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

NHẬP MÔN TRÍ TUỆ NHÂN TẠO

Bài tập lớn 1

TÌM KIẾM (SEARCHING)

Giảng viên: Vương Bá Thịnh
Nhóm sinh viên: Nguyễn Lê Huy - 1611290
Trương Hoàng Huy - 1611352
Nguyễn Anh Khoa - 1611617
Nguyễn Quang Hoàng Lâm - 1611743
Trần Đăng Khôi - 1611660

TP. HỒ CHÍ MINH, THÁNG 5/2018

Mục lục

1 Đề bài

Sinh viên phải hiện thực ba giải thuật: Depth-first search, Breadth-first search, và Simple Hill Climbing (hoặc 1 giải thuật heuristic khác) để giải 33 màn của trò chơi **Bloxorz**.

2 Cài đặt

Dự án này dùng console, đã được kiểm tra trên linux, còn trên window thì không đảm bảo hoạt động ổn định.

2.1 Yêu cầu

Để có thể chạy được các file python, ta cần phải có đủ các yếu tố sau:

- Phải có python 3.6 trở lên
- Có pip để cài đặt các packages cần thiết

2.2 Cách cài đặt và chạy

```
> pip install -r requirements.txt
> python main.py
```

Hoặc ta có thể dùng virtualenv

```
> virtualenv bloxorz-env
> source bloxorz-env/bin/activate
> pip install -r requirements.txt
> python main.py
```

Ta chạy trò chơi bằng câu lệnh "`python main.py`" hoặc "`python3 main.py`"

3 Cấu trúc files

```
/-----------------------
|---main.py
|---raw/
|---|---stage1
|---|---stage2
|---|---...
|---bloxorz/
|---|---common/
|---|---|---getKey.py
|---|---|---menu.py
|---|---|---moves.py
|---|---game/
|---|---|---play.py
|---|---|---Stage.py
|---|---|---Tile.py
|---|---|---TileType.py
|---|---solver/
|---|---|---Solver.py
|---|---|---State.py
|---|---|---Block.py
|---|---|---mode.py
|---|---|
|---|---|---algorithm implement files
|---|---stages/
|---|---|---GenStage.py
|---|---|---stage1.py
|---|---|---stage2.py
|---|---...
```

4 Cấu trúc trò chơi

4.1 Tiles (Các lát gạch)

4.1.1 Giải thích

Mỗi lát gạch là một ô trong trò chơi. Có nhiều loại gạch là:

1. Lát gạch bình thường (tile): Loại này chỉ là một ô bình thường.
2. Lát gạch mềm (soft tile): Loại này không thể giữ 2 khối cùng 1 lúc, tức là 2 khối lập phương chồng lên nhau cùng đứng trên ô gạch này thì sẽ làm vỡ gạch và rớt xuống.

3. Lát gạch "nút" (button tile): Loại này như là nút, khi khối lập phương đè lên nó thì nó sẽ kích hoạt một cái gì đó tùy thuộc vào màn chơi. Và ô gạch này cũng có nhiều loại.

4. Lát gạch "cầu" (bridge tile): Loại này là cái sẽ bị kích hoạt bởi ô gạch "nút". Nó có thể được đóng hoặc mở hoặc cả hai.

5. Lát gạch "mục tiêu" (goal tile): Khi cả 2 khối lập phương cùng đứng trên ô gạch này là sẽ chiến thắng màn chơi.

6. Lát gạch "phân chia" (split tile): Khi cả 2 khối lập phương cùng đứng trên ô gạch này thì sẽ bị phân chia thành 2 khối lập phương tách biệt nhau ở 2 vị trí khác nhau.

Lát gạch "nút" được chia thành 2 loại chính là **soft** và **hard**, **soft** chỉ cần 1 khối lập phương đè lên còn **hard** thì cần cả 2 khối đè lên để kích hoạt. Mỗi loại đó thì được chia thành các loại sau:

1. Loại thường: Loại này thì có nhiệm vụ toggle, tức là đóng và mở.

2. Special: Loại này chỉ có đóng hoặc mở chứ không toggle.

3. Hell: Loại này kết hợp cả loại thường và special.

Sau khi xem xét các loại trên, cuối cùng ta có được một lớp enum như sau:

```
// c++
enum class TileType : int {
    normal,
    goal,
    split,
    soft_ground,

    soft_button,
    hard_button,

    soft_special_button,
    hard_special_button

    soft_hell_button,
    hard_hell_button
};
```

4.1.2 Lớp Tile

Lớp Tile được hiện thực như sau:

```
// c++
class Tile {
    public:
    TileType type;
    bool valid;         // if standing on this tile is ok
    int split_place[4]; // place after split

    Tile*[] toggle;       // Tiles to toggle
    Tile*[] open;         // Tiles to change to open
    Tile*[] close;        // Tiles to change to close

    int* trigger(bool standing);
};
```

Do có nhiều loại gạch nên ta có nhiều cách construct:

Loại gạch thường là dễ nhất: `Tile()`

Loại gạch "mục tiêu" và "mềm" như sau:
`Tile(TileType.goal)` và `Tile(TileType.soft_ground)`

Loại gạch "cầu" cần phải biết lúc đầu nó có hay chưa nên có thêm biên bool như sau: `Tile(TileType.bridge, true)` hoặc `Tile(TileType.bridge, false)`

Loại gạch "nút" là khó nhất, và ta đã làm như sau:

```
// c++

// making a button which will toggle on active
Tile(TileType.soft_button, Tile*[] toggle);

// making a button which will open gates only
Tile(TileType.soft_special_button, Tile*[] open, NULL);

// making a button which will close gates only
Tile(TileType.soft_special_button, NULL, Tile*[] close);

// making a button which will open and closes set of bridges
Tile(TileType.soft_special_button, Tile*[] open, Tile*[] close);

// making a button which will open and closes set of bridges but also toggle some
Tile(TileType.soft_hell_button, Tile*[] toggle, Tile*[] open, Tile*[] close);

// the same for hard button
```

4.1.3 Quá trình kích hoạt

Một lát gạch có một hàm kích hoạt (trigger function), nó sẽ được thực hiện tùy theo loại gạch như sau:

- Loại "nút": tùy theo loại nút, hàm có thể đóng/mở/toggle các cây cầu
- Loại "phân chia": hàm sẽ trả về địa chỉ mới của các khối lập phương
- Loại "mềm": hàm sẽ ném (throw) tính hiệu lỗi "Fall"
- Loại "cầu": nếu không hiệu lực thì hàm sẽ throw "Hidden bridge"

Ngoài các trường hợp trên thì hàm sẽ không làm gì cả.

4.2 Nền (stage)

Lớp nền lưu màn chơi và vị trí của các khối lập phương thành các file binary.

```
// c++
class Stage {
    public:
    Tile[][] board;  // a 2 dimension matrix Tile
    int _x;
    int _y;
    char* name;

    Stage(char* name, Tile[][] b, int x, int y);
    void save(char* filename);
}
```

Lớp này được truyền vào lớp trạng thái khi chơi.

4.3 Khối lập phương (block)

Khối lập phương là nhân vật chính trong trò chơi, mọi thứ đều diễn ra xung quanh khối lập phương. Để phân biệt, khi 2 khối lập phương dính vào nhau thì ta gọi là Blox, còn riêng biệt thì ta gọi là Block. Block chính nó chỉ có 1 index, index còn lại được tính toán thông qua trạng thái (đứng, nằm dọc, nằm ngang).

Khi hàm phân chia kích hoạt, Block sẽ chuyển trạng thái của nó thành neutral. Vì thế ta cần thêm 1 block để đại diện cho block được phân chia. Blox được hiện thực là 1 array ở trong lớp trạng thái. Ta có một bit chọn block nào sẽ được di chuyển khi hàm di chuyển được gọi.

4.4 Các nước di chuyển (moves)

Đây là lớp enum như sau:

```
// c++
enum moves {
    nomoves,
    up,
    down,
    right,
    left,
    split,
    join,
    swap
};
```

4.5 Trạng thái (state)

Lớp trạng thái hơi đơn giản, nó chỉ gồm cái nền và khối lập phương và một số phương thức.

```
    // c++
    class State {
        Tile[][] board;
        Block blox[2];
        int selection = 1;
        moves[] moves;

        void toggleActive() {
            if (this->selection == 2)
                this->selection = 1;
            else
                this->selection = 2
        }
    };

    void move(State* s, moves m) {
        void* ret = NULL;
        Block* block = &(s->blox[x->selection - 1])
        try {
            block->move(m);

            if (block is out of bound)
                throw outofbound
```

```
            // trigger the tiles
            ret = s->board[block->index()].trigger();

            if !(block->standing || block->spliting)
                // trigger the other block
        }
        catch(everything) {
            // reverse the move
            block->move(m.reverse())
            throw error;
        }

        s->moves.append(m)

        if (ret != NULL) {
            // ret is now a index after split
            s->blox[0]->split(ret);
            // post process
            // blox[1] = new Block(ret);

            s->moves.append(moves.split)
        }

        // try to join
        // if not split, return false
        // if not able to join, return false
        // else join and return true and add to list of moves
        if (s->join())
            s->moves.append(moves.join)
    }
```

Khi viết các giải thuật, chỉ cần gọi `move(s, direction)` là xong. Nếu khi di chuyển mà có lỗi (ra ngoài nền, đứng trên lát gạch "mềm"), thì cứ việc bỏ qua nó.

4.6 Lời giải (solver)

Framework này dựa trên mẫu chiến lược, tùy theo chế độ nào mà ta chọn, nó sẽ chạy giải thuật tương ứng. Lớp chính của solver lưu trạng thái khởi đầu, trạng thái kết thúc nếu có, và chế độ giải thuật. Hơn nữa, khi tạo các trạng thái, trạng thái đã tồn tại có thể được tạo đi tạo lại. Để ngăn chặn điều này, lớp solver sẽ lưu một danh sách các "kí hiệu trạng thái".

"Kí hiệu trạng thái" là một chuỗi được định nghĩa rằng trạng thái được sinh ra là duy nhất.

Một lời gọi tới solver sẽ trông giống như thế này:

```
// c++
BinaryLoader* binary = new BinaryLoader();  // a fake binary loader
Stage* stage = binary->load("Filename");
State* init = new State(stage);
m = mode.bfs;  // bfs algo
Solver* problem = new Solver(init, mode);
problem->solve();
problem->printSolution();
```

5 Các giải thuật

5.1 BFS

5.1.1 Giải thích

Giải thuật được hiện thực theo mã giả sau:

```
Tạo queue với 1 phần tử là trạng thái khởi đầu
Khởi tạo dãy dấu vết
Khởi tạo bộ đếm bằng 0
Lặp cho tới khi queue rỗng:
    Nếu bộ đếm > 50000 thì trả về False
    Tăng bộ đếm lên thêm 1
    Lấy cur_state = dequeue(queue)
    Nếu cur_state là đích thì trả về True
    Lặp từng khối:
        Lặp từng nước đi:
            thử:
                sinh ra trạng thái mới là new_state
                lấy dấu vết của new_state
                nếu dấu vết không có trong dãy dấu vết thì:
                    enqueue(new_state)
                    thêm dấu vết vào dãy dấu vết
            thấy lỗi thì cứ tiếp tục
Trả về False
```

5.1.2 Bảng số liệu

Bảng số liệu về thời gian:

Màn chơi	Thời gian(s)
1	0.266571521759033
2	1.3626389503479
3	0.355221509933472
4	0.422888994216919
5	1.83321666717529
6	0.495889902114868
7	0.80048131942749
8	4.72867488861084
9	5.53647089004517
10	106.905389547348
11	1.13060832023621
12	2.10779356956482
13	1.10481071472168
14	3.52388048171997
15	81.3969686031342
16	2.33484292030334
17	5.38768458366394
18	4.20741415023804
19	1.68268489837646
20	73.6282043457031
21	1.71589279174805
22	2.1993715763092
23	78.8143999576569
24	3.81749391555786
25	3.30008316040039
26	159.813554048538
27	2.79411315917969
28	131.718682527542
29	13.9431304931641
30	5.53034996986389
31	8.13841199874878
32	4.42125082015991
33	7.48211884498596

5.2 DFS

5.2.1 Giải thích

Giải thuật được hiện thực theo mã giả sau:

```
Tạo stack là list với 1 phần tử là trạng thái khởi đầu
Khởi tạo dãy dấu vết
Khởi tạo bộ đếm bằng 0
```

```
Lặp cho tới khi stack rỗng:
    Nếu bộ đếm > 50000 thì trả về False
    Tăng bộ đếm lên thêm 1
    Lấy cur_state = phần tử cuối của stack
    Xóa phần tử cuối của stack đi
    Nếu cur_state là đích thì trả về True
    Lặp từng khối:
        Lặp từng nước đi:
            thử:
                sinh ra trạng thái mới là new_state
                lấy dấu vết của new_state
                nếu dấu vết không có trong dãy dấu vết thì:
                    thêm new_state vào list stack
                    thêm dấu vết vào dãy dấu vết
            thấy lỗi thì cứ tiếp tục
Trả về False
```

5.2.2 Bảng số liệu

Bảng số liệu về thời gian:

Màn chơi	Thời gian(s)
1	0.135706424713135
2	1.58651113510132
3	0.313766956329346
4	0.474039077758789
5	0.696768522262573
6	0.413698673248291
7	0.701162338256836
8	0.7951500415802
9	8.05996417999268
10	11.2849471569061
11	1.04428720474243
12	2.22384285926819
13	1.14733099937439
14	1.85471248626709
15	6.61174869537354
16	4.19367742538452
17	3.84662222862244
18	2.80050277709961
19	1.08713054656982
20	64.9240102767944
21	0.979677677154541
22	1.60459566116333
23	29.9765365123749
24	0.794204235076904
25	1.6055965423584
26	69.1387176513672
27	2.36821866035461
28	120.781942844391
29	12.1229801177979
30	2.8791708946228
31	5.11534285545349
32	2.44287276268005
33	7.44981098175049

5.3 HILL CLIMBING

5.3.1 Giải thích

Giải thuật được hiện thực theo mã giả sau:

```
Khởi tạo biến prevEval
Khởi tạo dãy dấu vết
Khởi tạo bộ đếm bằng 0
```

```
Khởi tạo cur_state là trạng thái khởi đầu
Cứ lặp:
    Nếu bộ đếm > 50000 thì trả về False
    Tăng bộ đếm lên thêm 1
    Nếu cur_state là đích thì trả về True
    Tạo cannotMove là list nếu khối bị chia thì là 2 phần tử,
        không thì 1 phần tử
    Đặt tất cả các phần tử của cannotMove là True
    Lặp từng khối:
        Lặp từng nước đi:
            thử:
                sinh ra trạng thái mới là new_state
                lấy dấu vết của new_state
                gán giá trị trả về của hàm EVAL(new_state) là newEval
                nếu dấu vết không có trong dãy dấu vết
                    và newEval tốt hơn prevEval thì:
                        gán prevEval = newEval
                        gán cur_state = new_state
                        thêm dấu vết vào dãy dấu vết
                        phần tử của cannotMove tại khối thứ i = False
                        thoát vòng lặp nước đi
            thấy lỗi thì cứ tiếp tục
    Nếu (cannotMove có 2 phần tử và cả 2 đều là True)
        hoặc (1 phần tử là True) thì trả về False
Trả về False
```

Hàm EVAL được thực hiện như sau:

5.3.2 Bảng số liệu