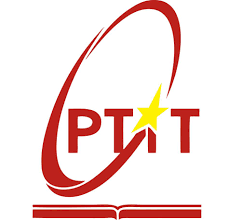
**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG**

KHOA CÔNG NGHỆ THÔNG TIN 1



**BÁO CÁO BÀI TẬP LỚN AI**

**Game Tetris tự chơi sử dụng genetic algorithm**

|  |  |  |
| --- | --- | --- |
| ***Nhóm học phần:*** | ***Nhóm 11*** | |
| ***Nhóm bài tập lớn:*** | ***Nhóm 16*** | |
| ***Sinh viên thực hiện:*** |  |  |
|  | ***Trần Quang Huy:***  ***(Trưởng nhóm)*** | ***B20DCCN327*** |
|  | ***Nguyễn Cảnh Hưng:*** | ***B20DCCN339*** |
|  | ***Nguyễn Trọng Hướng:*** | ***B20DCCN005*** |
| ***Giảng viên giảng dạy:*** | ***Cô … Quỳnh*** | |

**Hà Nội 2023**

Lời mở đầu

Contents

[**I.** **Giới thiệu game Tetris** 5](#_Toc133679657)

[**II.** **Giới thiệu về giải thuật di truyền – genetic algorithm** 5](#_Toc133679658)

[1. Khái niệm 5](#_Toc133679659)

[2. Nguyên lý chung 5](#_Toc133679660)

[3. Biểu diễn lời giải và không gian tìm kiếm 5](#_Toc133679661)

[4. Giải thuật 6](#_Toc133679662)

[**III.** **Áp dụng giải thuật di truyền vào trò chơi Tetris** 7](#_Toc133679663)

[1. Ý tưởng 7](#_Toc133679664)

[2. Xây dựng hàm Heuristics 12](#_Toc133679665)

[3. Các thao tác 12](#_Toc133679666)

[4. Đánh giá độ thích hợp 12](#_Toc133679667)

[5. Triển khai code 12](#_Toc133679668)

1. **Giới thiệu game Tetris**
2. **Giới thiệu về giải thuật di truyền – genetic algorithm**
   1. Khái niệm

Giải thuật di truyền (genetic algorithm) hay thuật toán di truyền là thuật toán tìm kiếm được thiết kế dựa trên sự tương tự với quá trình chọn lọc tự nhiên và thuyết tiến hoá của Charles Darwin. Giải thuật di truyền là trường hợp riêng của một lớp giải thuật được gọi là giải thuật tiến hoá (evolutionary algorithms).

Về mặt ý tưởng, giải thuật di truyền học tập từ quá trình sinh tồn và chọn lọc tự nhiên của sinh vật trong tự nhiên, trong đó cá thể với khả năng thích nghi cao hơn sẽ chiếm ưu thế và tồn tại. Nếu ta coi đây là bài toán tối ưu thì quá trình tiến hoá sẽ sinh ra cá thể tối ưu, tức là cá thể thích nghi cao.

* 1. Nguyên lý chung

Mỗi thế hệ gồm một số nhất định lời giải, còn gọi là ***cá thể***. Mỗi lời giải hay cá thể được biểu diễn bằng một ***nhiễm sắc thể***, chứa các ***gen***.

Độ tốt của lời giải được đánh giá bằng ***hàm thích nghi (fitness)*.**

Thông thường, hàm thích nghi chính là hàm mục tiêu của bài toán tối ưu. Các lời giải trong mỗi thế hệ được biến đổi trong quá trình thích nghi để tạo ra thế hệ tiếp theo.

Giải thuật di truyền sử dụng các quy tắc sau để tạo ra thế hệ lời giải tiếp theo:

* Các cá thể cạnh tranh với nhau. Cá thể với độ thích nghi cao hơn sẽ tạo ra nhiều hậu duệ hơn cá thể kém thích nghi *(Chọn lọc).*
* Gen từ các cá thể thích nghi tốt được kết hợp với nhau, nhờ đó có thể tạo ra hậu duệ tốt hơn tổ tiên *(Lai ghép).*
* Từng cá thể cũng có thể thay đổi gen của mình *(Đột biến).*
  1. Biểu diễn lời giải và không gian tìm kiếm

Để sử dụng giải thuật di truyền, trước hết mỗi lời giải phải được biểu diễn bằng một ***nhiễm sắc thể***, chứa các ***gen***. Ở đây, nhiễm sắc thể là một vector có độ dài xác định, ***mỗi phần tử của vetor là một gen***. Thông thường, lời giải được biểu diễn nhiều cách khác nhau: vector nhị phân, vector số, vector chữ, …

Mỗi lời giải được gán một ***độ thích nghi*** thể hiện khả năng cạnh tranh của lời giải. Thông thường, độ thích nghi được tính từ hàm mục tiêu của bài toán, hàm mục tiêu tốt tương đương với độ thích nghi cao.

Giải thuật thực hiện qua nhiều bước lặp, mỗi bước lặp tương ứng với một thế hệ. Tại mỗi thế hệ, giải thuật duy trì một quần thể. Quần thể trong thế hệ đầu tiên được khởi tạo ngẫu nhiên bằng cách sinh ngẫu nhiên các lời giải. Từ quần thể hiện tại, từng cặp cá thể được lựa chọn dựa trên độ thích nghi và được lai ghép với nhau để tạo ra cá thể con. Cá thể với độ thích nghi cao được lựa chọn với xác suất cao hơn. Cá thể con được tạo ra như vậy sẽ thay thế cá thể bố mẹ để tạo ra quần thể mới trong thế hệ tiếp theo.

* 1. Giải thuật

Quần thể đầu tiên được khởi tạo ngẫu nhiên. Sau đó thuật toán được thực hiện qua nhiều bước lặp, tại mỗi bước lặp thuật toán sinh ra một quần thể mới. Các quần thể tiếp theo được tạo ra từ quần thể trước đó bằng cách áp dụng ba thao tác: *chọn lọc, lai ghép* và *đột biến.*

**Chọn lọc** (selection), Chọn lọc cá thể để tham gia việc tạo ra thế hệ tiếp theo. Cá thể có độ thích nghi càng cao càng có nhiều khả năng được chọn.

**Lai ghép** (crossover), còn gọi là trao đổi chéo, hay tái tổ hợp. Lai ghép là thao tác kết hợp các phần từ nhiễm sắc thể của bố và mẹ để tạo ra hai cá thể con với một xác suất nhất định. Cá thể bố và mẹ được lựa chọn bằng cách sử dụng thao tác chọn lọc ở trên.

**Đột biến** (mutation). Thay đổi giá trị các gen của cá thể con vừa tạo ra với một xác suất nhất định. Cụ thể, với mỗi cá thể con, duyệt các gen và với một xác suất rất nhỏ thay đổi giá trị của gen đó. Mục đích của thao tác đột biến là để tạo ra những đoạn gen hoàn toàn mới, chưa có trong quần thể cha mẹ. Đột biến cũng cho phép hạn chế việc hội tụ quá sớm của thuật toán (vượt qua cực trị địa phương)

Pseudo code

GA(X, f, N, c, m)

Đầu vào: bài toán tối ưu với không gian trạng thái X

hàm thích nghi f

kích thước quần thể N

xác suất lai ghép c

xác suất đột biến m

Đầu ra: lời giải (cá thể) với độ thích nghi cao

Khởi tạo: Khởi tạo ngẫu nhiên quần thể G gồm N lời giải

------------------------------------------------

While (chưa thoả mãn điều kiện dừng) do

1. For i = 1 to N do

Tính giá trị hàm thích nghi f(i) cho cá thể thứ i

2. For i = 1 to ⎣N/2⎦ do

i. Chọn lọc: chọn 2 cá thể bố mẹ x và y từ G tuỳ theo giá trị thích nghi

ii. Lai ghép: với xác suất c, đổi chỗ đoạn gen trên x và y

iii. Đột biến: với xác suất m, thay đổi giá trị các gen trên cá thể mới tạo ra

iv. Thêm cá thể mới vào quần thể mới G’

3. Gán G ← G’

Return: Lời giải thuộc G với giá trị thích nghi tốt nhất

1. **Áp dụng giải thuật di truyền vào trò chơi Tetris**
   1. Ý tưởng

Mục tiêu của chúng ta là đạt càng nhiều điểm càng tốt, điều này cũng có nghĩa là xóa càng nhiều dòng càng tốt, và do đó, thực hiện càng nhiều nước đi càng tốt.

Để đạt được mục tiêu này, model của chúng ta sẽ quyết định nước đi tốt nhất cho một Tetromino hiện tại bằng cách thử tất cả các nước đi có thể (xoay và di chuyển nó trên khu vực chơi). Nó tính điểm cho mỗi nước đi có thể (cùng với quân đi trước) và chọn quân có điểm tốt nhất làm nước đi tiếp theo.

Để xác định sự tốt xấu khi đặt các Tetromino vào khu vực chơi, chúng ta sẽ cố gắng xây dựng hàm heuristics giúp đánh giá các nước đi. Bằng những suy nghĩ đơn giản khi chúng ta chơi Tetris, nhóm đưa ra 4 tham số chính của hàm heuristics:

* **Aggregate Height (Tổng chiều cao)**

`Tham số này đại diện cho chiều cao của lưới trong khu vực chơi. Được tính bằng tổng chiều cao của các cột trong lưới. Tức là vị trí của block cao nhất trong cột đó



Hình : Aggregate heights

Chúng ta muốn khi chơi chiều cao của các cột phải nhỏ nhất có thể, dĩ nhiên các cột càng thấp thì chúng ta có thể thả càng nhiều các Tetromino lên khu vực chơi. Do đó, trọng số của tham số này chúng ta mong muốn là **1 số không dương.**

**Implement code:**

def height\_for\_column(self, column):

\_, height = self.size()

for i in range(0, height):

if self.field[i][column] != 0:

return height-i

return 0

def heights(self):

result = []

width, \_ = self.size()

for i in range(0, width):

result.append(self.height\_for\_column(i))

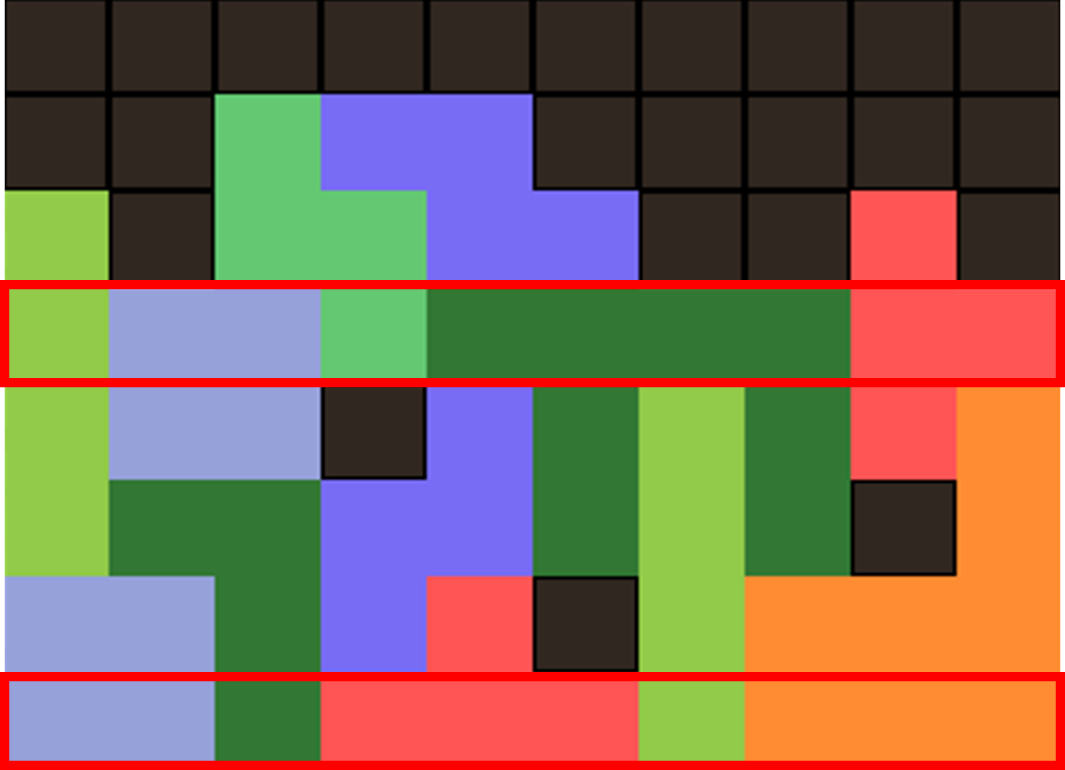
return result

def aggregate\_height(self, heights):

result = sum(heights)

return result

* **Complete lines (Số hàng đã hoàn thành)**

****

Hình : Complete lines

Số điểm trong trò chơi của chúng ta được tính thông qua số hàng mà chúng ta phá được khi xếp các Tetromino vào vừa đủ 1 hàng mà không khoảng trống. Đây là cách trực quan nhất, tham số đơn giản nhất trong hàm heuristics mà ta muốn xây dựng. Chúng ta muốn số điểm cao vì vậy số hàng được hoàn thành phải càng nhiều càng tốt, vì vậy trọng số của tham số này chúng ta mong muốn là **1 số dương.**

**Implement code:**

def complete\_line(self):

result = 0

\_, height = self.size()

for i in range (0, height) :

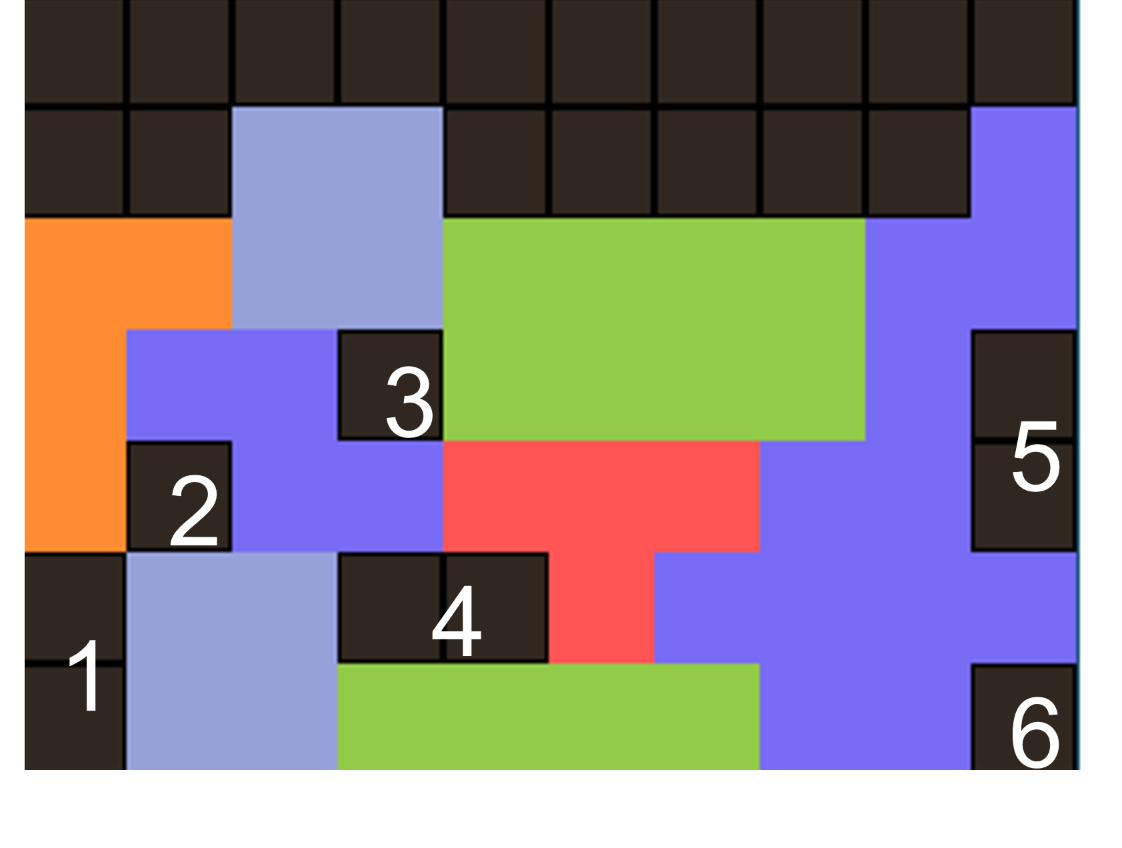
if 0 not in self.field[i]:

result+=1

return result

* **Holes (Số ô trống)**

Một khoảng trống (hole) trong 1 cột được định nghĩa là 1 vùng không gian mà có ít nhất 1 block nằm trên nó. Vì vậy, một khoảng có thể chứa nhiều ô trong lưới.



Hình : Holes

Muốn xóa càng hàng có lỗ trống, chúng ta cần phải loại bỏ những hàng ở bên trên trước rồi mới có thể đưa các tetromino mới vào lấp các lỗ. Do đó, chúng ta muốn số lỗ trống càng nhỏ càng tốt điều này tương đương với trọng số gắn với tham số này là **1 số không dương.**

**Implement code:**

def number\_of\_holes(self, heights):

result = 0

width, height = self.size()

for j in range(0, width) :

  for i in range (0, height) :

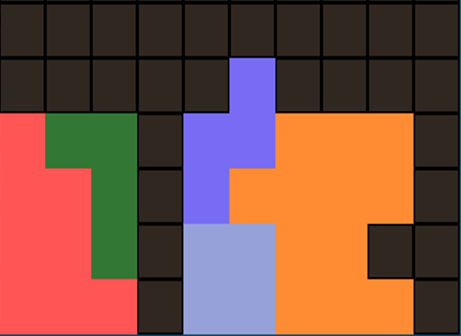
if self.field[i][j] == 0 and height-i < heights[j]:

result+=1

return result

* **Bumpiness (Độ chênh lệch chiều cao)**

Một cách tổng quát, khi chơi Tetris ta xét trạng thái sau



Hình : Well

Trạng thái gồm các cột có rất ít hoặc không có block nào và các cột bên cạnh thì lại chứa rất nhiều block. Chúng ta tạm gọi các cột có rất ít hoặc không có block nào là các “giếng”. Việc xóa các hàng trong lưới chỉ đơn giản với khối hình chữ I. Tuy nhiên các tetromino được tạo ra 1 cách random, với các tetromino khác thì việc đưa vào giếng là rất khó. Để khái quát ý tưởng về độ sâu của giếng, chúng ta tạo nên 1 tham số gọi là bumpiness.



Hình : Bumpiness

Tham số bumpiness được tính bằng tổng độ chênh lệch giữa 2 cột liền kề.

Ở hình 5, chúng ta có thể tính được tham số bumpiness

Bumpiness=|3 - 5| + |5 -5| + |5 - 5| + |5 - 6| + |6 - 5| + |5 - 5| + |5 – 6| + |6 - 5| + |5 - 0|  
 = 11

Dễ thấy sự chênh lệch chiều cao giữa giếng và các cột bên cạnh là rất lớn, dựa vào điều này ta có thể thấy tham số bumpiness này càng lớn thì trò chơi của chúng ta càng ở trong nguy cơ thua cuộc. Vì vậy, model của chúng ta phải tối hiểu hóa trong số của tham số này, tương tự với lập luận bên trên, mong muốn của chúng ta về trọng số của bumpiness là **1 số không dương.**

**Implement code:**

def bumpinesses(self, heights):

result = []

for i in range(0, len(heights)-1):

result.append(abs(heights[i]-heights[i+1]))

return sum(result)

* 1. Xây dựng hàm Heuristics

Sau khi xây dựng được 4 tham số cho hàm Heuristics, chúng ta tiến hành đưa các tham số này vào cùng 1 hàm. Một điều lưu ý là các tham số này là các số nguyên dương và thay đổi trong quá trình chơi, việc của chúng ta là đi tìm trọng số tương ứng để cho hàm Heuristics là tối ưu.

Giả sử chúng ta đang giữ 1 Tetromino T, với mỗi phép di chuyển hoặc xoay rồi đặt xuống khu vực chơi chúng ta sẽ thu được 1 bộ 4 tham số khác nhau.

Sau khi đặt được Tetromino, ta sẽ thu được:

là vector với 4 tham số kể trên

là vector với 4 trọng số tương ứng

Khi đó, để đánh giá mức độ tốt xấu của 1 hành động này, hàm Heuristics được định nghĩa:

Hành động đặt Tetromino ở vị trí thứ nhất (move) x1 được gọi là tốt hơn hành động đặt khối Tetromino ở vị trí thứ hai (move) x2 khi H(x1) > H(x­2).

Khi đặt 1 Tetromino tại các vị trí khác nhau trên khu vực chơi, model AI có thể dựa vào giá trị hàm H quyết định rằng vị trí và số lần xoay nào là tốt nhất.

Implement code:

def heuristics(self):

        heights = self.heights()

        return [self.aggregate\_height(heights), self.complete\_line(), self.number\_of\_holes(heights), self.bumpinesses(heights)]

* 1. Các thao tác
     1. Chọn lọc (Tournament selection) và lai tạo (Cross-over)

Khi một quần thể được đưa vào tự chơi Tetris, sau mỗi lần lặp (batch) mỗi cá thể sẽ mang 1 số điểm thích nghi (fitness) và trọng số (weight). Nhóm chọn ra 70% cá thể tốt nhất từ thế hệ này để đưa vào quần thể trong thế hệ tiếp theo.

survivors\_rate = 0.7 # 70% số lượng

number = 100 # số lượng cá thể trong 1 quần thể

def select\_survivors(scores, number): # chọn ra number cá thể tốt nhất

    bests = list(reversed(sorted(scores, key=itemgetter(0))))[0:number]

    return list(map(lambda x: x[0], bests)), list(map(lambda x: x[1], bests))

[...]

survivors\_score, survivors = select\_best\_individuals(scores, int(len(scores)\*survivors\_rate))

Sau đó tiếp tục chọn ra 10% số lượng cá thể thích nghi tốt nhất trong quần thể ban đầu, tiến hành lai tạo, chọn ra 2 cá thể cha mẹ bất kì trong 10% đó. Tiến hành lấy các gen của cha mẹ với xác suất 50%. Sau khi lai tạo xong, đột biến các cá thể đã được sinh ra với tỷ lệ 5%. Tiếp tục như vậy cho đến khi tạo lại đủ số lượng cá thể như quần thể ở thế hệ trước.

mutate\_chance = 0.05

def cross\_over(x, y): # lai chéo giữa các  cá thể bố mẹ.

    result = []

    for i in range(0, len(x)):

        if random.uniform(0, 1) > 0.5:

            result.append(y[i])

        else:

            result.append(x[i])

    result = normalize(result)

    return result

[...]

generation = survivors # thế hệ tiếp theo bằng 70% cá thể tốt nhất ban đầu

while len(generation) < number: # lặp cho đên khi đạt được số lượng ban đầu.

individual = cross\_over(\*random.sample(survivors[:int(bests\_rate\*number)], k=2))

if random.uniform(0, 1) < mutate\_chance: # đột biến với tỷ lệ 5%

individual = mutate(individual)

generation.append(individual)

* 1. Đánh giá độ thích hợp

Để đánh giá mức độ thích nghi của một cá thể trong quần thể, chúng ta cần xây dựng hàm thích nghi. Với trò chơi Tetris, hàm thích nghi có thể xây dựng nhiều cách khác nhau. Với một cá thể trong 1 quần thể, model AI tự chơi 5 lần. Mỗi lần chơi xong (thua hoặc đạt đến số lượng tetromino giới hạn) thì sẽ trả về giá trị bằng   
return self.lines\*1000 + self.nbPiece, với lines là số dòng đã được hoàn thành và nbPiece là số khối tetromino đã được xuất hiện. Vì chúng ta muốn tối đa hóa số dòng đã hoàn thành (tham số thứ 2 của hàm Heuristics) nên chúng ta đặt trọng số của lines là 1000. Sau đó, Hàm fitness sẽ được tính bằng trung bình cộng của 5 lần chơi đó.

* 1. Triển khai code