

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 | 2 |
| 2 | Cài đặt | 2 |
| 2.1 | Yêu cầu | 2 |
| 2.2 | Cách cài đặt và chạy | 2 |
| 3 | Cấu trúc files | 3 |
| 4 | Cấu trúc trò chơi | 3 |
| 4.1 | Tiles (Các lát gạch) | 3 |
| 4.1.1 | Giải thích | 3 |
| 4.1.2 | Lớp Tile | 4 |
| 4.1.3 | Quá trình kích hoạt | 5 |
| 4.2 | Nền (stage) | 6 |
| 4.3 | Khối lập phương (block) | 6 |
| 4.4 | Các nước di chuyển (moves) | 6 |
| 4.5 | Trạng thái (state) | 7 |
| 4.6 | Lời giải (solver) | 8 |
| 5 | Các giải thuật | 8 |
| 5.1 | BFS | 8 |
| 5.1.1 | Giải thích | 8 |
| 5.1.2 | Bảng số liệu | 9 |
| 5.2 | DFS | 10 |
| 5.2.1 | Giải thích | 10 |
| 5.2.2 | Bảng số liệu | 11 |
| 5.3 | HILL CLIMBING | 13 |
| 5.3.1 | Giải thích | 13 |
| 5.3.2 | Bảng số liệu | 13 |



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

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

Hoặc ta có thể dùng virtualenv

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

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

3 Cấu trúc files

```
1 /-----
2 |---main.py
3 |---raw/
4 |---|---stage1
5 |---|---stage2
6 |---|---...
7 |---bloxorz/
8 |---|---common/
9 |---|---|---getKey.py
10 |---|---|---menu.py
11 |---|---|---moves.py
12 |---|---game/
13 |---|---|---play.py
14 |---|---|---Stage.py
15 |---|---|---Tile.py
16 |---|---|---TileType.py
17 |---|---solver/
18 |---|---|---Solver.py
19 |---|---|---State.py
20 |---|---|---Block.py
21 |---|---|---mode.py
22 |---|---|
23 |---|---|---algorithm implement files
24 |---|---stages/
25 |---|---|---GenStage.py
26 |---|---|---stage1.py
27 |---|---|---stage2.py
28 |---|---|---
```

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:

```
1 // c++
2 enum class TileType : int {
3     normal,
4     goal,
5     split,
6     soft_ground,
7
8     soft_button,
9     hard_button,
10
11     soft_special_button,
12     hard_special_button
13
14     soft_hell_button,
15     hard_hell_button
16 };
```

4.1.2 Lớp Tile

Lớp Tile được hiện thực như sau:

```
1 // c++
2 class Tile {
3     public:
4     TileType type;
5     bool valid;           // if standing on this tile is ok
6     int split_place[4]; // place after split
7 }
```

```
8   Tile*[] toggle;          // Tiles to toggle
9   Tile*[] open;           // Tiles to change to open
10  Tile*[] close;          // Tiles to change to close
11
12  int* trigger(bool standing);
13 };
```

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 biến bool như sau: `Tile(TileType.bridge, true)` hoặc `Tile(TileType.bridge, false)`

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

```
1 // c++
2
3 // making a button which will toggle on active
4 Tile(TileType.soft_button, Tile*[] toggle);
5
6 // making a button which will open gates only
7 Tile(TileType.soft_special_button, Tile*[] open, NULL);
8
9 // making a button which will close gates only
10 Tile(TileType.soft_special_button, NULL, Tile*[] close);
11
12 // making a button which will open and closes set of bridges
13 Tile(TileType.soft_special_button, Tile*[] open, Tile*[] close);
14
15 // making a button which will open and closes set of
16 bridges but also toggle some
17 Tile(TileType.soft_hell_button, Tile*[] toggle,
18      Tile*[] open, Tile*[] close);
19
20 // 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.

```
1 // c++
2 class Stage {
3     public:
4         Tile[][] board; // a 2 dimension matrix Tile
5         int _x;
6         int _y;
7         char* name;
8
9         Stage(char* name, Tile[][] b, int x, int y);
10        void save(char* filename);
11 }
```

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:

```
1 // c++
2 enum moves {
3     nomoves,
4     up,
5     down,
6     right,
7     left,
8     split,
9     join,
```

```
10     swap
11 };
```

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.

```
1 // c++
2 class State {
3     Tile[][] board;
4     Block blox[2];
5     int selection = 1;
6     moves[] moves;
7
8     void toggleActive() {
9         if (this->selection == 2)
10             this->selection = 1;
11         else
12             this->selection = 2
13     }
14 };
15
16 void move(State* s, moves m) {
17     void* ret = NULL;
18     Block* block = &(s->blox[x->selection - 1])
19     try {
20         block->move(m);
21
22         if (block is out of bound)
23             throw outofbound
24
25         // trigger the tiles
26         ret = s->board[block->index()].trigger();
27
28         if !(block->standing || block->splitting)
29             // trigger the other block
30     }
31     catch(everything) {
32         // reverse the move
33         block->move(m.reverse())
34         throw error;
35     }
36
37     s->moves.append(m)
38
39     if (ret != NULL) {
40         // ret is now a index after split
41         s->blox[0]->split(ret);
```



```
42         // post process
43         // blox[1] = new Block(ret);
44
45         s->moves.append(moves.split)
46     }
47
48     // try to join
49     // if not split, return false
50     // if not able to join, return false
51     // else join and return true and add to list of moves
52     if (s->join())
53         s->moves.append(moves.join)
54 }
```

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:

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

5 Các giải thuật

Sự tiêu tốn bộ nhớ được kiểm tra bằng `memory_profiler` ở hàm `solve` trong file `Solver.py`

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 và bộ nhớ:



| Màn chơi | Thời gian (s) | Bộ nhớ (MiB) |
|----------|-------------------|--------------|
| 1 | 0.266571521759033 | 17.410 |
| 2 | 1.3626389503479 | 17.680 |
| 3 | 0.355221509933472 | 17.168 |
| 4 | 0.422888994216919 | 17.320 |
| 5 | 1.83321666717529 | 17.316 |
| 6 | 0.495889902114868 | 17.332 |
| 7 | 0.80048131942749 | 17.164 |
| 8 | 4.72867488861084 | 19.746 |
| 9 | 5.53647089004517 | 18.215 |
| 10 | 106.905389547348 | 22.871 |
| 11 | 1.13060832023621 | 17.219 |
| 12 | 2.10779356956482 | 17.316 |
| 13 | 1.10481071472168 | 17.484 |
| 14 | 3.52388048171997 | 17.699 |
| 15 | 81.3969686031342 | 24.523 |
| 16 | 2.33484292030334 | 17.312 |
| 17 | 5.38768458366394 | 18.148 |
| 18 | 4.20741415023804 | 17.844 |
| 19 | 1.68268489837646 | 17.121 |
| 20 | 73.6282043457031 | 25.082 |
| 21 | 1.71589279174805 | 17.441 |
| 22 | 2.1993715763092 | 17.305 |
| 23 | 78.8143999576569 | 22.977 |
| 24 | 3.81749391555786 | 17.492 |
| 25 | 3.30008316040039 | 17.328 |
| 26 | 159.813554048538 | 23.262 |
| 27 | 2.79411315917969 | 17.312 |
| 28 | 131.718682527542 | 27.406 |
| 29 | 13.9431304931641 | 18.266 |
| 30 | 5.53034996986389 | 17.633 |
| 31 | 8.13841199874878 | 17.926 |
| 32 | 4.42125082015991 | 17.477 |
| 33 | 7.48211884498596 | 19.141 |

Bảng số liệu cho thấy những màn chơi có thời gian giải lâu tương ứng với sự tiêu tốn bộ nhớ nhiều. Tuy nhiên, có những màn chơi thời gian giải ít hơn nhưng tốn bộ nhớ nhiều hơn những màn chơi khác là do cấu trúc màn chơi đó vốn dĩ đã tốn bộ nhớ hơn.

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 và bộ nhớ:



| Màn chơi | Thời gian (s) | Bộ nhớ (MiB) |
|----------|-------------------|--------------|
| 1 | 0.135706424713135 | 17.336 |
| 2 | 1.58651113510132 | 18.691 |
| 3 | 0.313766956329346 | 17.371 |
| 4 | 0.474039077758789 | 17.320 |
| 5 | 0.696768522262573 | 18.344 |
| 6 | 0.413698673248291 | 17.285 |
| 7 | 0.701162338256836 | 17.477 |
| 8 | 0.7951500415802 | 20.508 |
| 9 | 8.05996417999268 | 27.438 |
| 10 | 11.2849471569061 | 40.230 |
| 11 | 1.04428720474243 | 17.402 |
| 12 | 2.22384285926819 | 17.883 |
| 13 | 1.14733099937439 | 18.090 |
| 14 | 1.85471248626709 | 17.871 |
| 15 | 6.61174869537354 | 24.254 |
| 16 | 4.19367742538452 | 19.793 |
| 17 | 3.84662222862244 | 19.805 |
| 18 | 2.80050277709961 | 18.938 |
| 19 | 1.08713054656982 | 17.617 |
| 20 | 64.9240102767944 | 48.371 |
| 21 | 0.979677677154541 | 17.281 |
| 22 | 1.60459566116333 | 17.855 |
| 23 | 29.9765365123749 | 51.020 |
| 24 | 0.794204235076904 | 17.324 |
| 25 | 1.6055965423584 | 17.871 |
| 26 | 69.1387176513672 | 60.266 |
| 27 | 2.36821866035461 | 17.734 |
| 28 | 120.781942844391 | 78.012 |
| 29 | 12.1229801177979 | 18.688 |
| 30 | 2.8791708946228 | 18.031 |
| 31 | 5.11534285545349 | 19.148 |
| 32 | 2.44287276268005 | 17.867 |
| 33 | 7.44981098175049 | 21.773 |

Qua bảng số liệu này cho thấy DFS giải nhanh hơn BFS. Tuy nhiên một số màn tốn bộ nhớ rất nhiều, vì có thể kết quả ở màn chơi đó nằm ở nhánh cuối và ở sâu dưới nên giải thuật phải lưu lại tất cả các nhánh phía trước.

5.3 HILL CLIMBING

5.3.1 Giải thích

Giải thuật này chính xác hơn là Steepest ascent climbing. Ta giải quyết vấn đề tối ưu cục bộ bằng backtracking. Giải thuật được hiện thực theo mã giả sau:

```
Khởi tạo dãy dấu vết
Khởi tạo bộ đếm bằng 0
Khởi tạo state là trạng thái khởi đầu
Duyệt tìm vị trí goal bằng cách dùng state
Khởi tạo stack là mảng trạng thái 1 phần tử là trạng thái ban đầu
Lặp cho đến khi stack rỗng:
    Nếu bộ đếm > 50000 thì trả về False
    Tăng bộ đếm lên thêm 1
    Gán cur_state là phần tử cuối cùng của stack
    Xóa phần tử cuối của stack đi
    Nếu cur_state là đích thì trả về True
    Khởi tạo priority là mảng rỗng
    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, goal) là newEval
                nếu dấu vết không có trong dãy dấu vết:
                    thêm dấu vết vào dãy dấu vết
                    thêm cặp (newEval, new_state) vào priority
            thấy lỗi thì cứ tiếp tục
        Sắp xếp priority giảm dần
        Lấy hết trạng thái trong priority bỏ vào stack theo thứ tự đó
    Trả về False
```

Hàm EVAL được thực hiện như sau:

```
EVAL(trạng thái, vị trí goal):
    Khởi tạo ans = 0
    Lặp từng khối lập phương:
        ans += khoảng cách giữa từng khối đến goal
    Trả về ans
```

5.3.2 Bảng số liệu

Bảng số liệu về thời gian và bộ nhớ:



| Màn chơi | Thời gian (s) | Bộ nhớ (MiB) |
|----------|---------------------|--------------|
| 1 | 0.0395197868347168 | 17.305 |
| 2 | 1.7283692359924316 | 18.398 |
| 3 | 0.24228239059448242 | 17.551 |
| 4 | 0.6023290157318115 | 17.328 |
| 5 | 0.8682773113250732 | 18.129 |
| 6 | 0.5019304752349854 | 17.336 |
| 7 | 0.7346084117889404 | 17.332 |
| 8 | 0.2118821144104004 | 18.234 |
| 9 | 0.9325628280639648 | 19.246 |
| 10 | 8.855451345443726 | 31.543 |
| 11 | 1.0376548767089844 | 17.246 |
| 12 | 1.1754610538482666 | 18.195 |
| 13 | 0.46993041038513184 | 17.355 |
| 14 | 1.731593370437622 | 17.879 |
| 15 | 6.74385929107666 | 22.254 |
| 16 | 0.36205363273620605 | 17.035 |
| 17 | 4.150902032852173 | 20.875 |
| 18 | 2.7860164642333984 | 18.922 |
| 19 | 0.7718052864074707 | 17.207 |
| 20 | 74.86791396141052 | 52.602 |
| 21 | 1.0988316535949707 | 17.285 |
| 22 | 0.9383811950683594 | 18.355 |
| 23 | 29.081502676010132 | 43.809 |
| 24 | 1.2967350482940674 | 18.074 |
| 25 | 1.707021713256836 | 17.516 |
| 26 | 58.00024366378784 | 39.652 |
| 27 | 2.257538318634033 | 17.945 |
| 28 | 46.117361068725586 | 31.828 |
| 29 | 11.1941556930542 | 18.434 |
| 30 | 2.666714906692505 | 18.082 |
| 31 | 5.266216039657593 | 19.316 |
| 32 | 3.187586545944214 | 18.809 |
| 33 | 6.046768665313721 | 21.145 |

Giải thuật này giải nhanh hơn hai giải thuật DFS và BFS. Nhìn chung thì hill climbing tiêu tốn bộ nhớ nhiều hơn BFS chút và ít hơn DFS.