GIÁO VIÊN HƯỚNG DẪN

THẦY CHÂU THÀNH ĐỨC
THẦY NGÔ ĐÌNH HY
CÔ PHAN THỊ PHƯƠNG UYÊN

BÁO CÁO
PROJECT 1
TÌM ĐƯỜNG ĐI TỐI ƯU

Sinh viên: Nguyễn Đức Nam (nhóm trưởng)
MSSV: 19127048
Email: 19127048@student.hcmus.edu.vn

Sinh viên: Lê Ngọc Minh Nhật
MSSV: 19127493
Email: 19127493@student.hcmus.edu.vn

Contents

I. Adversarial Search

Adversarial Search là loại search khi mà có một Agent "đối thủ" đang cố làm thay đổi state của vấn đề đang cần giải quyết ở mỗi bước theo hướng mà chúng ta không mong muốn. Thường là trong các trò chơi mang tính đối kháng ví dụ như: cờ vua, caro, poker, …

1. Zero-sum Game

Zero-sum Game là những trò chơi thuần tuý về mặt đối kháng. Trong đó, nếu Agent 1 đang gia tăng phần thắng lên bao nhiêu thì Agent 2 sẽ giảm đi phần thắng đúng bằng bấy nhiêu và ngược lại.

Trong zero-sum Game, người chơi 1 sẽ cố gắng tìm giá trị lớn nhất (Maximize) trong khi người chơi còn lại sẽ cố tìm giá trị nhỏ nhất (Minimize).

Ví dụ các trò chơi Zero-sum Game: cờ vua.

2. Formalize bài toán

- Initial state: trò chơi được thiết lập như thế nào khi bắt đầu
- Player: ai được di chuyển ở state hiện tại
- Action: danh sách các nước đi hợp lệ
- Result: kết quả của nước đi ở state hiện tại sau khi thực hiện Action
- Terminal-test: kiểm tra trò chơi đã kết thúc hay chưa, nếu trả về True thì game đã kết thúc. State mà game đã kết thúc được gọi là Terminal States.
- Utility: trị số của Terminal State s của người chơi p

3. Thuật toán

a. Thuật toán Minimax

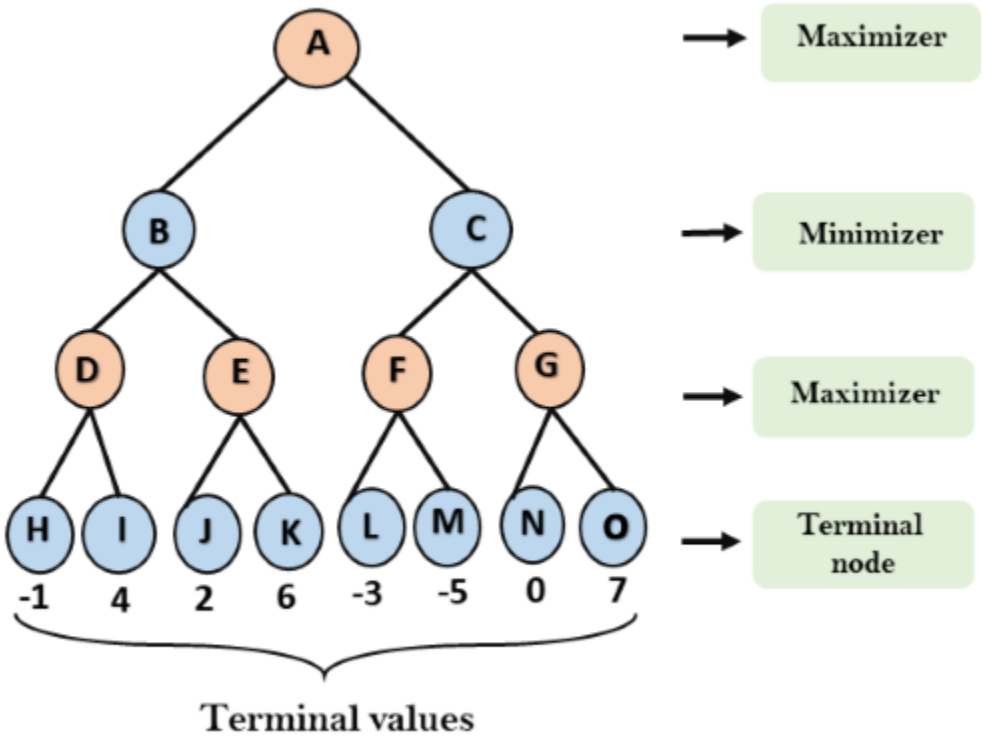
Thuật toán tính toán quyết định tối thiểu ở State hiện tại. Cung cấp nước đi hiệu quả nhất cho người chơi, giả định rằng đối thủ cũng đang chơi một cách hiệu quả. Trong thuật toán, một người chơi sẽ Max trong khi người còn lại là Min.

Thuật toán sẽ khử đệ quy từ gốc xuống tới lá của cây sau đó giá trị Minimax sẽ được trả ngược lại (backtracking).

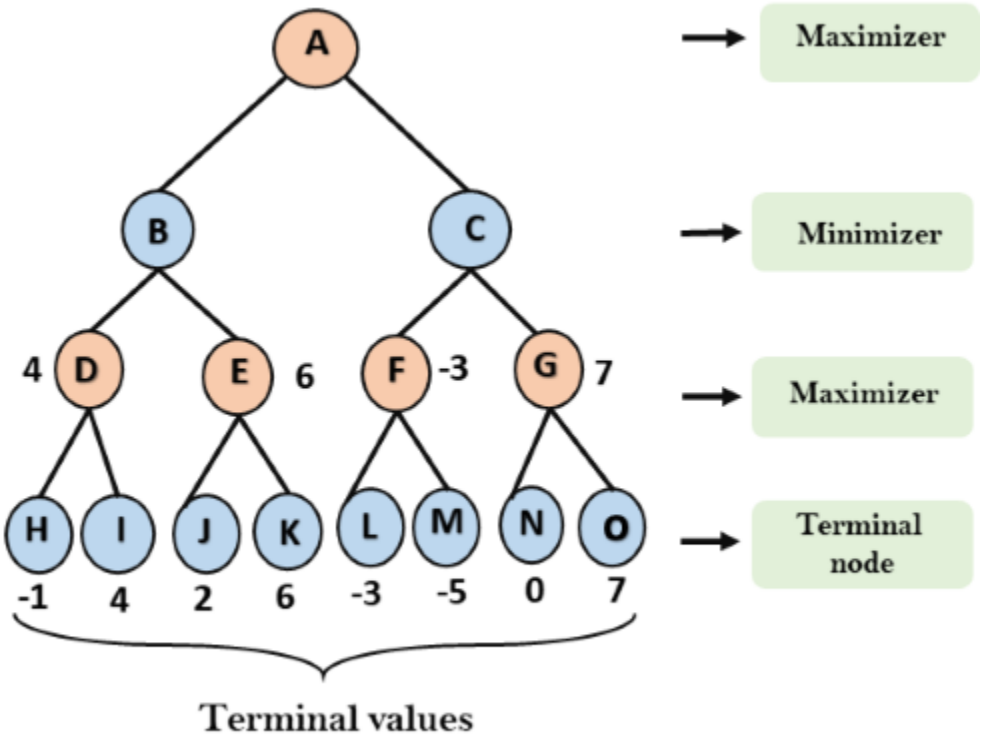
Cách chạy của thuật toán như sau:

Độ sâu của cây từ trên xuống dưới lần lượt là 0, 1, 2, 3.

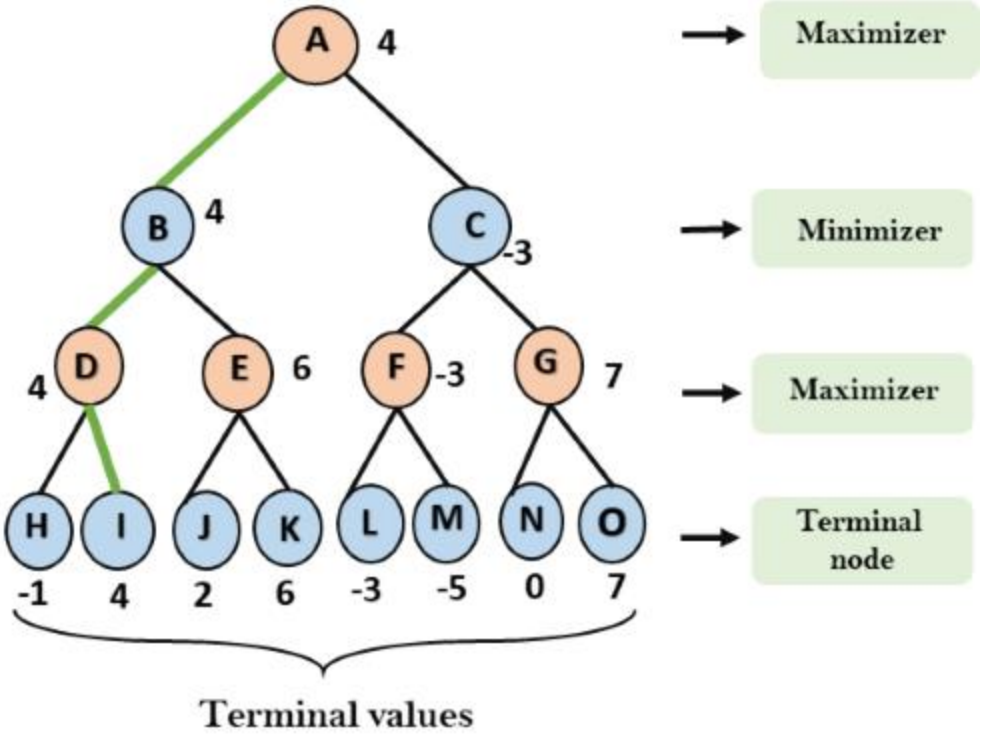
- B1: đệ quy đến lá của cây Minimax, ta sẽ nhận được các giá trị tại độ sâu 3: -1, 4, 2, 6, -3, -5, 0, 7.



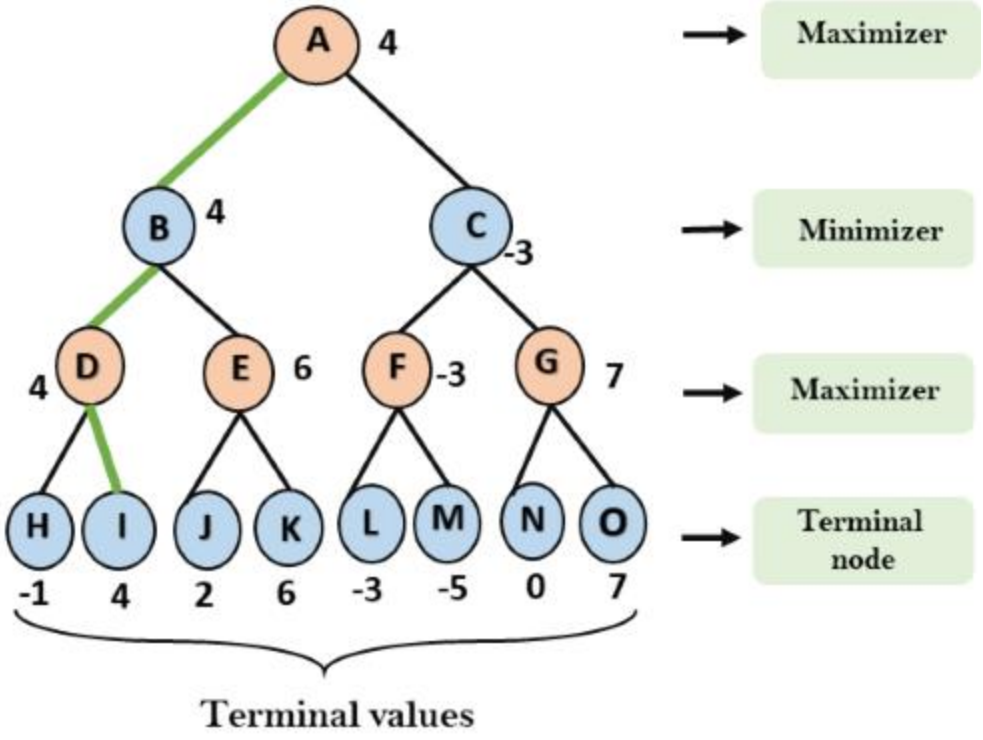
- B2: ở độ sâu 2 yêu cầu phải lấy node con nào có giá trị lớn nhất (Maximize), sau đó gán giá trị đó vào cho các node cha.



- B3: tương tự như B2 nhưng thay vì lấy giá trị lớn nhất thì ta sẽ lấy giá trị nhỏ nhất (Minimize), sau đó lại gán giá trị đó cho node cha.



- B4: tương tự như B2.



Các độ sâu liền kề sẽ lấy giá trị Maximize hoặc Minimize. Ví dụ độ sâu 0 đã lấy giá trị Maximize thì độ sau 1 phải lấy giá trị Minimize và ngược lại.

Thuật toán thường được dùng trong các trò như: cờ vua, tic – tac – toe.

b. Thuật toán Alpha – Beta Pruning

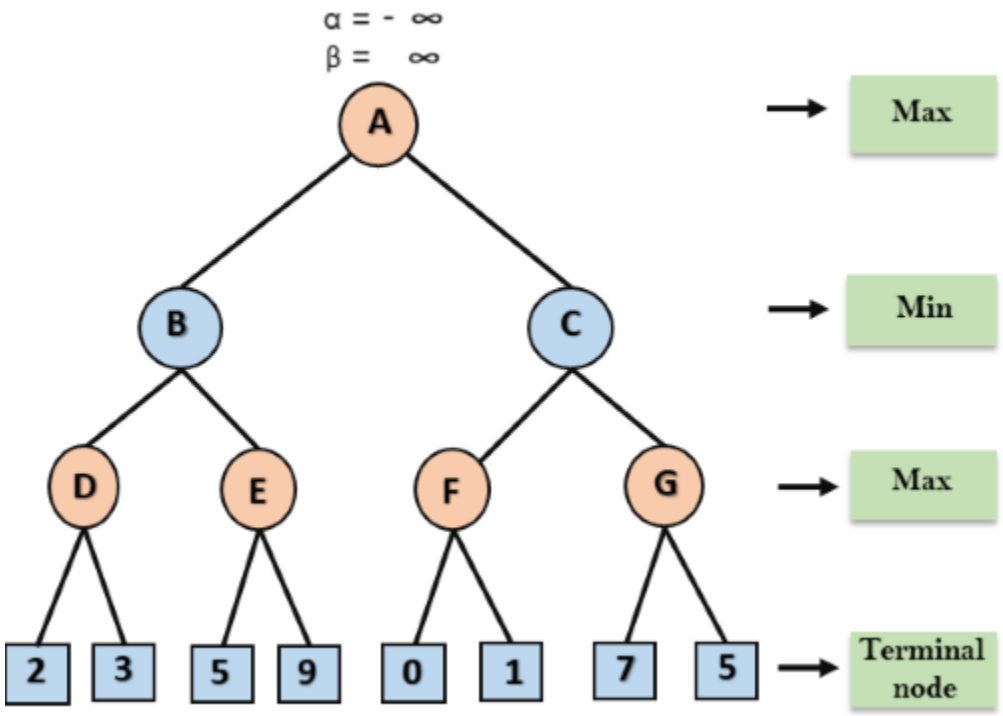
Thuật toán Minimax khi chạy sẽ chạy hết tất cả các node lá theo cơ chế DFS vì vậy sẽ rất tốn tài nguyên. Alpha – Beta Pruning là phiên bản cải tiến của Minimax nhằm mục đích tối ưu hoá hơn bằng cách cắt bỏ bớt các nhánh không cần thiết phải xét đi.

Thuật toán gồm tham số là Alpha: lưu giá trị Maximize, Beta: lưu giá trị Minimize. Nếu như Alpha >= Beta thì cắt bỏ nhánh.

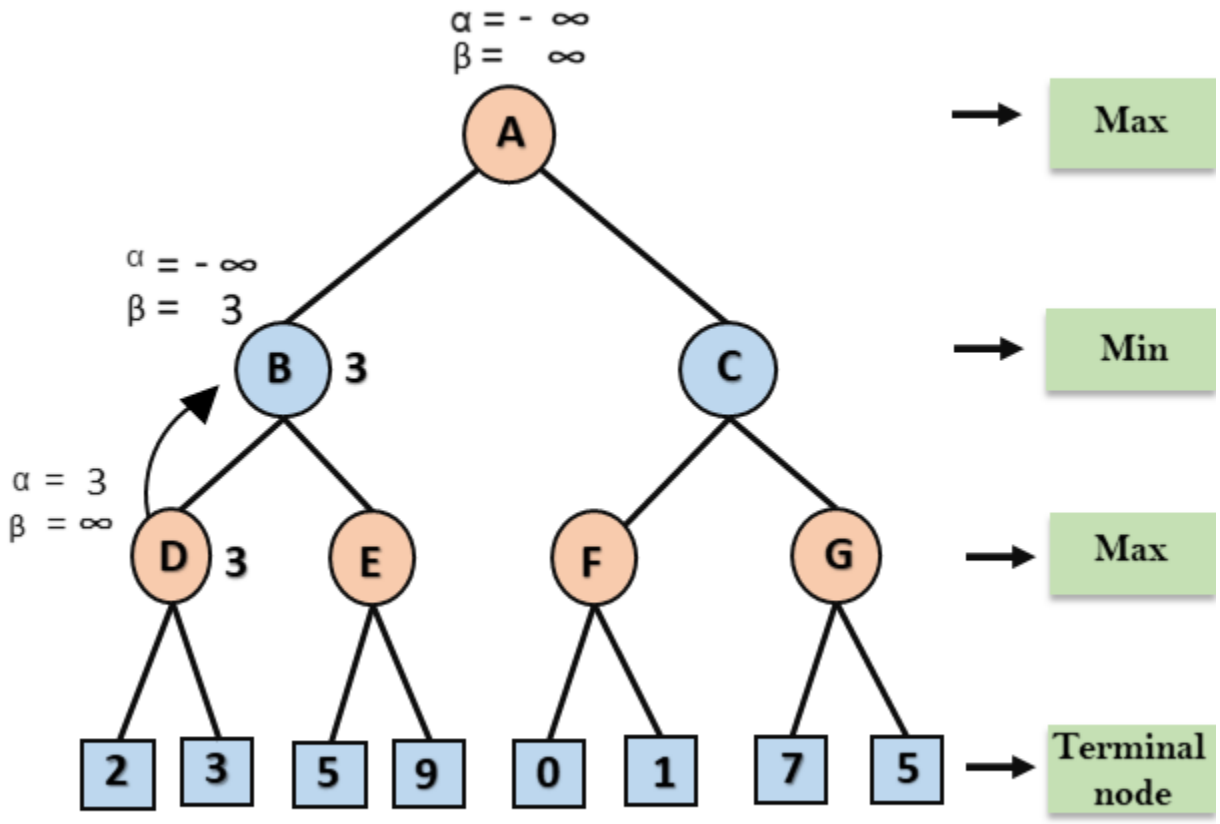
Cách chạy của thuật toán như sau:

Độ sâu của cây từ trên xuống dưới lần lượt là 0, 1, 2, 3.

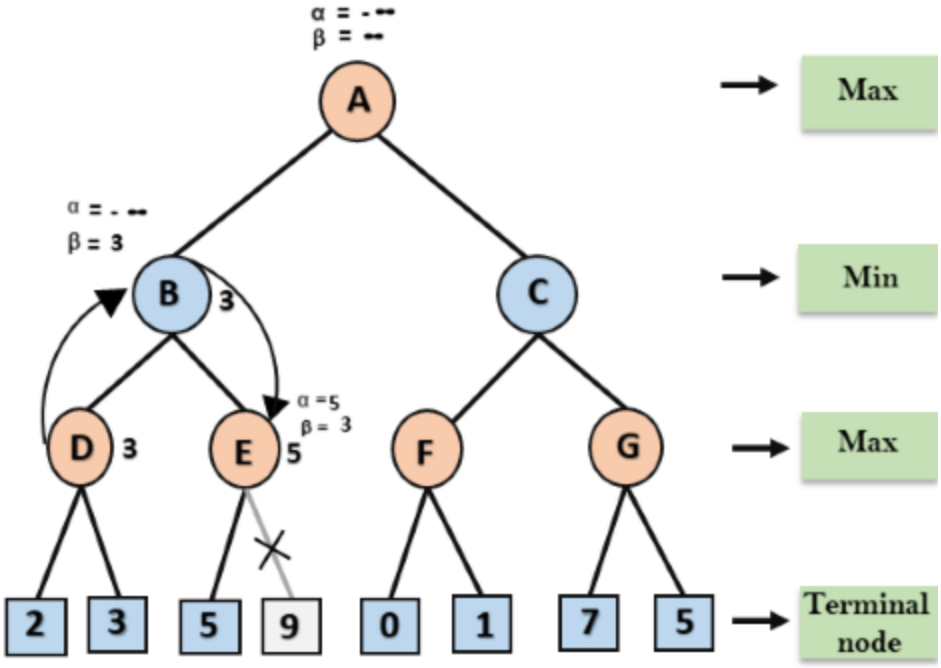
- B1: khởi tạo $Alpha = -\infty$ và $Beta = +\infty$, đệ quy đến lá của cây Minimax, ta sẽ nhận được các giá trị tại độ sâu 3: 2, 3, 5, 9, 0, 1, 7, 5.



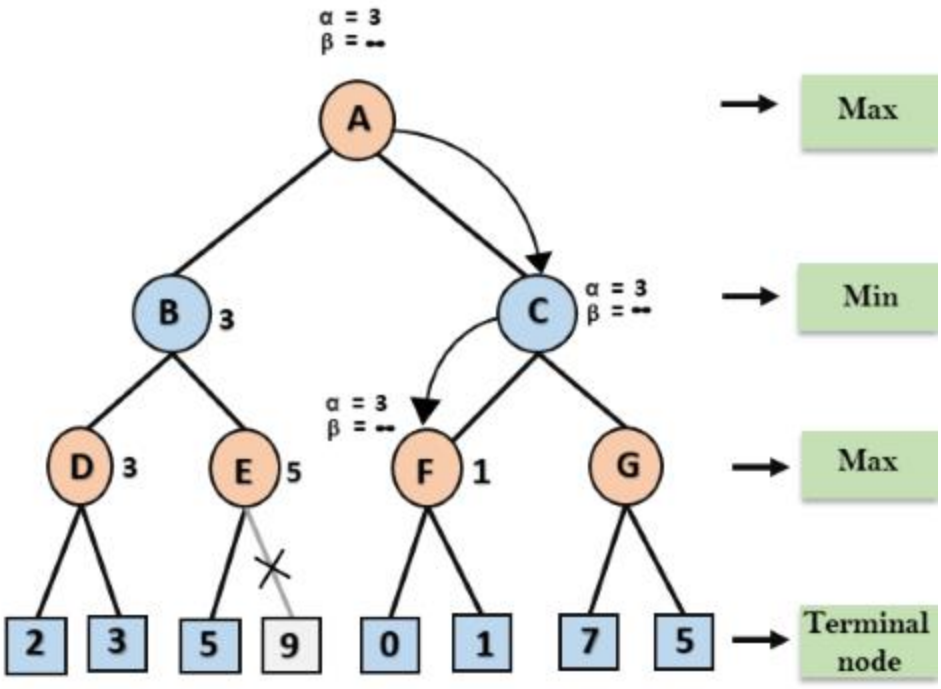
- B2: Ở độ sâu 2 yêu cầu node con lấy giá trị Maximize, gán vào cho Node cha. Ở node D sẽ lấy vào gán vào giá trị 3, Alpha cập nhật giá trị 3.
- B3: backtracking lên node cha của D là B, ở độ sâu này yêu cầu lấy giá trị Minimize nên cập nhật giá trị node B = 3, Beta = 3.



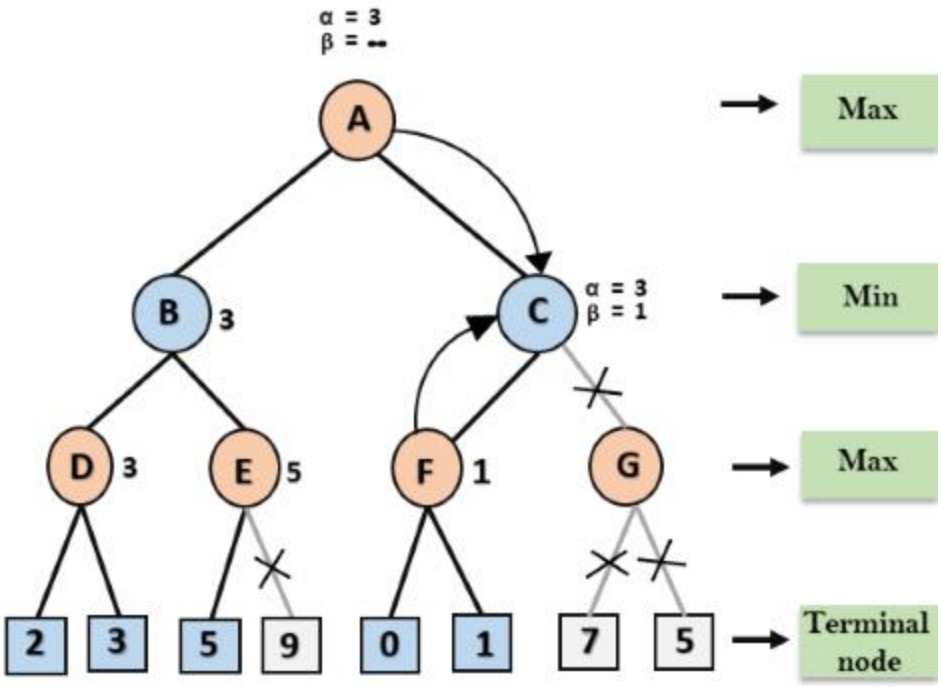
- B4: xét node con còn lại của B là E, node E lúc nào có giá trị Alpha là 5 và Beta là 3. Lúc này Alpha > Beta (5 > 3) nên ta sẽ cắt bỏ nhánh còn lại của E.



- B5: so sánh giá trị node B và E thì thấy giá trị 3 nhỏ hơn 5 nên không cập nhập.
- B6: backtracking lên node cha của B là A, ở độ sâu này yêu cầu lấy giá trị Maximize nên cập nhật giá trị node A = 3, Alpha = 3.
- B7: đệ quy xuống node F, giá trị của F lúc này là 1, tuy nhiên do 1 vẫn nhỏ hơn so với giá trị Alpha lúc này là 3 nên k cập nhật.



- B8: backtracking lên node cha của F là C, C sẽ mang giá trị = 1 và Beta = 1. Alpha > Beta (3 > 1) nên ta sẽ cắt bỏ nhánh con còn lại của node C.
- So sánh giá trị 3 và 1 thì thấy 3 lớn hơn nên giá trị của node A không đổi, kết thúc thuật toán.



Rõ ràng ta có thể thấy so với Minimax, thuật toán Alpha – Beta có cải thiện hơn khá nhiều.

II. Thuật toán tìm đường A*

1. Thuật toán A*

open_lst là 1 Priority Queue để lưu các tạo độ đã được duyệt và dùng để mở rộng tìm kiếm đường đường đi với giá trị priority là giá trị f(n) = g(n) + h(n)

List **G_Score** là 1 ma trận cùng kích thước ma trận **bitmapArray** giá trị mặc định là infinity, để lưu giá trị g(n) của từng tọa độ.

Visitor là biến để đếm số điểm đã tương tác.

List **Path** dùng để lưu tập đường đi.

- ***PSEUDOCODE***:

Khởi tạo open_lst, G_Score, visitor.

G_Score of start = 0

visitor = 1.

Thêm start vào open_lst với priority = heuristic of start

While open_lst is not empty {

Lấy phần tử đầu tiên của open_lst (phần tử có priority nhỏ nhất) gán vào current.

If current == goal {

totalCost = G_Score of current

Khởi tạo vòng lập để cập nhật Path.

Return Path, totalCost, Visitor

}

Tạo list Successor lưu tạo độ các điểm lân cận có thể đi qua, đồng thời cập nhật visitor

Duyệt từng phần tử của Successor {

successor_current_cost = G_Score of current + distance (successor, current)

If successor_current_cost < G_Score of successor {

Cập nhật G_Score of successor = successor_current_cost

H_Score = heuristic of successor

Thêm phần tử successor vào open_lst với priority = G_Score of successor + H_Score

Cập nhật tạo độ cha của successor = current

}

}

}

Return None, None, Visitor

- **Ưu điểm**

A* là một dạng thuật toán Informed Search, tức là có 1 trọng số dự đoán đường đi tới đích nên khi so sánh với các thuật toán Uninformed như BFS, DFS, UCS,… A* sẽ tối ưu hơn khá nhiều khi giải quyết các đường đi phức tạp.

Có thể hoàn thành thuật toán mà không chạy vô tận nếu tất cả các bước thực hiện là một số hữu hạn. Tối ưu với điều kiện hàm heuristic admissible hoặc consistent.

❖ **Nhược điểm**

Nhược điểm lớn nhất cũng là ưu điểm tốt nhất của thuật toán A* đó là hàm heuristic. Nếu hàm heuristic không đạt 1 trong 2 điều kiện đã nói ở trên, thuật toán sẽ không tối ưu và tìm đường đi dài hơn, phải đi qua nhiều node hơn so với thực tế.

❖ **Cải thiện**

Hàm heuristic tốt là khi trị số của nó gần với đường đi thực tế nhất nhưng vẫn nhỏ hơn đường đi thực tế $h \leq h^*$ (hay còn được biết đến làm hàm heuristic admissible). Nếu bỏ qua trường hợp không admissible, vấn đề còn lại là hàm heuristic quá nhỏ so với đường đi thực tế, dẫn đến trọng số dự đoán bị sai.

Để khắc phục bớt phần nào, ta có biến thể của thuật toán A* là Weighted A*. Trong công thức Weighted A* chọn đường đi tối ưu sẽ nhân thêm 1 trọng số w để tăng độ lớn của hàm Heuristic lên.

Công thức tổng quát:

$$f(n) = g(n) + w(n) \times h(n)$$

2. Mã nguồn

Trước khi sử dụng mã nguồn phải cài các thư viện sau:

Numpy: pip install numpy

Sympy: pip install sympy

PIL: pip install pillow

Installation — Pillow (PIL Fork) 8.3.1 documentation

a. Bitmap

Dữ liệu input là file hình ảnh dưới định dạng .bmp, để xử lí file hình ảnh ta dùng thư viện PIL (Python Imaging Library).

Để mở ảnh ta sử dụng Image.open(<đường dẫn file hình>). Vì file ảnh đang dạng RGBA nên trước khi dùng ta phải convert về Grayscale bằng cách sử dụng convert("L"), rồi chuyển bitmap thành ma trận số. Ma trận lúc này sẽ chứa các phần tử từ 0 – 255 đại diện cho màu của mỗi pixel.

```
bitmap = Image.open(self.bitmapPath)
bitmap = bitmap.convert("L")
self.__bitmapArray = np.asarray(bitmap)
bitmap.close()
```

b. Coordinate

Class Coordinate dùng để lưu toạ độ x và y. Chứa các phương thức so sánh, tính khoảng cách, kiểm tra giữa 2 toạ độ có thể di chuyển qua lại hay không.

c. Astar

- B1: Khởi tạo hai biến row và col là số hàng và số cột của ma trận thể hiện giá trị độ cao ứng với mỗi tọa độ.
- B2: Khởi tạo open_lst là 1 priority queue để lưu các tạo độ đã được duyệt để tìm kiếm đường đi. Ta thêm cái tọa độ vào open_lst với giá trị priority là giá trị của f() = g() + h(). Việc này nhằm mục đích khi lấy phần tử đầu tiên của queue thì phần tử này luôn có giá trị f() nhỏ nhất.
- B3: Thêm start vào tập open_lst, priority of start = h(start)
- B4: Khởi tạo list G_Score có số dòng, cột ứng với row và col và mang giá trị mặc định là infinity. List này nhằm mục đích lưu giá trị g() của các tạo độ khi duyệt qua và cũng sử dụng để tính số điểm đã tương tác.
- B5: Khởi tạo G_Score of start = 0. Khởi tạo biến điếm số điểm tương tác là visitor = 1 (Chính là điểm start).
- B6: Tạo vòng lặp kiểm tra open_lst đã rỗng hay chưa. Nếu open_lst vẫn còn phần tử có nghĩa là các tạo độ cần được duyệt vẫn chưa hoàn tất tiến tới B7. Nếu đã duyệt hết các phần tử rồi nhưng hàm vẫn không trả về kết quả đường đi thì thông báo không tìm được đường và kết thúc.

- B7: Trong vòng lặp trên, tiến hành lấy phần tử đầu tiên từ open_lst gắn vào current. Kiểm tra current có phải là goal hay không. Nếu không thì bỏ qua. Nếu có thì khởi tạo biến totalCost để lưu độ dài đường đi từ start đến goal chính là giá trị G_Score of current. Thực hiện vòng lặp truy cập về tạo độ cha của tọa độ hiện tại để xác định đường đi. Nếu current không phải start thì gắn current = parents of current. Kết thúc vòng lặp thông báo hoàn tất và trả về tập đường đi, độ dài đường đi và số điểm đã tương tác.
- B8: Khởi tạo tập các tạo độ con Successor bằng cách gọi hàm getSuccessor. Trong hàm getSuccessor, thực hiện duyệt qua 8 tọa độ lân cận của current. Nếu tọa độ đó có giá trị nhỏ hơn 0 hoặc lớn hơn số dòng hoặc số cột tương ứng thì bỏ qua và tiếp tục vòng lặp xét tạo độ khác. Nếu tọa độ lân cận thỏa mãn điều kiện trên, kiểm tra gái trị G_Score của tọa độ này. Nếu bằng infinity thì cộng giá trị visitor lên 1 (tọa độ chưa tương tác), nếu khác infinity có nghĩa là tọa độ đã tương tác → không tăng giá trị visitor. Tiếp tục khởi tạo điểm ứng với tọa độ trên, kiểm tra điều kiện current có đi đến điểm vừa tạo được hay không, nếu thỏa mản thì thêm điểm này vào list Successor. Kết thúc vòng lặp trả về list Successor và visitor để cập nhật vào hàm Astar.
- B9: Khi có được tập Successor, duyệt qua từng phần tử:

Khởi tạo biến succ_current_cost = G_Score of current + distance(current, succ)

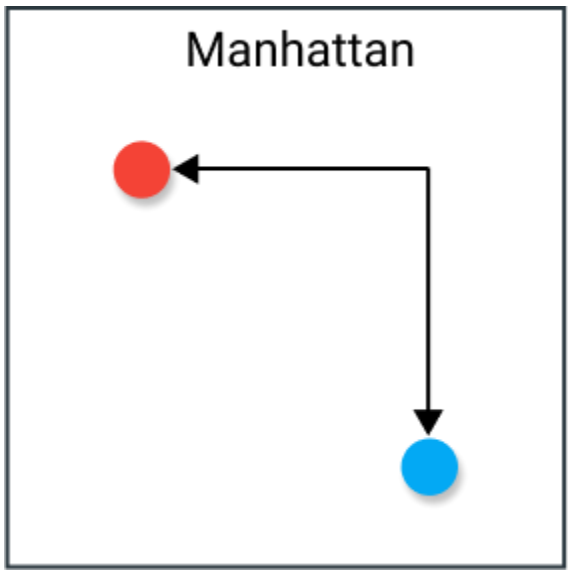
Đây là giá trị g() mới của succ.

Kiểm tra succ_current_cost có nhỏ hơn giá trị G_Score of succ hay không. Nếu không thì bỏ qua, tiếp tục vòng lặp. Nếu có thì cập nhật giá trị G_Score of succ bằng giá trị succ_current_cost. Khởi tạo H_Score of succ = h(succ). Thêm succ vào tập open_lst với priority = G_Score of succ + H_Score of succ. Cập nhật tọa độ cha của succ là current.

3. Hàm heurisctic

a. Manhattan

Khoảng cách **Manhattan**, còn được gọi là khoảng cách L1 hay khoảng cách trong thành phố, là một dạng khoảng cách giữa hai điểm trong không gian Euclid với hệ tọa độ Descartes. Đại lượng này được tính bằng tổng chiều dài của hình chiếu của đường thẳng nối hai điểm này trong hệ trục tọa độ Descartes.



Nguồn: [9 Phép đo Khoảng cách trong Khoa học Dữ liệu (ichi.pro)]

Công thức tổng quát: $D(x,y) = \sum_{i=1}^{n} |x_i - y_i|$

Công thức dùng để tính hàm heuristic trong đồ án:

$$D(a_1, a_2) = |a_1.x - a_2.x| + |a_1.y - a_2.y|$$

Dựa vào ý tưởng của công thức khoảng cách Manhattan, nhóm đã cải tiến công thức dùng để tính hàm heuristic cho đồ án:

$k = |m - D(a_1, a_2)|$.

$k = k + 1$ *cho đến khi* k *là* 1 *số nguyên tố.*

$=> h_{CustomManhattan} = D(a_1, a_2) + k$

Trong đó m là hằng số kiểm tra độ chênh lệch độ cao.

❖ So sánh:

Ta sẽ xét trường hợp đường dài của thuật toán, đi từ gốc trái trên cùng xuống gốc phải dưới cùng.

Text > input.txt

```
1   (0;0)
2   (511;511)
3   10
```

➢ Hàm Manhattan:

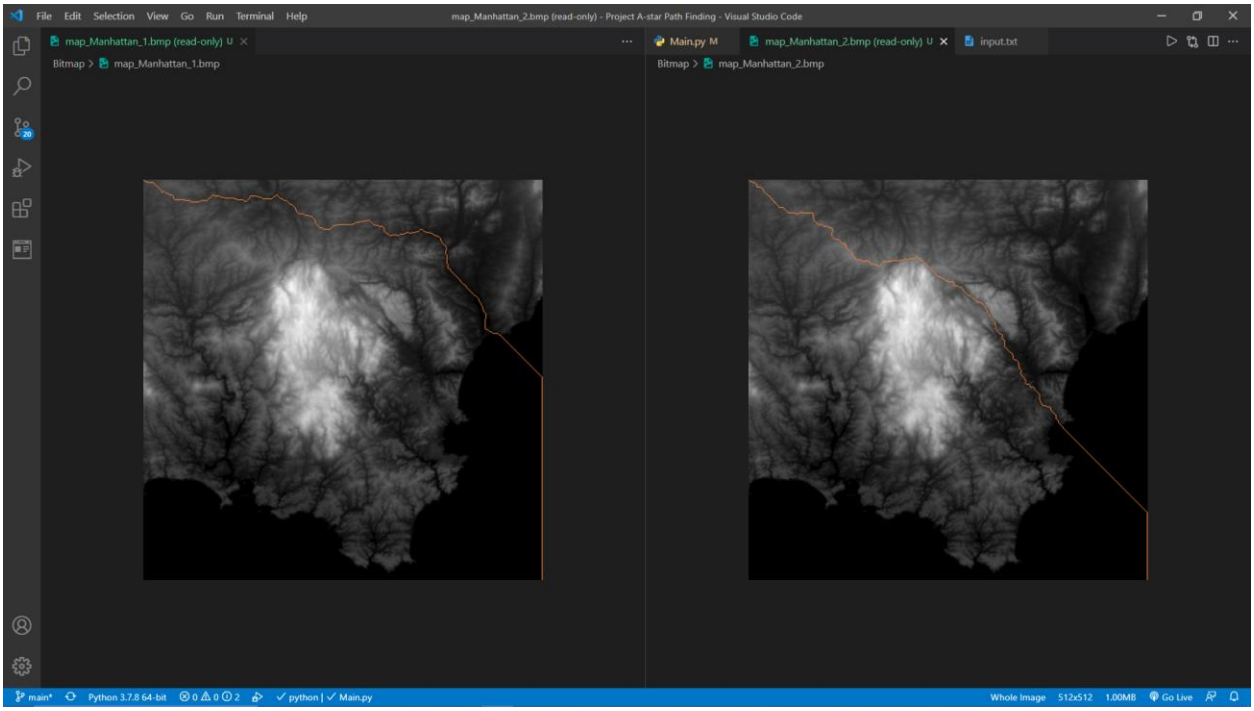
Đường đi ngắn hơn, tốn ít thời gian hơn nhưng số điểm chạm lại nhiều hơn.

```
Done!
Total distance:1270.23715677735
Total points: 92831
Time cost: 11.471019983291626
```

➢ Hàm CustomManhattan:

Ngược lại so với Manhattan, đường đi dài hơn, tốn nhiều thời gian hơn nhưng số điểm chạm lại ít hơn rất nhiều.

```
Done!
Total distance:1551.6585130146607
Total points: 35257
Time cost: 31.609899044036865
```



Hàm Manhattan (hình trái) có xu hướng đi vòng cung, bám biên trong khi hàm CustomManhattan (hình phải) lại có xu hướng đi chéo về phía Goal.

Trường hợp đường ngắn:

```
1   (74;213)
2   (96;311)
3   10
```

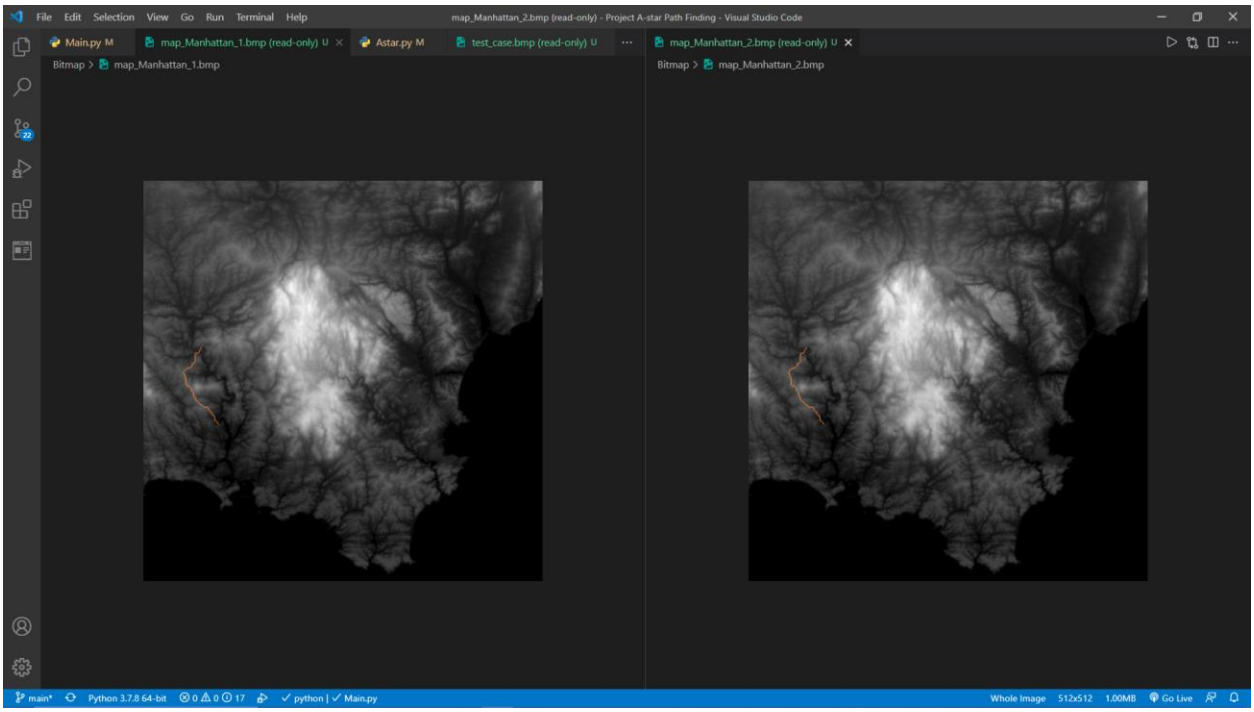
- Hàm Manhattan:

```
Done!
Total distance:317.5391052434012
Total points: 15871
Time cost: 1.8045156002044678
```

- Hàm CustomManhattan:

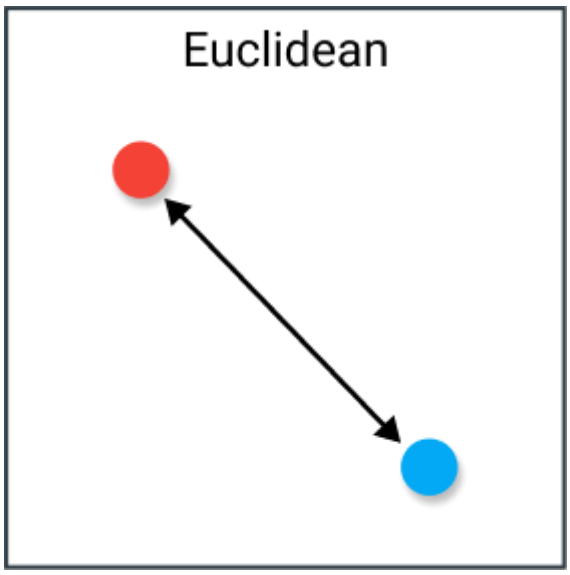
```
Done!
Total distance:317.5391052434012
Total points: 7962
Time cost: 2.156965494155884
```

Đường đi của 2 hàm là như nhau, thời gian tốn không chênh lệch đáng kể, tuy nhiên số điểm tương tác giữa 2 hàm chênh lệch thấy rõ.



b. Euclid

Công thức của khoảng cách Euclid chính là công thức tính độ lớn của vector.



Nguồn: [9 Phép đo Khoảng cách trong Khoa học Dữ liệu (ichi.pro)]

Công thức tổng quát: $D(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$

Công thức dùng để tính hàm heuristic trong đồ án:

$$D(a_1,a_2) = \sqrt{(a_1.x - a_2.x)^2 + (a_1.y - a_2.y)^2}$$

Tương tự như hàm CustomManhattan, CustomEuclid cũng sẽ cộng thêm 1 số nguyên tố k:

$k = |m - D(a_1,a_2)|$.

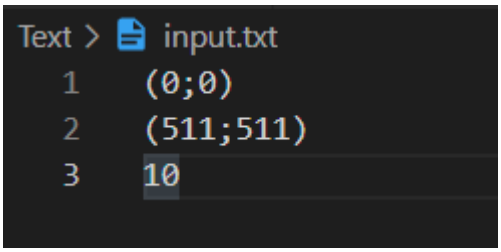
$k = k + 1$ *cho đến khi* k *là* 1 *số nguyên tố.*

$=> h_{CustomEuclid} = D(a_1,a_2) + k$

Trong đó m là tham số kiểm tra độ chênh lệch độ cao.

❖ So sánh:

Ta sẽ xét trường hợp đường dài của thuật toán, đi từ gốc trái trên cùng xuống gốc phải dưới cùng.



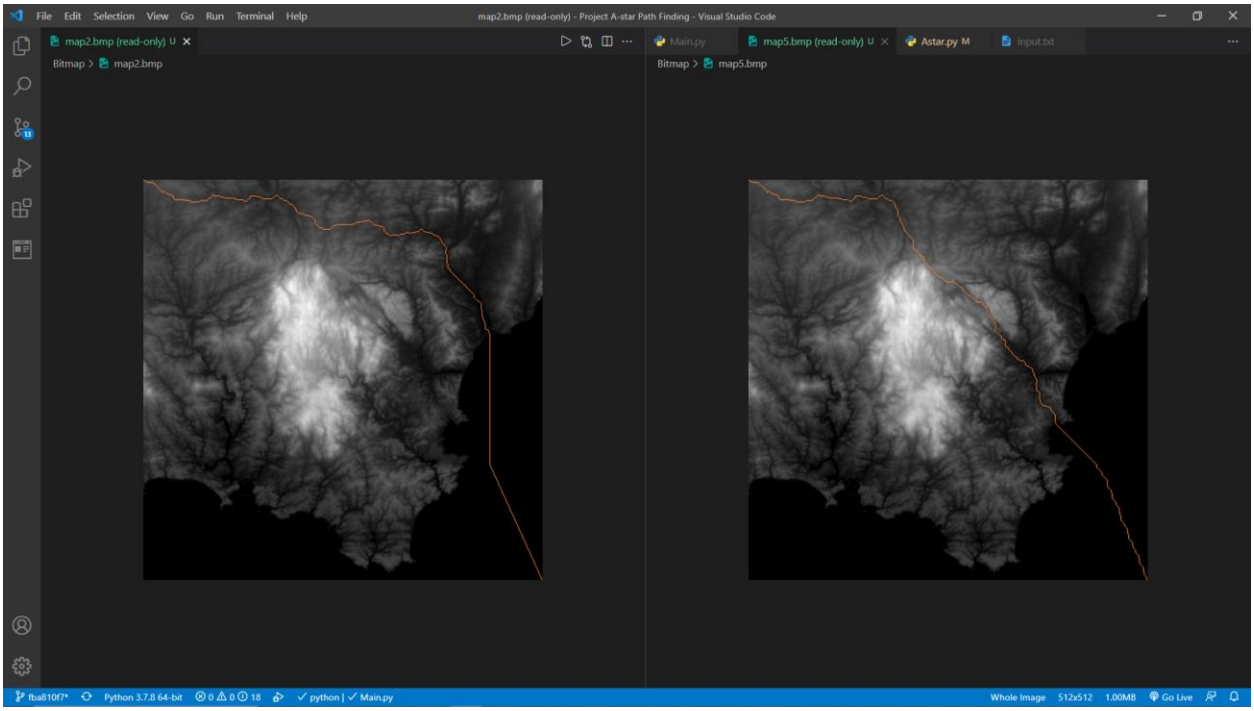
- Hàm Euclid:

```
Done!
Total distance:1264.1366517139618
Total points: 169515
Time cost: 16.974458694458008
```

- Hàm CustomEuclid:

```
Done!
Total distance:1374.5874452027952
Total points: 110114
Time cost: 26.300299406051636
```

Độ dài đường đi cũng như thời gian tiêu tốn chênh lệch nhiều, nhưng số điểm tương tác chênh lệch thấy rõ.



Hàm Euclid (hình trái) Hàm CustomEuclid (hình phải)

Trường hợp đường đi ngắn:

```
1  (74;213)
2  (96;311)
3  10
```

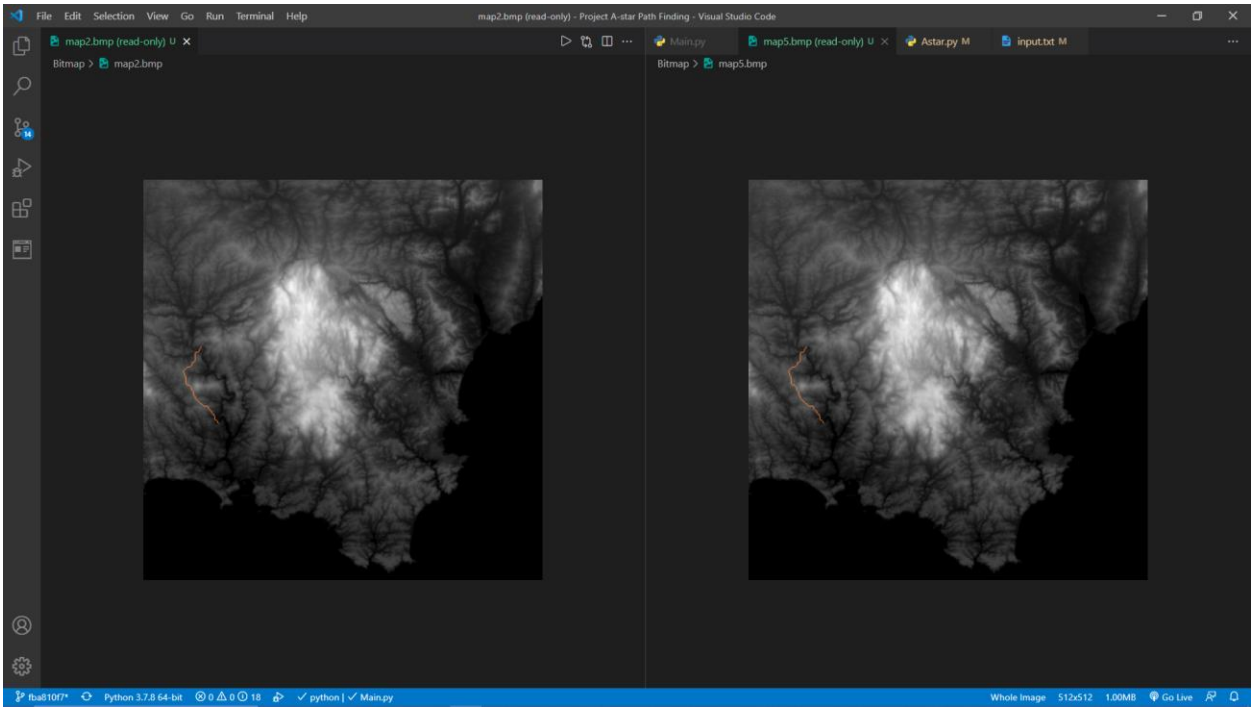
- Hàm Euclid:

```
Done!
Total distance:317.5391052434012
Total points: 23618
Time cost: 2.1069180965423584
```

- Hàm CustomEuclid:

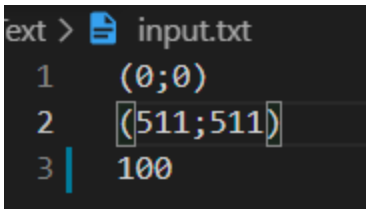
```
Done!
Total distance:317.5391052434012
Total points: 13929
Time cost: 1.5113754272460938
```

Độ dài đường đi là như nhau, số điểm chạm và thời gian tiêu tốn của hàm Custom tối ưu hơn so với Euclid.

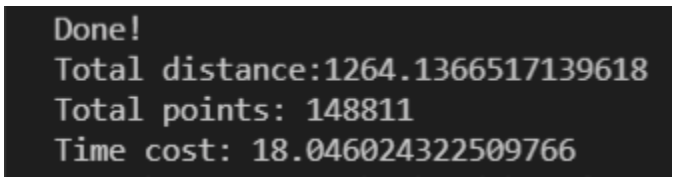


Hàm Euclid (hình trái) Hàm CustomEuclid (hình phải)

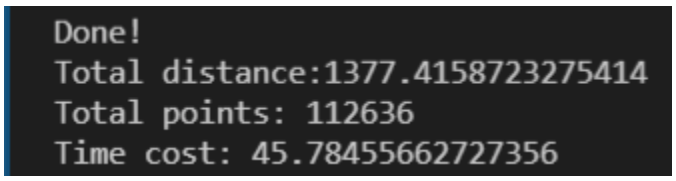
- Do các hàm Custom đề phụ thuộc vào hằng số m kiểm tra chênh lệch độ cao nên khi thay đổi tham số cũng có ảnh hưởng đến hiệu năng của thuật toán. Ví dụ hàm Euclid và CustomEuclid:

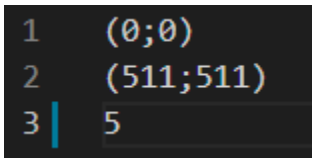


- Hàm Euclid:

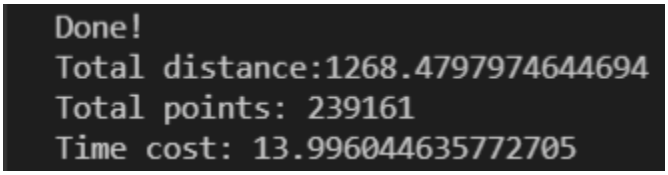


- Hàm CustomEuclid:

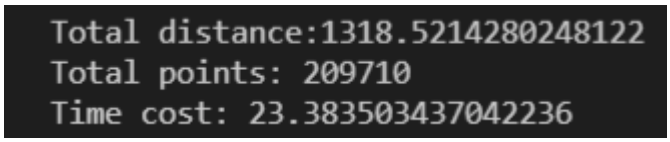




- Hàm Euclid:



- Hàm CustomEuclid:



Rõ ràng khi m càng nhỏ thời gian tìm kiếm càng nhanh và số điểm tương tác cũng tăng đáng kể. Tuy nhiên, có những hàm lại tìm được đi ngắn hơn chẳng hạn như CustomEuclid m càng lớn thì đường đi càng dài.

c. Circle Area

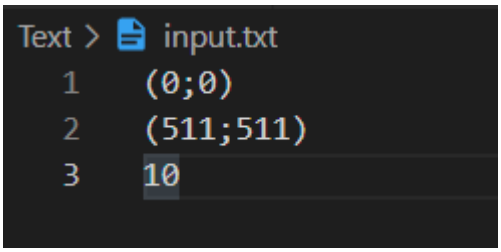
Thuật toán A* khi expand node về phía goal theo hình Elip nên ý tưởng diện tích hình Elip làm hàm heuristic. Tuy nhiên để tính diện tích hình Elip sẽ khá phức tạp thay vào đó ta sử dụng công thức tính diện tích hơi giống với Elip là hình tròn.

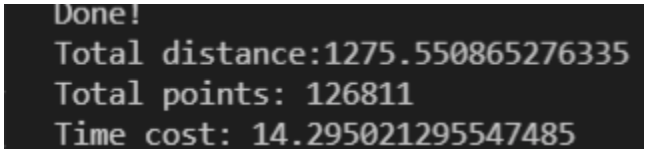
Công thức dùng để tính hàm heuristic trong đồ án:

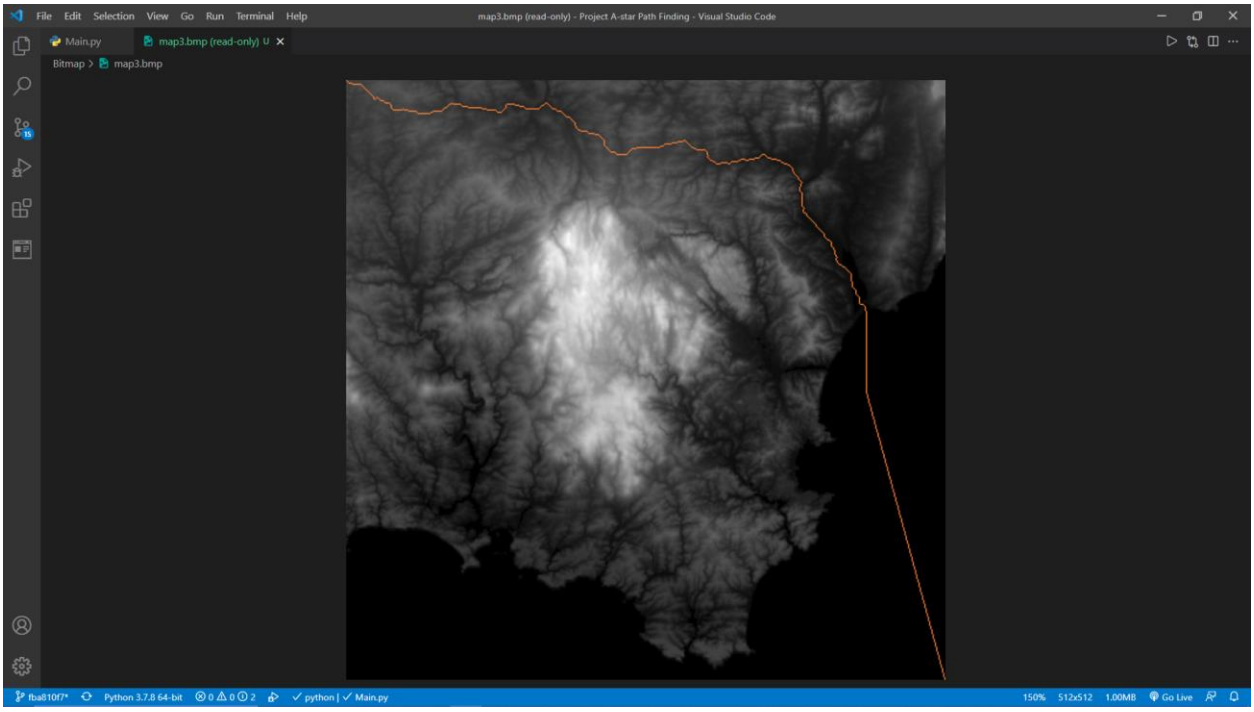
$$D(a_1, a_2) = \sqrt{(a_1.x - a_2.x)^2 + (a_1.y - a_2.y)^2}$$

$$h_{CA} = \pi \frac{D(a_1, a_2)}{2}$$

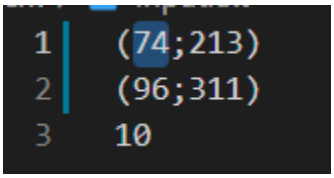
Ta sẽ xét trường hợp đường dài của thuật toán, đi từ gốc trái trên cùng xuống gốc phải dưới cùng.

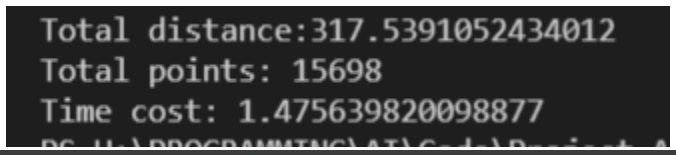


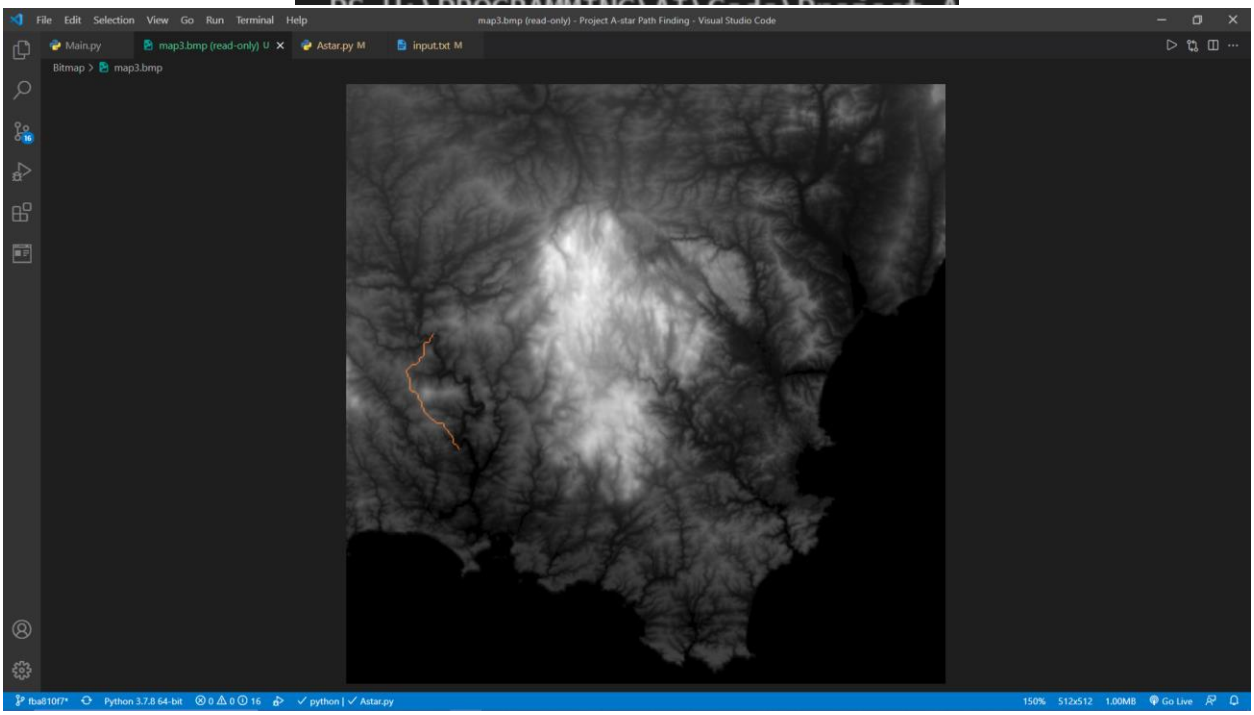




Trường hợp đường ngắn:







❖ **Tóm lại**

Các hàm Custom đường dài thường có xu hướng đường đi bám sát vào vách núi. Số điểm tương tác của các hàm Custom luôn luôn ít hơn so với các hàm gốc nhưng run-time lại cao hơn. Một vài trường hợp lại cho đường đi cùng với điểm tương tác ít hơn.

Hàm Manhattan cho tiêu tốn ít thời gian chạy và tổng số điểm tương tác hơn so với Euclid nhưng đường đi lại không tối ưu bằng Euclid.

Hàm CA cho kết quả tối ưu nhất so với 2 hàm Euclid và Manhattan trong trường hợp ngắn. Trong trường hợp tìm đường đi dài, CA có tổng số điểm tương tác không tối ưu bằng Manhattan. Đường đi không tối ưu bằng Euclid và

Manhattan nhưng không lệch quá nhiều. Tuy nhiên, CA lại có tính ổn định cao hơn so với Euclid vàManhattan.

- **LƯU Ý:** Các hàm heuristic trên đều do thực nghiệm và cho kết quả chấp nhận được.

III. Mở rộng

Kỹ thuật tìm đường đi tối ưu nhất được áp dụng rộng rãi trong lĩnh vực phát triển game hiện nay, điển hình là các game có tính đối kháng với máy (bot) như: FIFA, Chess, Street Fighter,… nhằm tìm ra state tốt nhất cho bot để đối đầu với người chơi, tạo cảm giác đối phó khó khăn hơn so với đối đầu giữa người và người. A* là một trong số các thuật toán thường được sử dụng trong lĩnh vực này.

IV. Nguồn tham khảo

Variants of A* (stanford.edu)

Optimized Application and Practice of A* Algorithm in Game Map Path-Finding | IEEE Conference Publication | IEEE Xplore

9 Phép đo Khoảng cách trong Khoa học Dữ liệu (ichi.pro)

https://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf

A* search algorithm - Wikipedia

Implementation of A* (redblobgames.com)