều này giúp cho ta theo dõi được toàn bộ hành động để thực hiện từ đầu đến cuối và giải quyết vấn đề tìm ra con đường đi tới kết quả của bài toán

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

```
class NodeForBFS:
    def __init__(self, state, parent=None, action=None):
        self.state = state
        self.parent = parent
        self.action = action
```

2.1.2 Trạng thái ban đầu

Trạng thái ban đầu là trạng thái đầu tiên của bài toán, nó được xác định từ các dòng 3 trở về sau (với dòng 1 xác định số lượng cốc và dòng 2 xác định số lượng màu trong bài).

2.1.3 Trạng thái mục tiêu

Ở bài toán WaterSort, trạng thái mục tiêu là trạng thái mà tất cả các cốc đều chỉ chứa 1 màu duy nhất ở mỗi cốc hoặc rỗng. Do đó sẽ có nhiều lời giải khác nhau cho trạng thái mục tiêu

Trạng thái có là trạng thái mục tiêu hay không sẽ được kiểm tra bởi method **isGoal** sau:

```
def isGoal(self, state):
        count = 0
        for i in range(self.num_cups):
            if self.numColorSimilar(state[i]) == self.capacity or
               self.numColorSimilar(state[i]) == 0:
                count += 1
        if count == self.num_cups:
            return True
        else:
            return False
```

2.1.4 Các bước chuyển hợp lệ

Từ một trạng thái ta có thể sinh ra các bước chuyển bằng cách đổ màu nước từ cốc này sang cốc khác thỏa mãn quy luật chuyển trạng thái như sau:

1. Tồn tại stack A, stack B (ứng với 2 cốc A và B) thỏa mãn 2 cột chất lỏng đồng màu ở đỉnh hai cốc có màu giống nhau và stack B chưa đầy (cốc B còn sức chứa) $\Rightarrow$ A có thể chuyển qua B.

2. Tồn tại stack A, stack B thỏa mãn stack B rỗng $\Rightarrow$ A có thể chuyển qua B.

Quy luật chuyển trạng thái được xác định lại qua đoạn code như sau:

```
def getLegalMove(self, state, i, j):
        len_i = len(state[i])
        len_j = len(state[j])

        if len_i == 0 or len_j == self.capacity or (len_i != 0 and  len_j != 0
            and state[i][-1] != state[j][-1]):
            return None
        else:

            num = self.numColorSimilar(state[i])
            sum_check = num + len_j
            if sum_check > self.capacity:
                return i, j, self.capacity - len_j
            else:
                return i, j, num

    def move(self, state, legal_move_tuple):
        clone_state = []
        for cup in state:
            clone_state.append(cup)

        i, j, num_to_move = legal_move_tuple

        while num_to_move != 0:
            color_out = clone_state[i][-1]
            clone_state[i] = clone_state[i][:-1]
            clone_state[j] += color_out
            num_to_move -= 1
        return clone_state
```

Sau khi có được bước chuyển hợp lệ, ta có thể di chuyển nước màu đó bằng bước chuyển được định nghĩa như sau:

```
def move(self, state, legal_move_tuple):
        clone_state = []
        for cup in state:
            clone_state.append(cup)

        i, j, num_to_move = legal_move_tuple

        while num_to_move != 0:
            color_out = clone_state[i][-1]
            clone_state[i] = clone_state[i][:-1]
            clone_state[j] += color_out
            num_to_move -= 1
        return clone_state
```

2.2 Giải thuật

2.2.1 Mã giả cho giải thuật DFS, BFS và A*

Cả ba giải thuật BFS, DFS, A* nhìn chung có mã giả giống nhau. Điểm khác biệt duy nhất giữa chúng là cách chọn cấu trúc dữ liệu để tổ chức frontier (hàng đợi các node sắp được khai phá):

1. Frontier của giải thuật BFS được tổ chức theo FIFO queue.
2. Frontier của giải thuật DFS được tổ chức theo LIFO queue (stack).
3. Frontier của giải thuật A* được tổ chức theo priority queue, với độ ưu tiên là f, rồi mới đến h. Node có giá trị f càng cao sẽ càng được ưu tiên, trường hợp hai node có giá trị f bằng nhau, sẽ ưu tiên node có giá trị h cao hơn. Giá trị f, h sẽ được nói kĩ hơn ở mục tiếp theo.

Mã giả chung mô tả cho cả 3 thuật toán BFS, DFS, A*

```
Search(problem, type)
Node <- NODE(problem.INITIAL)
If problem.IS-GOAL(node.STATE) then return node
Frontier <- ( a FIFO queue if type is BFS, a LIFO queue if type is DFS, a
             Priority queue if type is A*), with node as an element
While not IS-EMPTY(frontier) do
        Node <- POP(frontier)
        For each child in EXPAND(problem, node) do
                S <- child.STATE
                If problem.IS-GOAL(s) then return chile
                If s is not in reached then
                        add s to reached
                        add child to frontier
return failure
```

2.2.2 Chi tiết hàm heuristic cho giải thuật A*

A* là giải thuật tìm kiếm trong đồ thị, tìm đường đi từ một đỉnh hiện tại đến đỉnh đích có sử dụng hàm để ước lượng khoảng cách hay còn gọi là hàm Heuristic.

Thứ tự ưu tiên cho một đường đi được quyết định bởi hàm Heuristic được đánh giá qua hàm:

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

- g(x) là chi phí của đường đi từ điểm xuất phát cho đến thời điểm hiện tại.
- h(x) là hàm ước lượng chi phí từ đỉnh hiện tại đến đỉnh đích f(x) thường có giá trị càng thấp thì độ ưu tiên càng cao.

Do đó, ta cần định nghĩa lại trạng thái của bài toán để phù hợp hơn với giải thuật A* Mỗi trạng thái được định nghĩa bằng một Node, trong Node sẽ bao gồm các thuộc tính state, parent, action, g_value, f_value

Trong đó

- *g_value* tương đương với g(x) - chi phí của đường đi từ điểm xuất phát cho đến thời điểm hiện tại

- *f_value* tương đương với f(x) = h(x) + g(x) - tổng chi phí cho A* search, dùng để đánh giá đường đi

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

```
class NodeForAStar:
    def __init__(self, state, parent=None, action=None, g_value=0, f_value=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.g_value = g_value
        self.f_value = f_value
```

Từ trạng thái hiện tại A* xây dựng tất cả các đường đi có thể đi dùng hàm ước lượng khoảng cách (hàm Heuristic) để đánh giá đường đi tốt nhất có thể đi. Tùy theo mỗi dạng bài khác nhau mà hàm Heuristic sẽ được đánh giá khác nhau. A* luôn tìm được đường đi ngắn nhất nếu tồn tại đường đi như thế.

Hàm g ở trong bài toán này chính là độ sâu của một node trong cây tìm kiếm

Và sau khi tìm hiểu, nhóm đã chọn ra được hàm heuristic h cho Water sort:

h(S) = c(S) + j(S) với S; với S là một trạng thái - là một list các chuỗi, mỗi chuỗi biểu thị cho một ống. (Ở đây ta dùng chuỗi để mô phỏng stack.)

Trong đó

- **c(S)** là chi phí ước tính tối thiểu để chuyển các cột chất lỏng đồng màu không nằm ở đáy qua các ống khác.
  Cách tính: tổng của các c(cup), mỗi c(cup) được tính như sau: mỗi cup được phân rã thành các chuỗi con liên tiếp, mỗi chuỗi con chỉ gồm kí tự (màu) giống nhau. Khi đó c(cup) bằng số chuỗi con trừ đi 1
  Ý nghĩa c(cup): giả sử cup 'XDDV' phân rã thành 'X', 'DD', 'V', thì khi đó cần tối thiểu 2 bước để chuyển 'V', và 'DD' qua cup khác.

- **j(S)** là chi phí ước tính tối thiểu để gom các cột chất lỏng đồng màu có cùng màu với nhau nằm rải rác ở đáy các ống khác nhau.
  Lập một list A gồm màu dưới bottom của mỗi cup, cup nào rỗng thì không tính. Khi đó, j(S) bằng số phần tử của A trừ đi số phần tử phân biệt của A.
  Ý nghĩa: Giả sử có hai cup đều có chung màu ở đáy, thì khi đó cũng cần ít nhất một bước để xử lý đưa cột màu ở bottom cup này qua cốc kia. Nếu có 3 cup có cùng màu ở bottom thì cũng cần ít nhất 2 bước để xử lý từ đó dễ dàng quy nạp lên và ta có cách tính như trên.

Hàm heuristic được hiện thực trong code như sau

```
def h_function(self, state):
        h_value = 0

        for cup in state:
            cup_value = 0
            cur_index = 0

            while cur_index < len(cup) - 1:
                if cup[cur_index] != cup[cur_index + 1]:
                    cup_value += 1
```

```
                cur_index += 1
            h_value += cup_value

        list_color__at_bottom = []
        for cup in state:
            if len(cup) > 0:
                list_color__at_bottom.append(cup[0])
        h_value += len(list_color__at_bottom) - len(set(list_color__at_bottom))
        return h_value
```

Chú ý: Do nhóm nhận thấy thuật toán **Weighted A*** với trọng số của hàm h là 3 (tức là $f(n) = 3 \cdot h(n) + g(n)$) đem lại kết quả nhanh hơn so với thuật toán A* dùng công thức $f(n) = h(n) + g(n)$ như thông thường. Do đó, nhóm quyết định dùng thuật toán 3-Weighted A* (sự kết hợp của thuật toán tham lam và best-first, nhưng ưu tiên yếu tố tham lam).

Thuật toán này đem lại kết quả xấp xỉ tối ưu, trong khi thuật toán A* thông thường luôn đảm bảo mang lại lời giải tối ưu. Thế nhưng thời gian của **3-Weighted A*** trong bài toán này thì nhanh vượt trội với A*. Nguyên nhân là vì hàm chi phí g ở bài toán này tăng tiến rất chậm so với hàm h, do đó giá trị h nên được ưu tiên hơn. Như vậy, ta chỉ hi sinh phần ít độ tối ưu của lời giải để đổi lấy lợi ích về mặt thời gian.

3 Kết quả demo

3.1 Hướng dẫn dùng GUI

- **Cài đặt môi trường và thư viện hỗ trợ**

  Yêu cầu môi trường : Python 3.8 trở lên, Pip Package version mới nhất.

  Để cập nhật pip package mới nhất:

  ```
  > py -m pip install -upgrade pip
  ```

  Cài đặt thư viện cần thiết: **queue, pygame, screeninfo**

- **Thực thi chương trình**

  Vào thư mục Watersort và mở IDE đã cài môi trường, thực thi dòng lệnh trên Terminal

  ```
  > python main.py testcase-number algorithm-option
  ```

Trong đó:

- **main.py:** là file python thực thi chương trình nằm trong thư mục Watersort
- **testcase-number:** là đường dẫn đến file testcase ứng với số được nhập trong thư mục ./testcase
- **algorithm-option:** là giải thuật muốn kiểm tra, gồm các giá trị: **DFS, BFS, A***

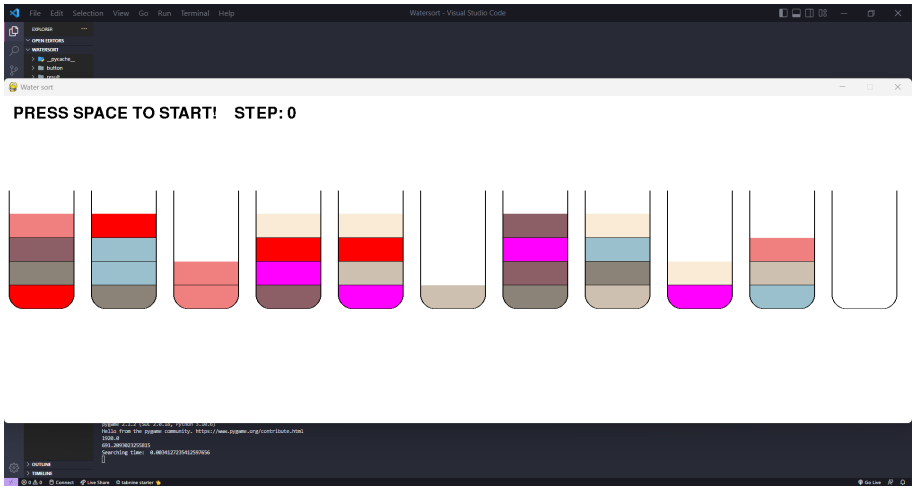
GUI chỉ demo từ testcase0 đến testcase9, do quá nhiều màu sẽ khiến mắt người quá phân biệt. Các testcase sau (tương đối nhiều màu) sẽ được in kết quả ở dạng file text, đặt tại folder result ở cùng cấp với file main.py. Ngoài ra, GUI đã được điều chỉnh để có tính **responsive**, chiều dài và chiều rộng GUI đã được chỉnh để tỉ lệ với màn hình của người xem, chi tiết cách điều chỉnh trong file Cups_UI.py.

3.2 Kết quả demo

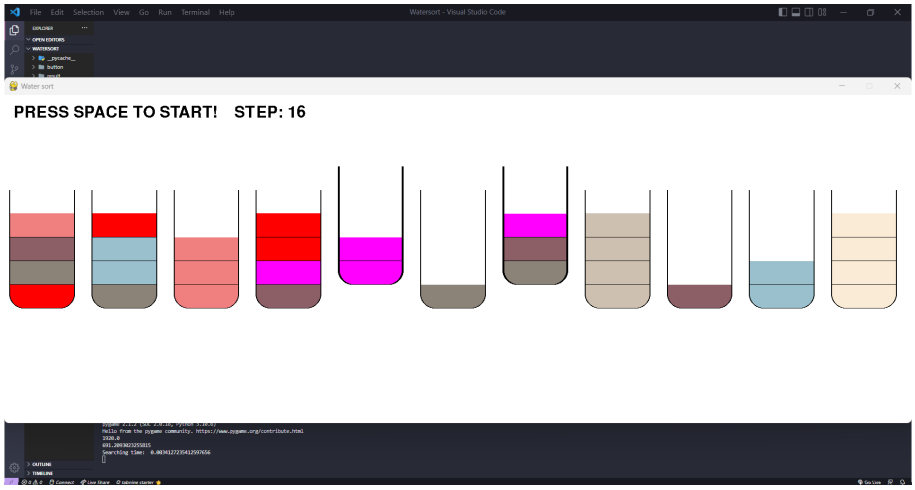
Ví dụ: Muốn chạy testcase2 có trong thư mục ./testcase bằng giải thuật DFS, ta thực hiện dòng lệnh

```
> python main.py 2 DFS
```

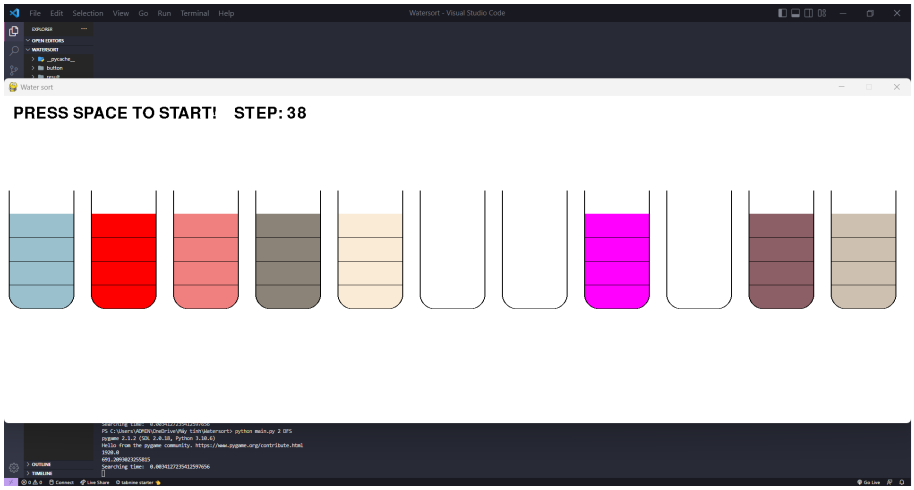
Sau khi thực thi dòng lệnh, màn hình PyGame sẽ hiển thị lên giao diện game Watersort tương ứng với testcase2 đã chọn. Nhấn Space để bắt đầu



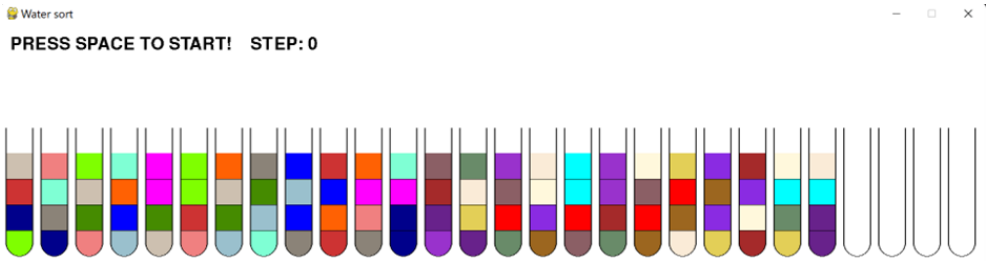
(a) Giao diện game khi vừa bắt đầu với Step = 0



(b) Game đang trong quá trình thực hiện đi đến kết quả



(c) Trò chơi kết thúc với Step = 38



(d) Giới hạn của GUI là tại testcase 9, vì quá nhiều màu (hơn 30 màu) sẽ khiến mắt người khó phân biệt màu

Mặc dù PyGame hiển thị được giao diện game tường minh rõ ràng, tuy nhiên nhóm vẫn chưa đủ thời gian để có thể tìm cách đưa số cốc phù hợp xuống các dòng bên dưới như ở trong giao diện như game Watersort khi có số cốc quá lớn (>25 cốc), các cốc hiển thị trên màn hình sẽ bị nhỏ lại và khó quan sát hơn, vì vậy mà nhóm đã đề ra giải pháp phù hợp hơn đối với những testcase khó bằng cách đưa kết quả in ra của chúng vào 1 file text khác được hiển thị trong thư mực result.

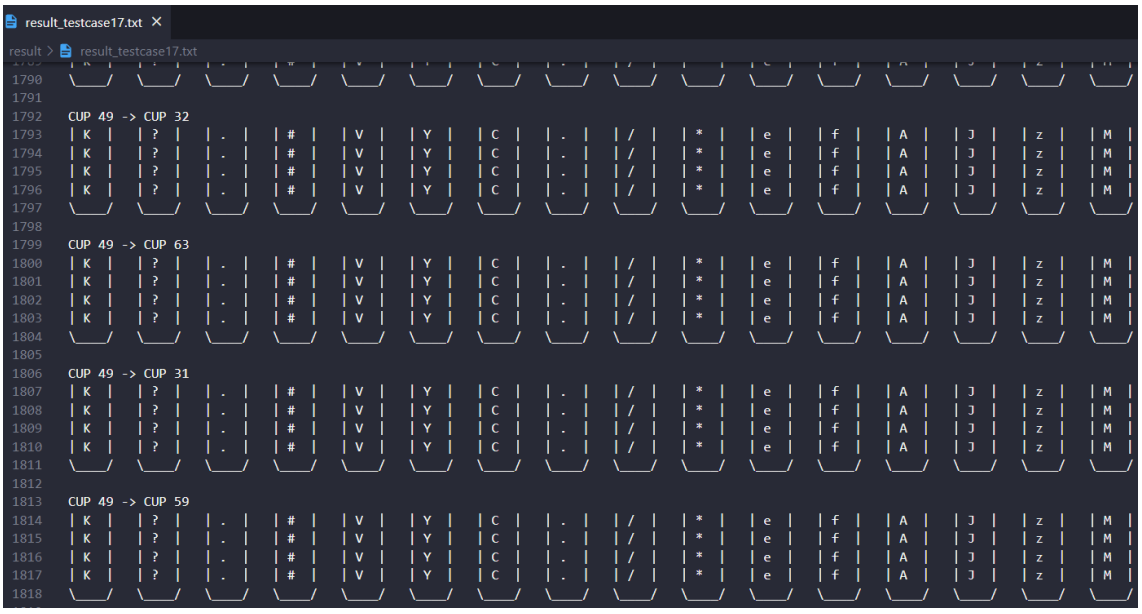
Ví dụ: Khi ta thực hiện với testcase17 bằng giải thuật A* Search, chương trình sẽ không hiển thị màn hình PyGame mà thay vào đó là lưu lại vào file **result_testcase17.txt** ở thư mục **result**. Ta thực hiện dòng lệnh sau để xem kết quả

```
> python main.py 17 A*
```

```
PS C:\Users\ADMIN\OneDrive\Máy tính\Watersort> python main.py 2 DFS
pygame 2.1.2 (SDL 2.0.18, Python 3.10.6)
Hello from the pygame community. https://www.pygame.org/contribute.html
1920.0
691.2093023255815
Searching time:  0.9253320693969727
Open result/result_testcase17.txt to check result
```

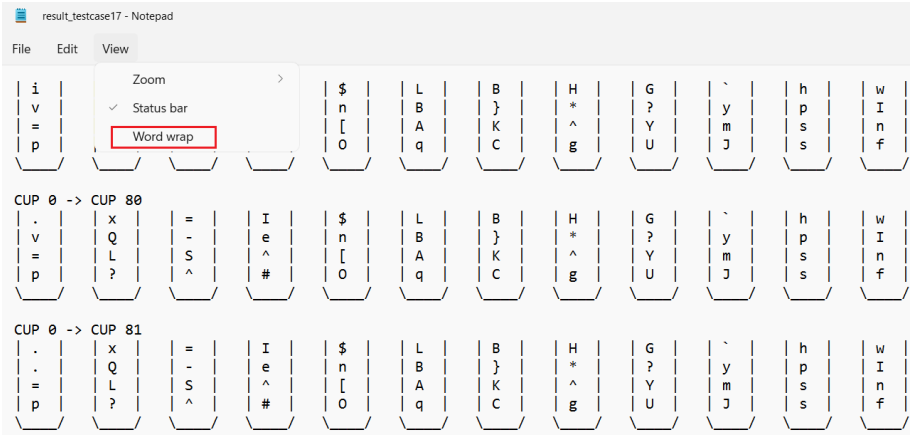
(a) Kết quả in ra từ Terminal

(b) File result_testcase17.txt trên IDE



(c) Kết quả của testcase17

Chú ý: Nếu muốn xem kết quả của result_testcase17.txt bằng Notepad, để ý bỏ tick Word wrap để xem đầy đủ các cốc bằng cách di chuyển thanh scroll ngang



3.3 Đánh giá hiệu năng

Để đánh giá được hiệu năng của từng giải thuật, nhóm đã tiến hành đo lường thời gian thực thi và sự tiêu tốn bộ nhớ trong mỗi test case

Mã nguồn được chạy để kiểm tra thời gian thực thi được thực hiện trên máy ảo Oracle VM VirtualBox chạy hệ điều hành Ubuntu 64-bit được cài đặt trên máy tính Windows 11 64-bit, AMD Ryzen 7 5700U, CPU 1.80 GHz, RAM 8 GB (6.82 GB usable)

Bằng cách import thêm các package time và resource. Nhóm đã tìm được cách lấy ra thời gian thực thi và sự tiêu tốn bộ nhớ

Ví dụ: Với testcase đơn giản như testcase3.txt chạy bằng giải thuật A*. Ta sẽ lấy số liệu trên Ubuntu bằng cách vào thư mục Water sort bao gồm tất cả source code mà nhóm đã cung cấp, chạy lệnh trên terminal chứa địa chỉ thư mục đó. Ta thực hiện lệnh sau:

```
$ python3 ./measure.py 3 DFS
```

Thực hiện các bước nhập test case giống với cách dùng GUI, ta có kết quả như sau

```
dinhtuan@dinhtuan-VirtualBox:~/Desktop/Watersort$ python3 ./measure.py 3 DFS
testcase3.txt

Num explored: 111
Time: 0.023 secs
Memory usage: 9540.0 KBytes
```

Hình 1: Mô tả cách hiện thực lấy số liệu

Để linh hoạt trong quá trình đo đạc, nhóm tạo ra một file **bash script** có chứa nhiều vòng lặp các command để đo toàn bộ các một testcase, mỗi testcase đo nhiều lần trong đúng một command (chạy file .sh). Ngoài ra, để mang tính chính xác cao trong khi đo số liệu, nhóm thực hiện chiến lược đo **10** lần rồi tính trung bình để kết quả đáng tin cậy hơn.

3.3.1 Bảng số liệu cho các giải thuật

Ở mỗi thuật toán tìm kiếm, ta đặt ra thêm điều kiện nếu thuật toán duyệt đến 45.000 node mà vẫn chưa ra kết quả thì trả về "No solution". Vì qua quá trình thử nghiệm, nhóm nhận thấy nếu thuật toán vẫn không đưa ra lời giải khi duyệt đến node thứ 45000 thì thuật toán sẽ chạy rất lâu, có thể lên tới 30 phút. Các testcase mà nhóm thiết kế thường có số cup dao động từ **7** đến **92** cup, tăng tiến dần dần từ testcase0 đến testcase19.

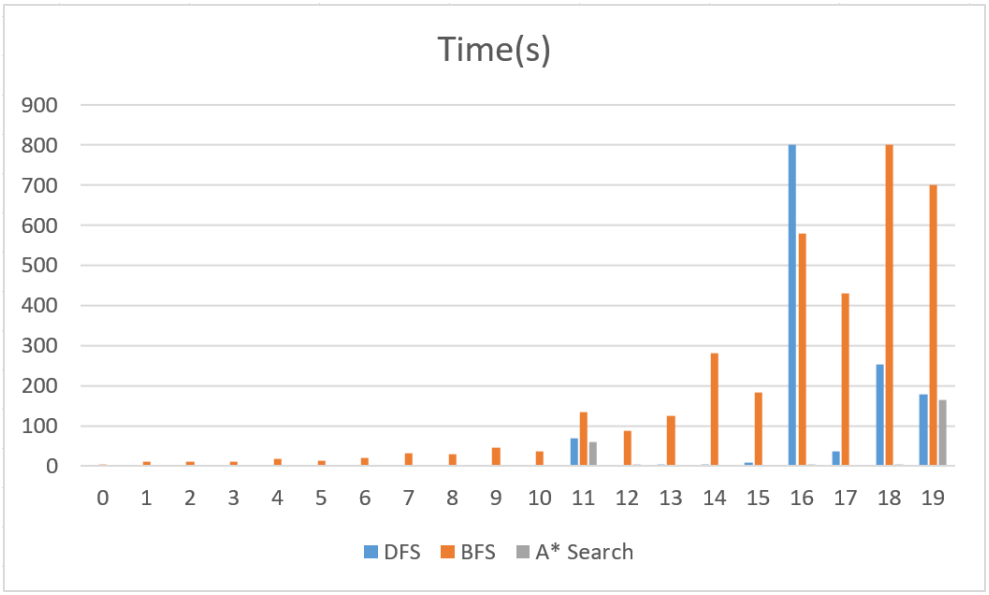
Bảng số liệu về thời gian và sự tiêu tốn bộ nhớ cho từng giải thuật được trình bày như sau:

Testcase	DFS			BFS			A* Search		
	Time	Number explored	Memory (Kbyte)	Time	Number explored	Memory (Kbyte)	Time	Number explored	Memory (Kbyte)
testcase0.txt	0.002s	22	9564	4.206s	No solution	61336	0.002s	17	9556
testcase1.txt	0.006s	34	9564	10.079s	No solution	188388	0.006s	22	9560
testcase2.txt	0.008s	39	9596	10.256s	No solution	164296	0.007s	25	9556
testcase3.txt	0.023s	111	9656	9.989s	No solution	85392	0.039s	168	9592
testcase4.txt	0.014s	48	9624	17.412s	No solution	253240	0.017s	34	9576
testcase5.txt	0.014s	63	9792	12.493s	No solution	209512	0.017s	48	9688
testcase6.txt	0.043s	97	10058	20.523s	No solution	229532	0.049s	64	10008
testcase7.txt	0.044s	98	10132	31.727s	No solution	558852	0.057s	63	10052
testcase8.txt	0.048s	117	10212	30.453s	No solution	558336	0.064s	67	10232
testcase9.txt	0.095s	123	10278	46.965s	No solution	564844	0.108s	75	10224
testcase10.txt	0.595s	829	10448	35.693s	No solution	133764	0.480s	603	10176
testcase11.txt	68.338s	No solution	27298	134.097s	No solution	1479574	58.881s	34833	24508
testcase12.txt	1.728s	1629	11316	86.914s	No solution	1080600	4.344s	3701	10352
testcase13.txt	4.233s	2716	12008	124.953s	No solution	1436200	2.285s	1327	11160
testcase14.txt	4.234s	1764	12016	281.65s	No solution	1526486	2.175s	822	11532
testcase15.txt	8.873s	4199	13632	183.26s	No solution	1487336	1.022s	417	11348
testcase16.txt	Killed	Recursion error	>32256	578.58s	No solution	>1052672	5.006s	507	20572
testcase17.txt	37.689s	5949	20748	431.4s	No solution	>1052672	2.361s	260	15996
testcase18.txt	254.202s	42131	31344	>600s	No solution	>1052672	3.773s	523	15304
testcase19.txt	177.744s	30940	27512	>600s	No solution	>1052672	165.652s	28439	22662

Một số giải thích:

- Cột Time là thời gian thực thi của giải thuật (đơn vị giây).
- Cột Number explored là số node đã được khai phá trong quá trình tìm kiếm lời giải.
- Cột Memory là bộ nhớ tiêu tốn để sử dụng cho giải thuật (đơn vị kilobyte).

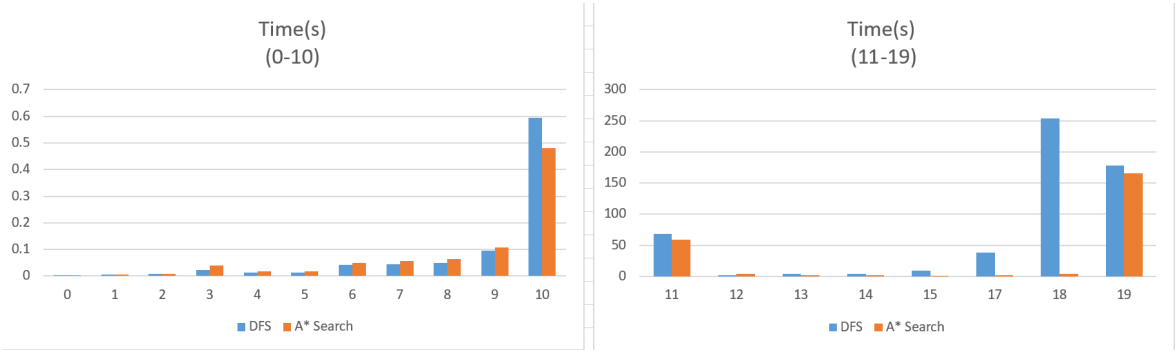
3.3.2 Các đồ thị trực quan so sánh các giải thuật



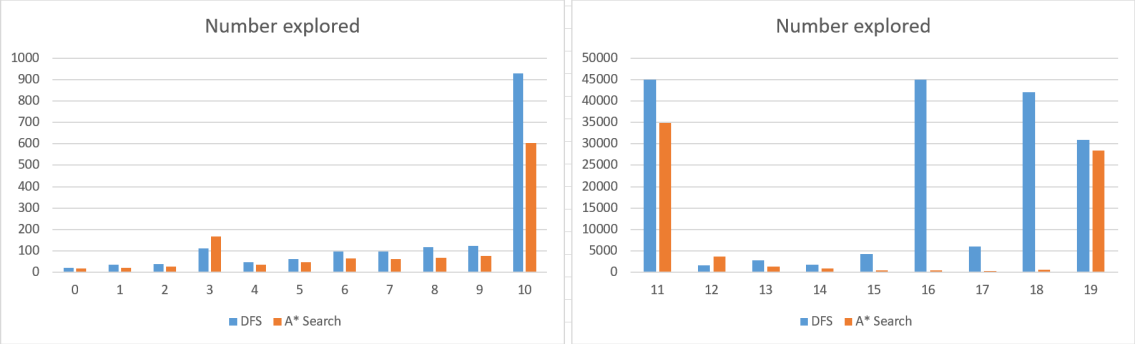
Hình 2: So sánh thời gian chạy của 3 giải thuật

Qua bảng số liệu về thời gian và bộ nhớ tiêu tốn, nhóm nhận thấy giải thuật BFS không nên được đưa vào đồ thị so sánh trực quan vì hầu như không cho ta kết quả mong muốn cùng với lượng bộ nhớ tiêu tốn nhiều. Nhóm chúng em chỉ đi sâu vào đồ thị so sánh 2 giải thuật còn lại

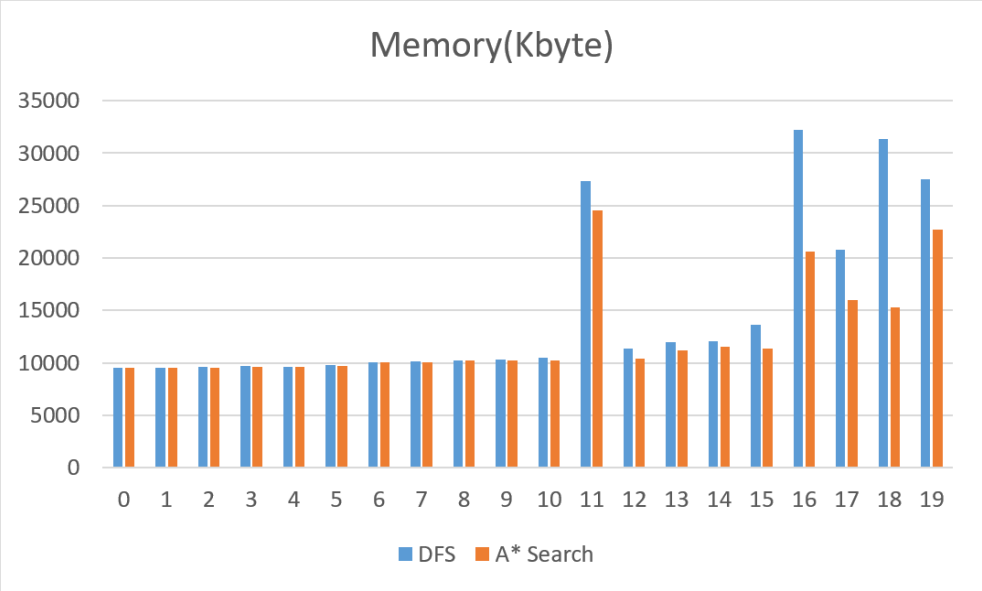
Về thời gian và số node khai phá, nhóm chia thành 2 đồ thị so sánh từ **testcase 0 - 10** và từ **testcase 11-19** để có thể thấy rõ hơn về sự khác biệt trong thời gian và số node khai phá



Hình 3: So sánh thời gian chạy trong DFS và A* Search



Hình 4: So sánh số node khai phá trong DFS và A* Search



Hình 5: So sánh bộ nhớ tiêu tốn của DFS và A* Search

3.3.3 Đánh giá giải thuật

Qua các đồ thị trực quan và số liệu đo lường, nhóm chúng em cơ bản nhận xét các giải thuật như sau:

	Thời gian & bộ nhớ tiêu tốn	Số trạng thái phải duyệt	Kết quả
A* Search	ít nhất	ít nhất	tốt nhất
Depth-First-Search	trung bình	trung bình	trung bình
Breadth-First-Search	nhiều nhất	nhiều nhất	kém nhất

Đánh giá chi tiết từng giải thuật:

- BFS
  - Dựa vào bảng kết quả đánh giá, ta thấy hầu hết các bài toán từ đơn giản đến khó, giải thuật BFS đều không thể mang lại kết quả của bài toán và luôn vượt qua ngưỡng khai phá cho phép (45000 node) cùng với lượng bộ nhớ tiêu tốn cực kì lớn. Giải thuật duyệt qua các node theo chiều rộng nên đã khiến cho việc tìm kiếm diễn ra lâu hơn.
  - Trong trường hợp có nhiều cốc và các cốc có số lượng màu chứa tối đa càng nhiều. Việc sử dụng BFS đa số không tìm được đích hoặc dừng lại ở vị trí chưa phải đích và có thể sẽ khiến cho lời giải bài toán chương tình tự hủy (Killed) vì thời gian chạy quá dài.
- DFS
  - Dựa vào bảng kết quả đánh giá, ta thấy được việc sử dụng giải thuật DFS là mang lại hiệu quả tốt hơn BFS vì giải thuật sẽ tìm kiếm ra kết quả bằng cách duyệt theo chiều sâu từ các bước đi trước đó. Bài toán Water sort có thể có nhiều trạng thái mục tiêu nằm ở các nhánh của cây làm cho trạng thái hiện tại ngày càng gần hơn với trạng thái mục tiêu
  - Tuy nhiên ở một số trường hợp khi lời giải của bài toán nằm ở quá sâu trong cây tìm kiếm, giải thuật DFS cứ liên tục tìm kiếm ngày càng sâu và không có giới hạn khiến cho việc xảy ra tình trạng độ sâu đệ quy tối đa vượt quá so với so sánh
  - DFS đã giải quyết được 1 phần về hiệu quả so với việc sử dụng BFS khi ở trong những trường hợp khó hơn, DFS có thể giúp cho ta tìm được lời giải với thời gian và bộ nhớ tiêu tốn ở mức trung bình. Tuy nhiên khi số lượng cốc và số lượng màu tăng lên, việc sử dụng DFS đã có phần khó khăn hơn trong việc tìm kiếm, thậm chí vượt qua ngưỡng khai khá cho phép
- A* Search
  - Việc chọn hàm lượng giá đúng đắn và hợp lý cho A* search đã khiến cho bài toán trở nên hiệu quả hơn đáng kể vì giải thuật sẽ chọn lấy node có chi phí thấp nhất để tìm duyệt tiếp trạng thái mới, làm cho số trạng thái cần khai phá giảm đi đáng kể
  - Để ý rằng ở bài toán Water sort đơn giản, luôn có 1 số trường hợp ta nhận thấy rằng hiệu năng và số node cần được khai phá của DFS thực hiện tốt hơn A* mặc dù không chênh lệch quá lớn, tuy nhiên khi số lượng cốc và màu ngày càng tăng, giải thuật A* đã cho thấy sự ưu việt về hiệu năng hơn hẳn so với DFS và BFS. Nhưng nếu số cốc rỗng của bài toán càng ít đi thì việc giải bài toán trở nên khó khăn hơn rất nhiều.

Phần II
Bloxorz Problem

1 Mô tả luật trò chơi và input file

1.1 Luật trò chơi

- 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 (Nhóm sử dụng 9 stage đầu của game để minh họa cho các giải thuật được sử dụng). Người chơi di chuyển khối block bằng cách sử dụng các phím mũi tên, và không để bị rơi ra khỏi map (bản đồ).

- Các bridges (cầu) và Switches (công tắc) được đặt trong nhiều levels của trò chơi. Switches được kích hoạt khi bị block đè lên.

- Có 2 loại swithches: Heavy X-shaped và Soft O-shaped. Soft switches được kích hoạt khi có bất kì phần nào của bock đè lên nó, trong khi đó Heavy switches yêu cầu nhiều áp lực hơn với việc block phải đứng trên switches để kích hoạt. Khi được kích hoạt, mỗi swiches sẽ thể hiện một động thái riêng. Một số sẽ chỉ có chức năng bật bridge, một số chỉ đóng bridge, số còn lại có thể luân phiên thực hiện cả 2 chức năng.

- 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 áp lực 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.

1.2 Mô tả input file

Test file là một test case dưới dạng file text cung cấp thông tin về shape của map (chiều dài, chiều rộng), vị trí ban đầu của block, map của bài toán, thông tin của các switch trong map.Tùy theo mỗi test case dễ hay khó mà trạng thái khởi đầu của bài toán sẽ khác nhau. Một số test file đã được nhóm chuẩn bị sẵn được đặt trong thư mục test_case/. Ví dụ stage2.txt sẽ có nội dung như sau:

```
6 15
4 1
0 0 0 0 0 0 1 1 1 1 0 0 1 1 1
1 1 1 1 0 0 1 1 X 1 0 0 1 G 1
1 1 O 1 0 0 1 1 1 1 0 0 1 1 1
1 1 1 1 0 0 1 1 1 1 0 0 1 1 1
1 1 1 1 0 0 1 1 1 1 0 0 1 1 1
1 1 1 1 0 0 1 1 1 1 0 0 0 0 0
O 1 2 2 4 4 4 5
X 1 1 8 4 10 4 11
```

- Dòng đầu tiên mô tả map có chiều rộng là 6 và chiều dài là 15
- Dòng thứ 2 mô tả vị trí ban đầu của block là (4,1). Với quy ước (x,y) thể hiện vị trí hàng thứ x và cột thứ y trong ma trận
- Từ dòng thứ 3 đến 8, biểu diễn map của game, với các quy ước được định nghĩa như sau:
  + Kí tự **'1'**: biểu diễn cho ô gạch
  + Kí tự **'0'**: Biểu diễn không có gạch (những ô trống trong map)
  + Kí tự **'G'**: Vị trí mà block cần phải đứng lên để hoàn thành trò chơi (GOAL)
  + Kí tự **'C'**: những viên gạch màu da cam
  + Kí tự **'X'**: Switch loại Heavy X-shaped
  + Kí tự **'O'**: Switch loại Soft O-shaped
  + Kí tự **'T'**: Switch teleport
- Từ dòng thứ 9 trở đi (theo hình ảnh được minh họa) là thông tin chi tiết của các switches được quy ước như sau:
  + Kí tự đầu tiên thể hiện loại switches được dung trong map (**X**: loại Heavy X-shaped, **O**: loại Soft O-shaped, **T**: loại teleport)
  + Kí tự kế tiếp thể hiện chức năng của từng loại switch:
    - Nếu là loại 1: khi được kích hoạt switch có thể luân phiên đóng mở các bridge
    - Nếu là loại 2: khi được kích hoạt switch chỉ có thể đóng bridge
    - Nếu là loại 3: Khi được kích hoạt, switch chỉ có thể mở bridge

    (Nếu switch là teleport: được mặc định kí tự thể hiện chức năng switch là **'1'**).
  + 2 kí tự tiếp theo biểu diễn vị trí của switch trong map. Với quy ước (x,y) thể hiện vị trí hàng thứ x và cột thứ y trong ma trận
  + Cứ mỗi cặp kí tự kế tiếp thể hiện vị trí của các bridge phụ thuộc đến loại switch đang xét
- Ví dụ: X 1 1 8 4 10 4 11
  Cho biết switch X loại 1 (có thể luân phiên đóng mở khi kích hoạt), vị trí trong map là (1,8), mỗi lần kích hoạt sẽ ảnh hưởng tới các vị trí trong map là (4,10) và (4,11)

2 Ý tưởng hiện thực

2.1 Thiết kế không gian trạng thái để duyệt DFS

Phần này không trình bày chi tiết về giải thuật DFS mà chỉ tập trung thiết kế không gian trạng thái, vì giải thuật này đã nói rõ ở bài Water Sort. Nhóm không chọn giải thuật BFS để khảo sát vì đối với các game khám phá bản đồ với dạng bản đồ hữu hạn, không quá lớn như thế này, giải thuật DFS luôn tỏ ra hiệu quả về mặt thời gian hơn so với BFS.

2.1.1 Không gian trạng thái

Là tập hợp các trạng thái có thể có của trò chơi. Một node trong cây tìm kiếm tương ứng với 1 trạng thái được định nghĩa bao gồm các thuộc tính của trạng thái: *tọa độ của block, map, locationSwitch, isSplit, state, parent, action*
Trong đó:

- *tọa độ của block* được lưu bằng danh sách các tupple (x,y) là các tọa độ của block, nếu block đứng thì có một tọa độ, nếu block nằm thì có hai tọa độ.
- *map* là biến lưu trữ trạng thái của map, lưu thuộc tính này có một điểm yếu rất lớn là tốn bộ nhớ. Nhưng phần bộ nhớ tốn thêm này là để lưu các biến "reference", chứ không phải một bản copy thật sự của cái map. Thuộc tính này sẽ giúp ích cho phần render GUI cho trò chơi.
- *locationSwitch* là danh sách tọa độ các công tắc trên map.
- *isSplit* là biến lưu trữ trạng thái tách rời của Block, bằng True Block đang tách rời, bằng False nếu Block đang liền nhau.
- *state* là biến lưu trữ phương của block, nhận giá trị 1 nếu block đang đứng, 2 nếu block đang nằm hướng theo trục x, 3 nếu block đang nằm hướng theo trục Y.
- *parent* là tham chiếm tới trạng thái cha của trạng thái hiện tại.
- *action* là hành động cần thực hiện.

2.1.2 Trạng thái ban đầu

Trạng thái ban đầu là trạng thái đầu tiên của block khi stage bắt đầu, **tọa độ của Block** được xác định từ dòng thứ hai của map file được truyền vào chương trình. **map** là ma trận các Box ban đầu. **locationSwitch** là danh sách các Switch. **isSplit** là False. **action** là None. **parent** là None.

2.1.3 Trạng thái mục tiêu

Trạng thái mục tiêu là trạng thái mà block đứng trên điểm đích. Khi đó tọa độ của Block phải có duy nhất một tọa độ và tọa độ đó trong ma trận là Box có ký tự 'G'.

2.1.4 Các bước chuyển hợp lệ

Hành động mà Block có thể thực hiện từ một trạng thái là **lăn trái, lăn phải, lăn về trước, lăn về sau**. Tùy theo vị trí mà block đang ở trong trạng thái mà chỉ có một số hành động có thể thực hiện được, thí dụ như là hình dưới thì chỉ có hành động là lăn trái, lăn phải, lăn về trước.
Trong mỗi hàm **moveUp, moveDown, moveLeft, moveRight**, kiểm tra dáng đứng của cái thùng rồi tính toán toạ độ tiếp theo và kiểm tra tính hợp lệ của tọa độ tiếp theo:
Thí dụ như hàm **moveUp**:

- Nếu Block đang liền nhau:
  - Nếu block đang đứng, gọi (x,y) là tọa độ hiện tại, tọa độ tiếp theo của nó sẽ là: [(x - 2,y), (x - 1,y)]

- Nếu block đang nằm theo trục x, gọi [(x_0, y_0), (x_1,y_1)] là tọa độ hiện tại của Block, tọa độ tiếp theo của nó sẽ là: [(x_0 - 1,y_0), (x_1 - 1,y_1)]
- Nếu block đang nằm theo trục y, gọi [(x_0, y_0), (x_1,y_1)] là tọa độ hiện tại của Block, tọa độ tiếp theo của nó sẽ là: (min(x_0, x_1) - 1,y_0)

- Nếu Block đang rời nhau:
  - Nếu đang thao tác với hình lập phương có id = 1 của Block, gọi [(x_0, y_0), (x_1,y_1)] là tọa độ hiện tại của Block, tọa độ tiếp theo của nó sẽ là [(x_0 - 1,y_0), (x_1,y_1)]
  - Nếu đang thao tác với hình lập phương có id = 2 của Block, gọi [(x_0, y_0), (x_1,y_1)] là tọa độ hiện tại của Block, tọa độ tiếp theo của nó sẽ là [(x_0,y_0), (x_1 - 1,y_1)]

Sau đó kiểm tra các tọa độ tiếp theo có hợp lệ hay không, nếu không thì trả về None, nếu có thì render lại map, tạo ra trạng thái mới và trả về trạng thái đó.

2.2 Thiết kế giải thuật di truyền

2.2.1 Mã giả tổng quan cho giải thuật

Dưới đây là mã giả tổng quan cho giải thuật:

```
Khởi tạo quần thể ban đầu (initialize).

while (chưa tìm thấy lời giải từ quần thể hiện tại):

    Chọn X cá thể bố mẹ tiềm năng theo độ fitness để sinh cá thể mới (reproduce).

    Lai tạo các cặp ngẫu nhiên từ tập bố mẹ tiềm năng (cross-over).

    Chọn Y cá thể con vừa sinh ra để đột biến (mutation).

    Đưa toàn bộ số con vào quần thể hiện tại, tạo thành quần thể mới.

    Xếp hạng (rank selection) quần thể mới theo fitness, chỉ giữ lại cá thể
    rank cao (refine).
```

Một số điểm đáng chú ý:

- Số cá thể chọn làm bố mẹ tiềm năng là X = kích thước quần thể * tham số x chọn trước.
- Số con đem đi đột biến là Y = số con vừa sinh ra * tham số y chọn trước.
- Mỗi cặp bố mẹ sẽ sinh ra hai cá thể con.
- Kích thước quần thể lúc khởi tạo là hằng số chọn trước (trong source code chọn là 1000). Kích thước của quần thể sau mỗi bước rank selection luôn không đổi và bằng với hằng số này.
- Vì các cá thể con sinh ra có thể không đủ tốt, ta bổ sung thêm bước combine và refine để đảm bảo fitness trung bình của quần thể không giảm.

2.2.2 Mô tả cấu trúc cá thể

Mỗi cá thể trong quần thể sẽ được mô tả bằng một chuỗi kí tự lấy từ tập kí tự $\{R, r, L, l, U, u, D, d\}$, ví dụ "$RrDULL$". Chuỗi kí tự này ứng với chuỗi hành động chuyển trạng thái của cái thùng từ vị trí ban đầu. Trong đó:

- R, r là kí tự biểu thị cho hành động lăn qua phải.
- L, l là kí tự biểu thị cho hành động lăn qua trái.
- U, u là kí tự biểu thị cho hành động lăn lên trên.
- D, d là kí tự biểu thị cho hành động lăn xuống dưới.
- Trong trường hợp cái thùng không bị phân tách, ta không phân biệt chữ hoa chữ thường, tức là L và l có ý nghĩa tương đương nhau.
- Trong trường hợp cái thùng bị phân tách (thành hai khối con có ID là 0 và 1), thì kí tự chữ hoa là hành động áp dụng cho khối con có ID là 0, chữ thường áp dụng cho khối có ID là 1. Ví dụ: R là lăn phải khối con ID bằng 0, r là lăn phải khối con ID bằng 1.
- Trong chuỗi, sẽ xuất hiện những kí tự **nhiễu** tượng trưng cho những bước đi khiến cho thùng rơi xuống vực. Khi đó ta coi những kí tự này là thừa và bỏ qua chúng.
- Các chuỗi không nhất thiết phải có độ dài bằng nhau, tức là chúng có tính chất **variable length**.

2.2.3 Ý tưởng hàm fitness

Ta sẽ chấm điểm fitness của mỗi cá thể (chuỗi hành động) dựa trên vị trí cuối cùng của cái thùng sau khi thực hiện chuỗi hành động, những ô đặc biệt đã đi qua. Những heuristic được dùng để chấm điểm fitness như sau:

- Sau khi thực hiện chuỗi hành động, khoảng cách từ vị trí cuối cùng của cái thùng càng gần vị trí goal thì cá thể có độ fitness càng cao. Khoảng cách này không thể tính bằng khoảng cách "Manhattan" , hãy tưởng tượng hình chữ 'U', hai điểm ở hai đỉnh đầu chữ 'U' tưởng như rất gần thông qua khoảng cách Mahntann, nhưng đường đi giữa chúng lại là xa nhất. Loại khoảng cách mà ta dùng phải là đường đi ngắn nhất nối từ vị trí cuối cùng đến goal thông qua các ô trên bản đồ, tính cả các ô bridge chưa mở. Độ dài đường đi này ta tạm gọi là "**magic distance**". Ta sẽ lập một bảng để tính toàn bộ magic distance của các ô tới goal bằng phương pháp quy hoạch động kết hợp với duyệt loang các ô trong bản đồ. Ta chỉ làm điều này đúng một lần duy nhất, là trước khi khởi tạo quần thể.
- Nếu trên đường đi, cái thùng kích hoạt nhiều switch để mở cầu (bridge) hoặc phân tách thì chuỗi hành động này có độ fitness cực kì cao. Điều này là dễ hiểu vì phải qua tất cả các cầu có trên bản đồ mới có thể tới được đích. Chú ý, độ fitness sẽ giảm mạnh nếu cái thùng khiến cho switch O, X đóng cầu.
- Nếu trên đường đi, cái thùng đi qua nhiều ô bridge hoặc nhiều ô màu cam thì (mỗi ô chỉ tính 1 lần), thì chuỗi hành động có độ fitness cực kì cao.
- Nếu tại vị trí cuối cùng, thùng đã kích hoạt mở cầu mà chưa đi qua thì độ fitness của cá thể sẽ giảm (bị phạt) dựa trên khoảng cách từ cái thùng đến bridge, tức là càng xa bridge thì bị phạt càng nhiều.

- Ở trạng thái cuối cùng, nếu cái thùng đang ở trạng thái phân tách và rất gần đích, thì khi đó chuỗi hành động sẽ có độ fitness càng cao (được điểm thưởng) nếu như khoảng cách giữa hai khối con càng gần. Nói tóm lại, hai khối con nên nhập vào nhau nếu chúng đều ở rất gần goal, còn nếu cả hai đều xa goal thì tạm thời chưa nên nhập lại.

Từ những heuristic như trên, ta có **công thức** tính fitness là như sau:

$$fitness(I) = [k - g_1(S)]^2 \cdot w_1 + [(k - g_2(S)]^2 \cdot w_2 + [k - g_3(S)]^2 \cdot w_3 + bonusScore(L)$$

Trong đó:

- S là trạng thái cuối cùng khi thực hiện chuỗi hành động tương ứng với chuỗi gene của cá thể I. L là tập hợp các vị trí ứng với bước đi tốt đã đi qua. Chú ý trong L chứa các vị trí cầu, switch, gạch cam. Các vị trí sinh ra từ bước đi làm đóng cầu sẽ không được tính trong tập hợp này. Các phần tử trong L là đôi một phân biệt.
- g_1 là hàm đo khoảng cách từ vị trí đứng của cái thùng đến goal, loại khoảng cách dùng là magic distance. Hàm này chỉ việc lấy kết quả từ bảng magic table đã tính sẵn.
- g_2 là hàm đo khoảng cách giữa hai khối con khi phân tách. Trường hợp khối không bị phân tách thì trả về giá trị là 0. Loại khoảng cách được dùng là manhattan distance.
- g_3 là hàm đo tổng khoảng cách giữa từ cái thùng đến những cây cầu đã được mở mà chưa đi qua, loại khoảng cách được dùng là manhattan distance.
- k là hằng số chặn trên của $g_1(S), g_2(S), g_3(S)$, lí do là có hàng số k và dấu trừ phía sau là vì ta muốn khoảng cách càng nhỏ thì điêm fitness càng cao. Hằng số k là số lớn nhất có trong bảng magic table.
- Hàm bonusScore là hàm tính điểm thường dựa trên các phần tử trong danh sách L. Trong đó:
  (a) Mỗi vị trí ứng với ô bridge sẽ được cộng 5 triệu điểm.
  (b) Mỗi vị trí ứng với ô gạch cam sẽ được cộng 3 triệu điểm.
  (c) Mỗi vị trí ngay sát goal sẽ được cộng 5 triệu điểm.
  (d) Mỗi vị trí ứng sinh ra sau bước kích hoạt cổng X, O (mở cầu) sẽ được cộng 3 triệu điểm.
  (e) Mỗi vị trí ứng sinh ra sau bước kích hoạt cổng T sẽ được cộng 3 triệu điểm.
- Khi chuỗi gene của cá thể I là lời giải cần tìm, $fitness(I)$ sẽ bằng 5 tỉ.
- Hệ số w_1, w_3 mà nhóm set trong source code là 1 và 100. Hệ số w_2 tùy thuộc vào giá trị khoảng cách tới goal $g_1(S)$, nếu giá trị này nhỏ hơn 15 thì w_2 bằng 3, ngược lại w_2 băng -0.5. Sở dĩ hệ số w_2 lớn như vậy là vì đi qua cầu là yếu tố quan trọng nhất để thắng một màn chơi, nếu màn chơi đó có các ô bridge. Yếu tố đi qua đầy đủ các cây cầu còn quan trọng hơn cả yếu tố khoảng cách tới đích.

Lí do xuất hiện số mũ là 2 ở các hạng tử là để đảm bảo càng gần đích, hoặc cầu, hoặc 2 khối cube con càng gần nhau, thì điểm fitness càng tăng mạnh. Để dễ hình dung xét dãy: $\{(k-30)^2, (k-29)^2, ...(k-1)^2, k^2\}$, ta thấy hiệu số giữa hai phần tử liên tiếp tăng dần theo chiều từ trái qua phải.

2.2.4 Ý tưởng các bước reproduce, cross-over, mutation

Reproduce

Ở bước này, ta sẽ chọn X (bằng size cá thể * tham số x chọn trước) cá thể bố mẹ tiềm năng để đưa vào 'parents pool'. Chiến lược chọn ở đây là ta dùng phương pháp '**fitness proportionate selection**', hay còn có tên gọi khác là '**roulette wheel selection**'. Trong source code có hiện thực hàm "roulette_wheel_selection" để thực hiện chiến lược này.

Ý nghĩa của phương pháp này là cá thể có độ fitness càng cao, xác suất được chọn của nó càng lớn. Xác suất được chọn của mỗi cá thể bằng điểm fitness của nó chia cho tổng fitness của quần thể.

Cross-over

Ở bước lai tạo, đa số giải thuật di truyền đều hay dùng phương pháp là **uniform-cross**. Nhưng qua quá trình thử nghiệm và phân tích, nhóm nhận thấy nếu lai tạo kiểu này, thì cá thể con không thừa hưởng những gì tốt nhất từ bố mẹ của nó, vì cấu trúc chuỗi gene này là một gene đặc biệt, kích thước chuỗi gene có thể tùy biến, mỗi gene trong chuỗi bị chi phối bởi gene trước nó. Điều này khác với chuỗi gene trong bài toán Knapsack, khi mà mọi gene trong chuỗi là độc lập, không phụ thuộc lẫn nhau.

Nhóm chuyển qua sử dụng **best-position-cross**, tức là dùng phương pháp "one-point cross" (một điểm cắt), nhưng điểm cắt ở đây phải là vị trí tốt nhất (best position) của cả bố và mẹ. Ý nghĩa ở đây, từ điểm tốt nhất thừa hưởng từ bố hoặc mẹ, hai cá thể con sẽ có những bước đi đột phá hơn. Điểm cắt tốt nhất, là hành động chuyển tới trạng thái gần với goal nhất trong quá trình thực hiện chuỗi hành động của cá thể.

Như vậy, khi thực hiện chiến lược này, toàn bộ các con sinh ra sẽ có độ fitness xấp xỉ những bố mẹ tiềm năng, hoặc thậm chí còn cao hơn.

Mutation

Như đã nói, đặc thù chuỗi gene bài toán Bloxorz này khác xa với nhiều chuỗi gene sử dụng cho các bài toán khác. Việc đột biến bằng cách chọn ngẫu nhiên một điểm trên chuỗi gene và đổi giá trị điểm này, sẽ không mang lại hiệu quả trong bài toán này. Thật vậy, giả sử ta có 1 chuỗi gene rất tốt là $(X_1, X_2, ...X_5, ...X_{100})$, nếu ta vô tình chọn ngẫu nhiên X_5, thì tức là ta đang phá đi chuỗi 95 gene đằng sau, vì gene liền sau sẽ bị ảnh hưởng bởi gene liền trước. Khi đó, xác suất để được một chuỗi gene "tệ hại" là vô cùng lớn, tức là việc đột biến là vô ích.

Để khắc phục tình trạng trên, nhóm chuyển qua chiến lược thêm gene (kí tự) vào cuối chuỗi gene của các cá thể được đem đi đột biến. SỐ gene được thêm là số ngẫu nhiên chọn từ khoảng 1 đến 20. Trong quá trình tính fitness của cá thể, có tích hợp với quy trình refine chuỗi gene, tức là ta xóa đi những gene nhiễu, vô nghĩa, tương ứng với những bước đi xuống vực, và những chuỗi genne con vô nghĩa tương ứng với các bước đi lòng vòng về lại trạng thái đã từng đi qua. Điều này sẽ giúp đảm bảo kích thước chuỗi gene của mỗi cá thể không quá dài, luôn bị chặn trên bởi một hằng số.

Combine và refine quần thể

Ở bước này, các cá thể con được hợp nhất với quần thể ban đầu, sau đó ta sẽ xếp hạng (rank

selection) quần thể theo fitness để chọn ra những cá thể tối ưu, mục đích là loại bỏ những cá thể yếu của quần thể trước, cũng như những cá thể con fitness không cao mới sinh ra.

2.2.5 Cấu trúc source code

Source code gồm các file:

- **dfs.py**: File này chứa code hiện thực cấu trúc trạng thái và giải thuật DFS.
- **ga.py**: File này chứa code hiện thực giải thuật Genetic Algorithm.
- **UI.py**: File này chứa code hiện thực phần GUI demo cho game Bloxorz.
- **main.py**: Đây là file để ta chạy demo chương trình.

Do phần hiện thực giải thuật DFS không quá phức tạp, tương tự như game Water Sort, ta chỉ tập trung vào file **ga.py** chứa code hiện thực giải thuật di truyền. Ở file này, ta có các class sau:

- **Individual**: Class này đại điện cho mỗi cá thể, lưu các thông tin như chuỗi gene, độ fitness, vị trí tốt nhất để làm điểm cắt khi lai (*best_position*).
- **Population**: Class này đại điện cho một quần thể, lưu các thông tin cần thiết như danh sách cá thể, cá thể tốt nhất, kích thước, tổng fitness quần thể.
- **MagicTableSolver**: Đây là class giúp ta giải quyết vấn đề tính toán "magic distance" của mỗi ô trong map tới vị trí goal. Mehthod quan trọng của class này nằm ở method *calc_recur*, thực hiện một vòng lặp duyệt loang từ ô goal đến tất cả các ô xung quanh, kết hợp với phương pháp quy hoạch động để tính magic distance cho mỗi ô.
- **GASolver**: Đúng như tên gọi, đây là là class mà ta sẽ dùng để giải quyết bài toán tìm kiếm bằng giải thuật di truyền GA (solver). Method *solve* của class chính là method mà ta sẽ gọi khi muốn giải một testcase nào đó.
  Các attribute của class là các tham số mà ta cài trước cho thuật toán di truyền. Các attribute của class này gồm có:
  (a) *alphabet*: tập hợp các kí tự mà mỗi gene của cá thể có thể nhận.
  (b) *game*: tham chiếu đến một object, đại diện cho game Bloxorz với một testcase đã chọn trước.
  (c) *x_para*: tham số x trong mã giả, là tỉ lệ cá thể được chọn làm bố mẹ tiềm năng trong quần thể ở bước reproduce.
  (d) *y_para*: tham số y trong mã giả, là tỉ lệ số con đem đi đột biến trong số con vừa được tạo ra.
  (e) *goal_score*: là mức điểm fitness ghi nhận cho cá thể đã tới đích. Ta cần biến này, vì sau mỗi vòng lặp ta sẽ kiểm tra phần tử có độ fitness cao nhất trong quần thể để kiểm tra đã tìm thấy lời giải.
  (f) *magic_table*: là bảng lưu "magic distance" của mọi ô trong map ban đầu đến goal. Bảng này chỉ được tính duy nhất lúc lúc thực hiện constructor một instance của class GASolver.
  (g) *mts*: là một instance của class MagicTableSolver, giúp ta tính toán bảng magic distance.
  (h) *len_chromosome_min*: Chiều dài tối thiểu chuỗi gene của mỗi cá thể trong quần thể lúc khởi tạo.

(i) *len_chromosome_max*: Chiều dài tối đa chuỗi gene của mỗi cá thể trong quần thể lúc khởi tạo.

(j) *population_size*: tham số kích thước quần thể.

Các method chính của class ứng với các bước trong mã giả của thuật toán gồm có:

(a) *initialize_population* (khởi tạo quần thể)

(b) *reproduce* (chọn tập bố mẹ tiềm năng)

(c) *cross-over* (lai tạo)

(d) *mutation* (đột biến)

(e) *rank_selection* (refine quần thể)

(f) *compute_fitness* (tính fitness toàn bộ quần thể)

Các method phụ trợ của class này gồm có:

(a) *roulette_wheel_selection*: Bộ chọn cá thể từ quần thể dựa trên xác suất được chọn của mỗi cá thể, xác suất được tính bằng fitness cá thể chia cho tổng fitness của quần thể.

(b) *best_position_cross*: Lai tạo hai cá thể theo phương pháp best position cross đã đề cập ở trên.

(c) *create_magic_table*: Dùng để khởi tạo bảng magic table.

(d) *execute_individual*: Thực thi chuỗi hành động ứng với chuỗi gene của cá thể, trả về trạng thái cuối cùng của cái thùng và danh sách các vị trí tốt đã di qua. Method này có thực hiện luôn việc refine chuỗi, tức là loại bỏ những gene nhiễu khiến thùng xuống vực, và những đoạn gene vô nghĩa khiến thùng trở lại vị trí ban đầu. Đồng thời, vị trí tốt nhất để làm điểm cắt để lai cũng được tính trong method này.

(e) *check_special_location*: Giúp xác định các vị trí tốt đã đi qua trong quá trình thực hiện chuỗi hành động ứng với chuỗi gene của cá thể.

(f) *fitness_function*: Dùng để tính fitness của mỗi quần thể, bằng cách gọi thực thi chuỗi hành động ứng với chuỗi gene của cá thể, sau đó chấm điểm dựa trên khoảng cách từ vị trí cuối cùng đến goal, và cộng điểm dựa trên các vị trí tốt đã đi qua.

(g) *refine_result*: chuyển kết quả tìm được thành danh sách trạng thái để GUI render.

(h) *is_goal*: Kiểm tra điều kiện hội tụ trước khi thực hiện vòng lặp tìm kiếm lời giải ở method *solve*.

3 Kết quả demo

3.1 Hướng dẫn dùng GUI

- **Cài đặt môi trường và thư viện hỗ trợ**

  Yêu cầu môi trường : Python 3.8 trở lên, Pip Package version mới nhất.

  Để cập nhật pip package mới nhất:

  ```
  > py -m pip install -upgrade pip
  ```

  Cài đặt thư viện cần thiết: **queue, pygame, openGL, cv2, numpy, random**

- **Thực thi chương trình**

  Vào thư mục Bloxorz và mở IDE đã cài môi trường, thực thi dòng lệnh trên Terminal

  ```
  > python main.py stage-number algorithm-option
  ```

Trong đó:

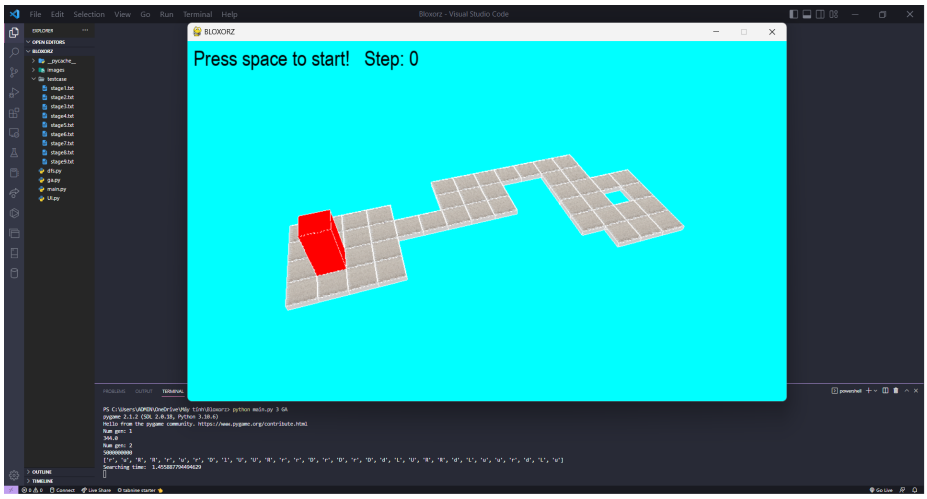
- **main.py:** là file python thực thi chương trình nằm trong thư mục Watersort
- **stage-number:** là đường dẫn đến file stage ứng với số được nhập trong thư mục ./testcase
- **algorithm-option:** là giải thuật muốn kiểm tra, gồm các giá trị: DFS, GA

3.2 Kết quả demo

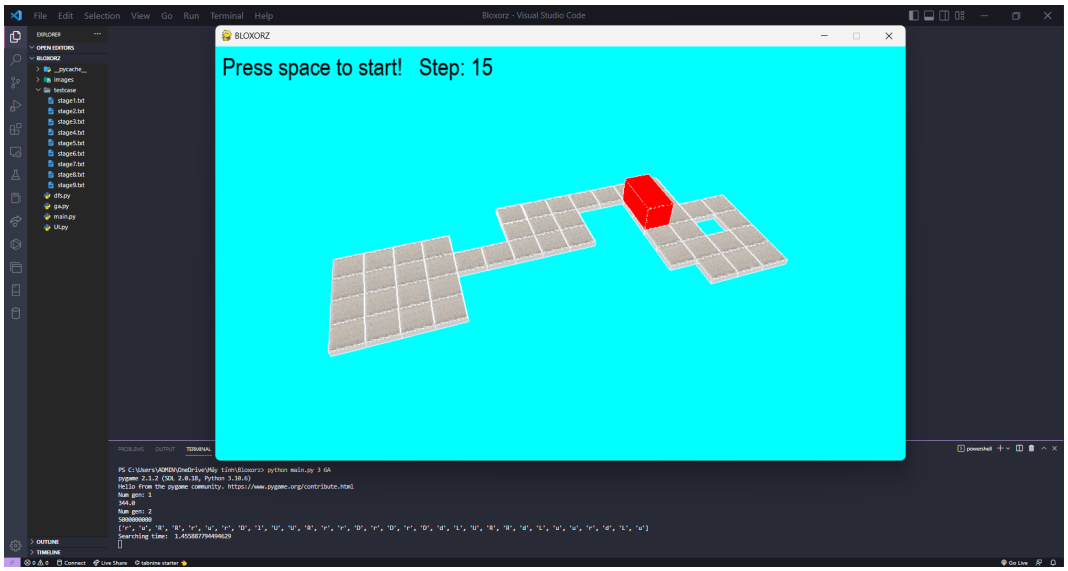
Ví dụ: Muốn chạy stage3 có trong thư mục ./testcase bằng giải thuật GA, ta thực hiện dòng lệnh

```
> python main.py 3 GA
```

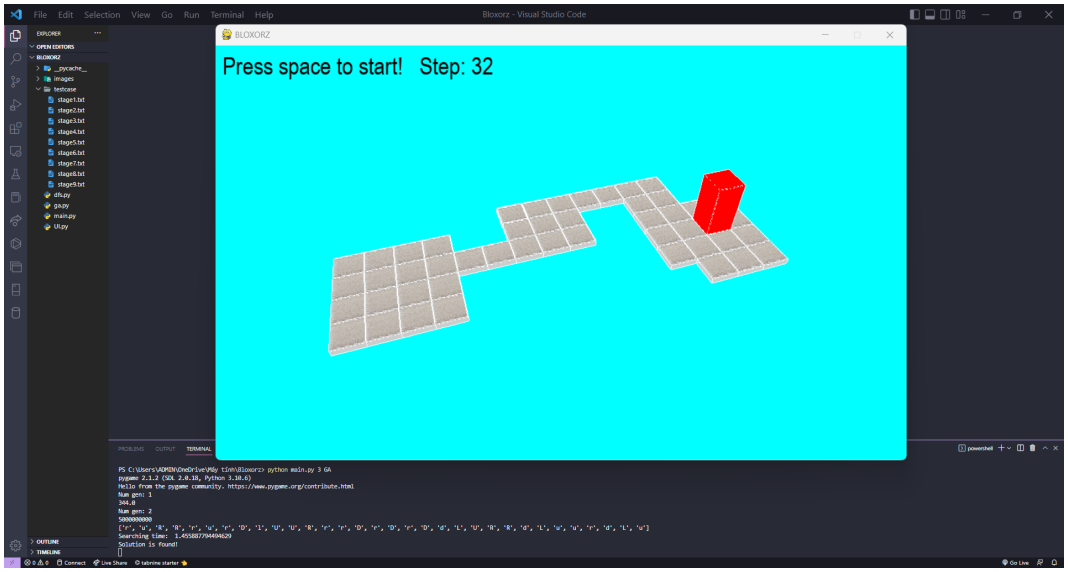
Sau khi thực thi dòng lệnh, màn hình PyGame sẽ hiển thị lên giao diện game Watersort tương ứng với stage3 đã chọn. Nhấn Space để bắt đầu



(a) Giao diện game khi vừa bắt đầu với Step = 0



(b) Game đang trong quá trình thực hiện đi đến kết quả



(c) Trò chơi kết thúc với Step = 32

3.3 Đánh giá hiệu năng

Mã nguồn được chạy để kiểm tra thời gian thực thi được thực hiện trên máy ảo Oracle VM VirtualBox chạy hệ điều hành Ubuntu 64-bit được cài đặt trên máy tính Windows 11 64-bit, AMD Ryzen 7 5700U, CPU 1.80 GHz, RAM 8 GB (6.82 GB usable)

Tương tự với Watersort, nhóm đã đo **10** lần rồi tính trung bình để có kết quả đáng tin cậy hơn.

Ví dụ: Với testcase đơn giản như stage3.txt chạy bằng giải thuật GA. Ta vào thư mục Bloxorz, chạy lệnh trên terminal chứa địa chỉ thư mục đó. Ta thực hiện lệnh sau:

```
$ python3 ./measure.py 3 GA
```

```
dinhtuan@dinhtuan-VirtualBox:~/Desktop/Bloxorz$ python3 ./measure.py 3 GA
Stage 3

Num gen: 1
332.25
Num gen: 2
5000000000
['U', 'r', 'D', 'L', 'd', 'r', 'U', 'L', 'u', 'r', 'd', 'R', 'R', 'R', 'u', 'r', 'd',
 'l', 'u', 'u', 'r', 'r', 'r', 'D', 'd', 'r', 'd', 'R', 'u', 'l', 'L', 'd', 'r', 'u',
 'L', 'D', 'R', 'U']
Time: 2.607 secs
Memory usage: 11144.0 KBytes
```

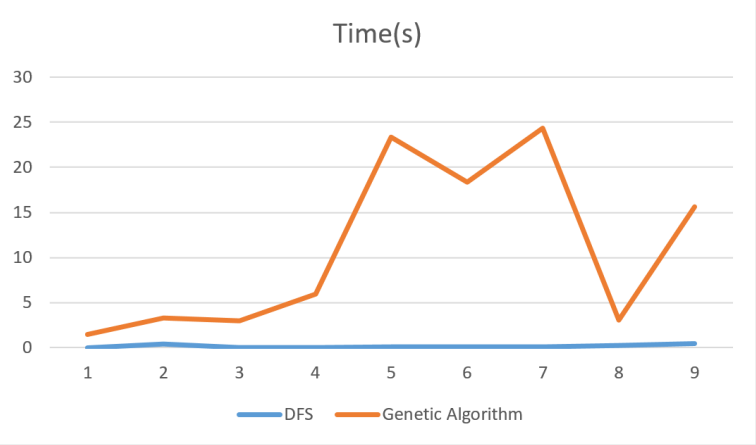
3.3.1 Bảng số liệu cho các giải thuật

Testcase	DFS			Genetic Algorithm		
	Time	Number explored	Memory (Kbyte)	Time	Number generation	Memory (Kbyte)
Stage 1	0.0155	38	10132	1.443	0	10352
Stage 2	0.397	671	16176	3.316	2	11392
Stage 3	0.034	66	10920	2.973	2	10880
Stage 4	0.054	82	11488	5.955	8	11432
Stage 5	0.117	155	13604	23.358	12	12740
Stage 6	0.058	81	11566	18.345	19	11720
Stage 7	0.135	239	11840	24.338	23	11854
Stage 8	0.232	148	19174	3.059	1	11284
Stage 9	0.492	488	25960	15.606	19	12165

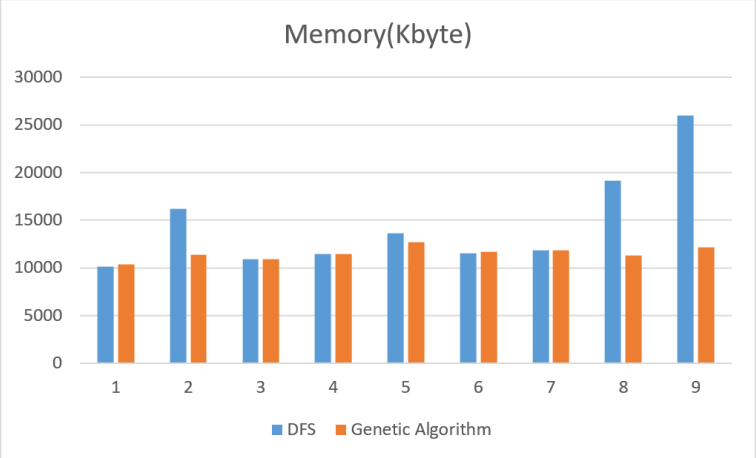
Một số giải thích:

- Cột Time là thời gian thực thi của giải thuật (đơn vị giây).
- Cột Number explored là số node đã được khai phá trong quá trình tìm kiếm lời giải.
- Cột Number generation cho biết thế hệ thứ mấy đã sinh ra lời giải, quy ước quần thể ban đầu là thế hệ 0.
- Cột Memory là bộ nhớ tiêu tốn để sử dụng cho giải thuật (đơn vị kilobyte).

3.3.2 Các đồ thị trực quan so sánh các giải thuật



Hình 6: So sánh thời gian chạy trong DFS và Genetic Algorithm



Hình 7: So sánh bộ nhớ tiêu tốn của DFS và Genetic Algorithm

3.3.3 Phân tích kết quả

Thông qua kết quả so sánh trực quan trên, ta có những phân tích sau đây:

- Sở dĩ giải thuật DFS luôn tìm được lời giải không quá 1 giây đối với game này là vì số lượng trạng thái trong không gian trạng thái của trò chơi này là quá ít, dưới 10000 state. Thật vậy giả sử kích thước map input file là $m \cdot n$, và ta có k switch. Khi đó số trạng thái có thể có không vượt quá $2^k[mn + (m-1)n + (n-1)m] = 2^k(2mn - m - n)$. Với các thông số m, n, k từ các màn của trò chơi (m, n thường xấp xỉ 20, k xấp xỉ 5) ta dễ tính ra số lượng trạng thái của trò chơi không vượt quá 24320, đa phần các màn chơi là dao động dưới 8000 trạng thái. Điều này là điều vô cùng may mắn khi giải bằng thuật toán DFS. Cho dù thuật toán có phải

duyệt hết không gian trạng thái thì vẫn đem lại lời giải trong thời gian vô cùng khả quan. Do đó, việc thử thêm testcase cho giải thuật DFS là không cần thiết.

- Giải thuật Genetic Algorithm có thời gian cho kết quả thường nhiều hơn 1 giây và trung bình là 10 giây cho một testcase, rất lâu so với giải thuật DFS. Điều này cũng dễ hiểu vì số lượng trạng thái phải duyệt của DFS là tương đối ít. Trong khi, giải thuật Genetic Algorithm trông chờ vào cách đột biến và lai tạo để tạo các cá thể con tốt cho đời sau, đặc biệt độ hiểu quả của thuật toán bị ảnh hưởng rất lớn bởi hàm fitness. Ngoài ra, do nhóm đo trên máy ảo, chạy trên hệ điều hành gốc là Windows nên hiệu năng về mặt thời gian sẽ lâu hơn so với chạy trực tiếp trên hệ điều hành gốc. Nếu chạy trên hệ điều hành gốc, hiệu năng thời gian có thể cải thiện rất nhiều.
- Hàm fitness của giải thuật Genetic Algorithm mà nhóm thiết kế chưa đủ tốt vì lượng heuristic dùng để tạo ra nó chưa đủ nhiều. Ngoài ra, ta có nhận xét với đặc trưng game Bloxorz, những màn khó từ màn 5 trở đi, mỗi màn sẽ có những heuristic riêng, đòi hỏi người lập trình phải nghiên cứu kĩ bản đồ để tìm chiến lược chơi hiệu quả. Việc thiết kế một hàm fitness dựa trên việc kết hợp nhiều heuristic sẽ tạo ra một hàm vô cùng phức tạp. Ngoài ra, việc này cũng như đòi hỏi sự can thiệp của người lập trình trong quá trình nghiên cứu bản đồ mỗi màn. Việc tạo ra một thuật toán như vậy là cực kì tiêu tốn công sức, nhưng vẫn không đảm bảo mang lại hiệu quả. Do đó nhóm quyết định dừng việc test ở màn 9, mặc dù có rất nhiều màn ở phía sau có địa hình phù hợp với heuristic mà nhóm đã xây dựng (11, 12, 14, 17, 18, 22, 26, 28,...).
- Giải thuật DFS tiêu tốn nhiều bộ nhớ vì nó phải duyệt nhiều trạng thái, mức tiêu tốn bộ nhớ tăng dần với số lượng trạng thái và độ khó của màn chơi. Giải thuật Genetic Algorithm tiêu tốn nhiều bộ nhớ vì nó phải tốn bộ nhớ lưu nhiều cá thể, nhưng do kích thước quần thể là số cố định, cũng như kích thước chuỗi gene của mỗi cá thể đã bị chặn trên nên nhìn chung độ tiêu tốn bộ nhớ của giải thuật xấp xỉ nhau mỗi testcase. Do đó, ta có thể kết luận độ tiêu tốn bộ nhớ của giải thuật Genetic Algorithm là nhỏ hơn so với giải thuật DFS.

Tài liệu

[1] Genetic Algorithm. Link: https://www.cs.ucc.ie/~dgb/courses/tai/notes/handout12.pdf.

[2] Variants of A*. Link: http://theory.stanford.edu/~amitp/GameProgramming/Variations.html.