**TRƯỜNG ĐẠI HỌC THỦ DẦU MỘT**

**VIỆN KỸ THUẬT - CÔNG NGHỆ**

****

**TIỂU LUẬN GIỮA KỲ**

**MÔN HỌC:**

**NHẬP MÔN TRÍ TUỆ NHÂN TẠO**

**Topic 8 : THUẬT TOÁN LEO ĐỒI**

**GVHD : Lê Thị Hiền Duyên**

**NHÓM THỰC HIỆN**

|  |  |
| --- | --- |
| ***Bùi Thành Được*** | ***1824801030100*** |
| ***Mai Văn Chánh*** | ***1824801030028*** |

*Bình Dương, tháng 03 năm 20*

## 1. khái niệm về thuật toán leo đồi

Trong khoa học máy tính, giải thuật Hill Climbing là một kỹ thuật tối ưu toán học thuộc họ tìm kiếm cục bộ. Nó thực hiện tìm một trạng thái tốt hơn trạng thái hiện tại để mở rộng. Để biết trạng thái tiếp theo nào là lớn hơn, nó dùng một hàm H để xác định trạng thái nào là tốt nhất.

Hill Climbing dễ dàng tìm thấy một giải pháp tốt cục bộ (local optimum) nhưng khó tìm thấy giải pháp tốt nhất (global optinum) trong tất cả các giải pháp được đưa ra (search space). Hill Climbing phù hợp để giải các bài toán “convex” (dịch tạm: lồi) như là tìm kiếm đơn giản (simplex programming) trong lập trình tuyến tính, tìm kiếm nhị phân.

Tính đơn dơn giản của giải thuật khiến nó trở thành lựa chọn đầu tiên trong số các giải thuật tối ưu. Nó được sử dụng rất nhiều trong trí tuệ nhân tạo, dùng cho mục đích đi đến trạng thái đích từ một node bắt đầu. Việc chọn node tiếp theo và node bắt đầu có thể thay đổi nhiều giải thuật khác nhau.

Hill climbing là một kỹ thuật tối ưu toán học (mathematical optimization).

Trong toán học, khoa học máy tính (computer science) và vận động học (operation research), tối ưu toán học (mathematical optimization, cũng gọi là optimization hay mathematical programing) là sự lựa chọn một thành phần tốt nhất (liên quan đến một vài tiêu chuẩn) từ tập hợp các lựa chọn có sẵn.

Các giải thuật thuộc loại tìm kiếm tối ưu như là: Hill Climbing, Stimulate Annealing, Generic Algorithm.

* Hill climbing thuộc họ tìm kiếm cục bộ (local search).

Trong khoa học máy tính, tìm kiếm cục bộ là một phương pháp heuristic để giải quyết các bài toán tối ưu khó. Tìm kiếm cục bộ có thể sử dụng trong các bài toán mà có thể được tính bằng cách tìm một giải pháp tối đa hóa một tiêu chí nào đó trong số các giải pháp được đưa ra. Giải thuật tìm kiếm cục bộ chuyển từ giải pháp này đến giải pháp khác trong không gian các giải pháp được đưa ra (không gian tìm kiếm) bằng cách áp dụng những thay đổi cục bộ cho đến khi một giải pháp được coi là tối ưu được tìm thấy hoặc thời gian giới hạn trôi qua.

Tìm kiếm cục bộ được áp dụng trong rất nhiều bài toán khác nhau, bao gồm các bài toán trong khoa học máy tính, toán học, tìm kiếm tối ưu, kỹ tuật xây dựng, sinh học. Ví dụ bài toán WalkSAT và giải thuật 2-opt trong bài toán người du lịch (Traveling Salesman Problem).

## 2. Cơ Chế Hoạt Động

B1: Xét trạng thái đầu: Nếu là đích => dừng

Ngược lại, thiết lập trạng thái bắt đầu = trạng thái hiện tại.

B2: Lựa một luật để áp dụng vào trạng thái hiện tại để sinh ra một trạng thái mới.

B3: Xem xét trạng thái mới này:

Nếu là đích => dừng.

Nếu không phải là đích nhưng tốt hơn trạng thái hiện tại thì thiết lập trạng thái hiệu t là trạng thái mới.

Nếu không tốt hơn thì đến trạng thái mới tiếp theo

Lặp đến khi: gặp đích hoặt không còn luật nào nữa chưa được áp dụng vào trạng thái hiện tại.

Bài toán được áp dụng là Leo đồi khởi động lại ngẫu nhiên là một [thuật toán tổng](https://en.wikipedia.org/wiki/Meta-algorithm) hợp được xây dựng dựa trên thuật toán leo đồi. Nó còn được gọi là leo đồi Shotgun . Nó lặp đi lặp lại việc leo đồi, mỗi lần với một điều kiện ban đầu ngẫu nhiên{\ displaystyle x\_ {0}}. Tốt nhất{\ displaystyle x\_ {m}} được giữ lại: nếu một đợt leo đồi mới tạo ra một {\ displaystyle x\_ {m}}hơn trạng thái được lưu trữ, nó thay thế trạng thái được lưu trữ.

Leo đồi khởi động lại ngẫu nhiên là một thuật toán hiệu quả đáng ngạc nhiên trong nhiều trường hợp. Nó chỉ ra rằng thường tốt hơn là dành thời gian cho CPU khám phá không gian hơn là tối ưu hóa cẩn thận từ điều kiện ban đầu.

## Xây dựng các lớp , hàm và thuật toán

* Tạo một lớp để lưu các trạng thái tìm đường

1. class State:
2. def \_\_init\_\_(self, route, distance, right):
3. self.route = route # mảng lưu vết các thành phố
4. self.right = right # biến trạng thái để biết đi đến đích chưa # Bucharest
5. self.distance = distance # mảng lưu khoảng cách của từng đoạn đường giữa các thành phố
6. # Tính khoảng cách từ điểm xuất phát đến đích
7. def Cal\_distance(self):
8. if self.right == False: return 99999 # nếu không tìm đến đích thì không tính khoảng cách
9. count = 0
10. for i in self.distance:
11. count += i
12. return count
13. # In ra quãng đường đi từ điểm xuất phát đến đích
14. def Print\_route(self, citylist):
15. if self.right == False:
16. print('not find path')
17. return # nếu không tìm đến đích thì báo không tìm thấy đường đi
18. for i in self.route:
19. print(citylist[i], end=' ')
20. # In ra mảng lưu khoảng cách giữa các thành phố đã đi qua
21. def print\_Distance(self):
22. print(self.distance)

* Tạo thuật toán leo đồi

def hill\_clibing(matrix, home, target):

    open = [] # mảng tạm đang xét

    close = [] # mảng lưu vết tìm đường

    distance = [0] # mảng lưu khoảng cách các thành phố đã đi

    right = False # biến kiểm tra xem đến đích chưa

    index = home # biến lưu thành phố cuối cùng đã đi qua

    close.append(index) # lưu thành phố bắt đầu đi

    if home == target: return State(close,distance,True)

    while True:

        # tìm các thành phố liền kề thành phố đang đứng cho vào mảng open

        for i in range(len(matrix)):

            if matrix[index][i] > 0:

                open.append(i)

        # xóa các thành phố đã đi qua trong mảng open

        for i in close:

            for j in open:

                if i == j:

                    open.remove(j)

        # nếu mảng open trống thì dừng lại

        if not open :break

        # chọn ngẫu nhiên 1 thành phố trong mảng open

        # lấy ra cho vào biến city

        city = random.choice(open)

        # lưu khoảng cách vào mảng distance

        distance.append(matrix[index][city])

        # lưu thành phố vào mảng close

        close.append(city)

        # di chuyển con trỏ đến thành phố vừa đến

        index = city

        # nếu thành phố vừa đến là đích cần tìm thì gắn cờ right = true và thoát vòng lặp

        if index == target:

            right = True

            break

        # xóa mảng open rồi lặp lại

        open.clear()

    # trả về quảng đường, khoảng cách và trạng thái đến đích

    return State(close, distance, right)

Tạo hàm tối ưu cho thuật toán

def optimize(matrix, home, target, citylist):

    # đặt trạng thái tốt nhất là trạng thái đầu tiền

    best\_state = hill\_clibing(matrix, home, target)

    # dùng vòng lặp để tìm ra trạng thái tốt hơn

    for i in range(10000):

        state = hill\_clibing(matrix, home, target)

        if state.Cal\_distance() < best\_state.Cal\_distance():

            best\_state = state

    # In kết quả

best\_state.Print\_route(citylist)

    print('\ndistance ' + str(best\_state.Cal\_distance()))

* Gán giá trị cho các vị trí đến

def main():

    Matrix = np.zeros((13,13))

    Matrix[0][1] = 71

    Matrix[0][12] = 75

    Matrix[1][0] = 71

    Matrix[1][2] = 151

    Matrix[2][1] = 151

    Matrix[2][3] = 99

    Matrix[2][6] = 80

    Matrix[2][12] = 140

    Matrix[3][2] = 99

    Matrix[3][4] = 211

    Matrix[4][3] = 211

    Matrix[4][5] = 101

    Matrix[5][4] = 101

    Matrix[5][6] = 97

    Matrix[5][7] = 138

    Matrix[6][2] = 80

    Matrix[6][5] = 97

    Matrix[6][7] = 146

    Matrix[7][5] = 138

    Matrix[7][6] = 146

    Matrix[7][8] = 120

    Matrix[8][7] = 120

    Matrix[8][9] = 75

    Matrix[9][8] = 75

    Matrix[9][10] = 70

    Matrix[10][9] = 70

    Matrix[10][11] = 111

    Matrix[11][10] = 111

    Matrix[11][12] = 118

    Matrix[12][0] = 75

    Matrix[12][2] = 140

    Matrix[12][11] = 118

    citylist = ['Zerind','Oradea','Sibiu','Fagaras','Bucharest','Pitesti','Rimnicu Vilcea','Craiova','Drobeta','Mehahadia','Lugoj','Timisoara','Arad']

    # 0   Zerind

    # 1   Oradea

    # 2   Sibiu

    # 3   Fagaras

    # 4   Bucharest

    # 5   Pitesti

    # 6   Rimnicu Vilcea

    # 7   Craiova

    # 8   Drobeta

    # 9   Mehahadia

    # 10  Lugoj

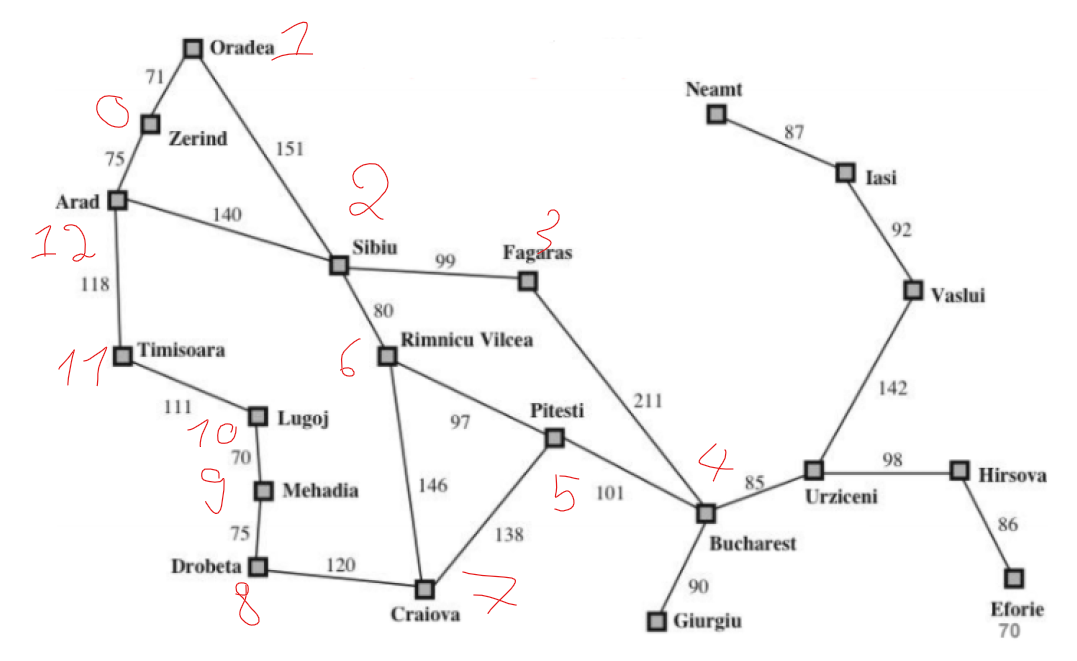
    # 11  Timisoara

    # 12  Arad

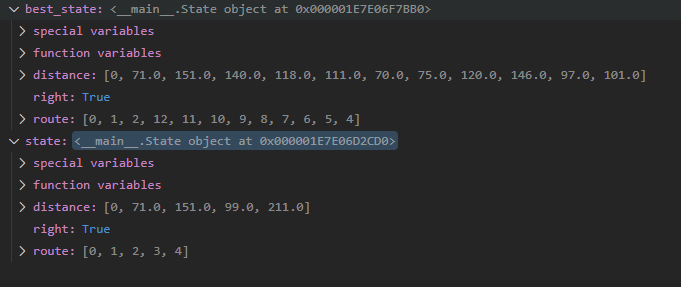
    optimize(Matrix, 0, 4, citylist)

if \_\_name\_\_ == "\_\_main\_\_": main()

## 4. Kết quả

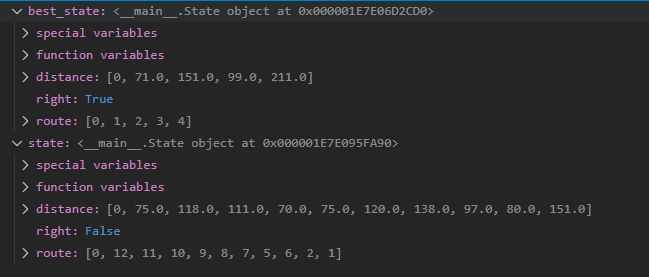


- sau khi khởi tạo các hàm, class và các giá trị đầu tiên “ Zerind” random đến các vị trí cạnh kề để chọn ra một đường đi ngắn nhất cho bài toán đến “Bucharest”



Sau khi chọn đường đi thứ nhất là True thì nó lưu lại đường đi và tiếp tục tìm kiếm các đường đi còn lại để so sánh .

Khi gặp mỗi đường đi ngắn hơn thì lưu lại và ngược lại bỏ qua khi giá trị lớn hơn.



Nó random cho tới khi hết đường đi và có giá trị tốt nhất , giá trị này không đổi trong quá trình random tiếp theo