HUTECH
Đại học Công nghệ Tp.HCM

BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ TP. HCM

ĐỒ ÁN MÔN HỌC

Trí Tuệ Nhân Tạo

<THUẬT TOÁN LEO ĐỒI >

Ngành: **CÔNG NGHỆ THÔNG TIN**

Giảng viên hướng dẫn : Phùng Thế Bảo

Sinh viên thực hiện :

1. Biện Huỳnh Công Khang (2011060425)
2. Hồ Huy Vũ (2011064073)
3. Nguyễn Tấn Trung (2011068762)
4. Nguyễn Quốc Huy (2011068325)
5. Võ Thành Lợi (2011065243)

Lớp: 20DTHB5 (Nhóm 5)

TP. Hồ Chí Minh, <2022>

Phụ Lục

Chương I. Tìm hiểu về Thuật toán Leo đồi

1. Mở đầu

1.1 Giới thiệu 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 đơn 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.

1.2 Nguồn gốc

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 cụ 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. Đặc điểm của Leo đồi:

Sau đây là một số tính năng chính của Hill Climbing Algorithm:

2.1 Biến thể Tạo và Thử nghiệm: Leo đồi là biến thể của Phương pháp Tạo và Thử nghiệm. Phương pháp Tạo và Kiểm tra tạo ra phản hồi giúp quyết định hướng di chuyển trong không gian tìm kiếm.

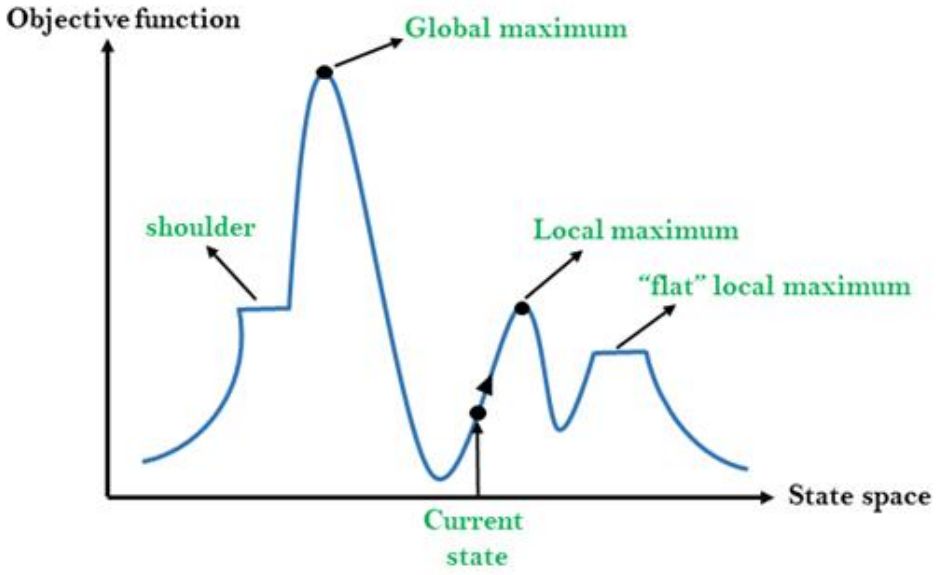
2.2 Cách tiếp cận tham lam: Tìm kiếm Hill Climbing Algorithm di chuyển theo hướng tối ưu hóa chi phí.

2.3 Không quay lui: Nó không quay lại không gian tìm kiếm, vì nó không nhớ các trạng thái trước đó.

3. Sơ đồ không gian trạng thái cho Leo đồi:

Cảnh quan không gian – trạng thái là một biểu diễn đồ họa của Hill Climbing Algorithm đang hiển thị một biểu đồ giữa các trạng thái khác nhau của thuật toán và Hàm mục tiêu / Chi phí.

Trên trục Y, chúng ta đã sử dụng hàm có thể là hàm mục tiêu hoặc hàm chi phí và không gian trạng thái trên trục x. Nếu hàm trên trục Y là chi phí thì mục tiêu của tìm kiếm là tìm giá trị tối thiểu toàn cục và giá trị tối thiểu cục bộ. Nếu hàm của trục Y là hàm Mục tiêu, thì mục tiêu của tìm kiếm là tìm giá trị cực đại toàn cục và cực đại cục bộ.



4. Các vùng khác nhau trong cảnh quan không gian trạng thái:

Local Maximum: Cực đại cục bộ là một trạng thái tốt hơn các trạng thái lân cận của nó, nhưng cũng có một trạng thái khác cao hơn nó.

Global Maximum: Tối đa toàn cầu là trạng thái tốt nhất có thể của cảnh quan không gian trạng thái. Nó có giá trị cao nhất của hàm mục tiêu.

Current state: Đó là một trạng thái trong sơ đồ cảnh quan mà một tác nhân hiện đang có mặt.

Flat local maximum: Là không gian phẳng trong cảnh quan mà tất cả các trạng thái lân cận của các trạng thái hiện tại có cùng giá trị.

Shoulder: Là một vùng cao nguyên có sườn dốc.

5. Các loại Thuật toán Leo đồi:

5.1 Leo đồi đơn giản (Simple hill Climbing):

5.1.1 Ý tưởng:

Leo đồi đơn giản là cách đơn giản nhất để thực hiện Hill Climbing Algorithm. Nó chỉ đánh giá trạng thái nút lân cận tại một thời điểm và chọn

trạng thái đầu tiên tối ưu hóa chi phí hiện tại và đặt nó làm trạng thái hiện tại. Nó chỉ kiểm tra đó là một trạng thái kế thừa và nếu nó thấy tốt hơn trạng thái hiện tại, thì chuyển sang trạng thái khác ở cùng trạng thái đó. Thuật toán này có các tính năng sau:

+ Ít tốn thời gian hơn

+ Giải pháp kém tối ưu hơn và giải pháp không được đảm bảo

5.1.2 Giải thuật:

Bước 1: Nếu trạng thái bắt đầu (T0) là trạng thái đích : thoát và báo là đã tìm được lời giải. Ngược lại, đặt trạng thái hiện hành (Ti) là trạng thái Khởi đầu (T0).

Bước 2: Lặp lại cho đến khi đạt đến trạng thái kết thúc hoặc cho đến khi không tồn tại một trạng thái tiếp theo hợp lệ (Tk) của trạng thái hiện hành:

2a. Đặt Tk là một trạng thái tiếp theo hợp lệ của trạng thái hiện hành Tk.

2b. Đánh giá trạng thái Tk mới:

b1. Nếu là trạng thái kết thúc thì trả về giá trị này và thoát.

b2. Nếu không phải là trạng thái kết thúc nhưng tốt hơn trạng thái hiện hành thì cập nhật nó thành trạng thái hiện hành.

b3. Nếu nó không tốt hơn trạng thái hiện hành thì tiếp tục vòng lặp.

5.2 Leo đồi dốc đứng (Steepest–Ascent hill Climbing):

5.2.1 Ý tưởng:

Giống như leo đồi đơn giản, chỉ khác ở điểm là leo đồi dốc đứng sẽ duyệt tất cả các hướng đi có thể và chọn đi theo trạng thái **tốt nhất** trong số các trạng thái kế tiếp có thể có (trong khi đó leo đồi đơn giản chỉ chọn đi theo trạng thái kế tiếp đầu tiên tốt hơn trạng thái hiện hành mà nó tìm thấy).

5.2.2 Giải thuật:

Bước 1: Nếu trạng thái bắt đầu cũng là trạng thái đích thì thoát và báo là đã tìm được lời giải. Ngược lại, đặt trạng thái hiện hành (Ti) là trạng thái khởi đầu (T0).

Bước 2: Lặp lại cho đến khi đạt đến trạng thái kết thúc hoặc cho đến khi (Ti) không tồn tại một trạng thái kế tiếp (Tk) nào tốt hơn trạng thái hiện tại (Ti).

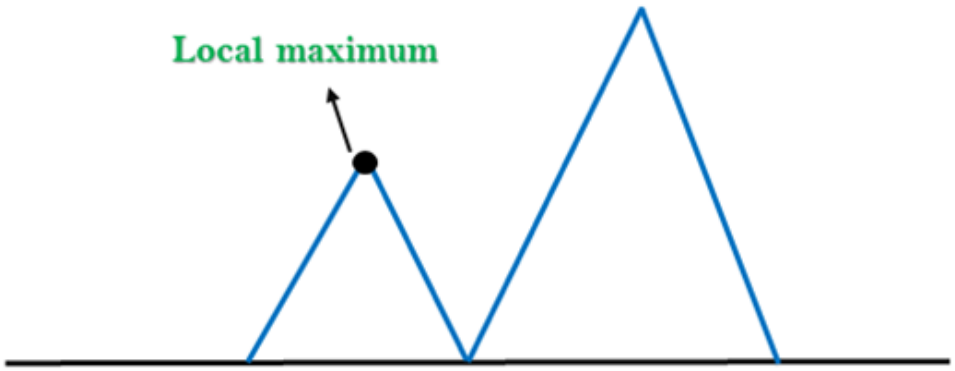
2a. Đặt S bằng tập tất cả trạng thái kế tiếp có thể có của Ti và tốt hơn Ti.

2b. Xác định Tkmax là trạng thái tốt nhất trong tập S. Đặt Ti = Tkmax.

6. Các vấn đề trong Thuật toán Leo đồi

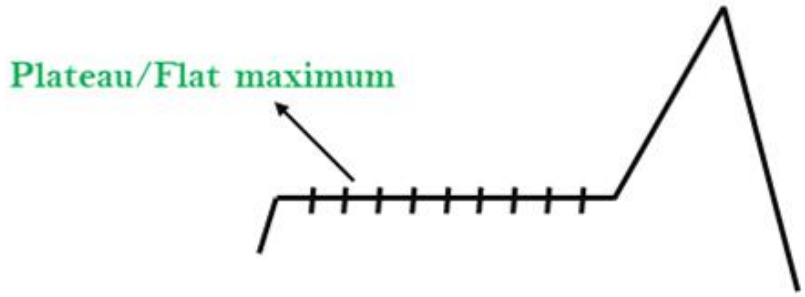
6.1 **Local Maximum**: Cực đại cục bộ là trạng thái cao nhất trong cảnh quan tốt hơn từng trạng thái lân cận của nó, nhưng có một trạng thái khác cũng hiện diện cao hơn trạng thái tối đa cục bộ.

Giải pháp: Kỹ thuật backtracking có thể là một giải pháp tối đa cục bộ trong cảnh quan không gian trạng thái. Tạo danh sách các đường dẫn đầy hứa hẹn để thuật toán có thể kiểm tra lại không gian tìm kiếm và khám phá các đường dẫn khác.



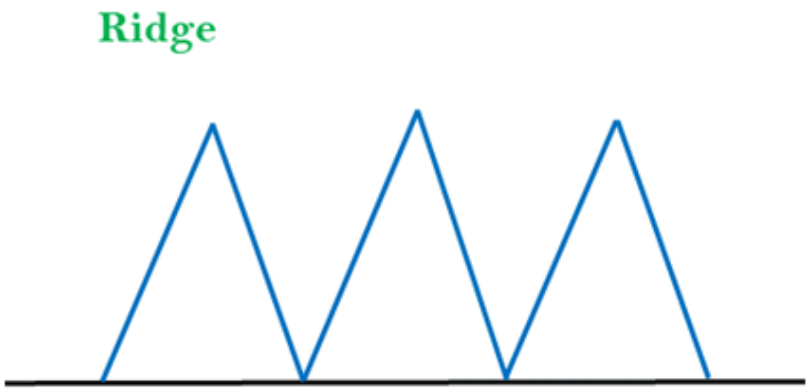
6.2 **Plateau**: Plateu là vùng phẳng của không gian tìm kiếm trong đó tất cả các trạng thái lân cận của trạng thái hiện tại đều chứa cùng một giá trị, vì thuật toán này không tìm ra hướng tốt nhất để di chuyển. Tìm kiếm leo đồi có thể bị thất lạc trong khu vực cao nguyên.

Giải pháp: Giải pháp cho bình nguyên là thực hiện các bước lớn hoặc rất ít bước trong khi tìm kiếm, để giải quyết vấn đề. Chọn ngẫu nhiên một trạng thái khác xa trạng thái hiện tại để thuật toán có thể tìm thấy vùng không bình nguyên.



6.3 **Ridges**: ridge là một dạng đặc biệt của **Local Maximum**. Nó có một khu vực cao hơn các khu vực xung quanh của nó, nhưng bản thân nó có một độ dốc và không thể đạt được trong một lần di chuyển.

Giải pháp: Với việc sử dụng tìm kiếm hai chiều hoặc bằng cách di chuyển theo các hướng khác nhau, chúng ta có thể cải thiện vấn đề này.



7. Mô phỏng Annealing:

Hill Climbing Algorithm không bao giờ di chuyển đến giá trị thấp hơn được đảm bảo là không hoàn chỉnh vì nó có thể bị mắc kẹt ở mức tối đa cục bộ. Và nếu thuật toán áp dụng một bước đi ngẫu nhiên, bằng cách di chuyển một người kế

nhiệm, thì nó có thể hoàn thành nhưng không hiệu quả. Ủ mô phỏng là một thuật toán mang lại cả hiệu quả và tính hoàn chỉnh.

Theo thuật ngữ cơ học Annealing là một quá trình làm cứng kim loại hoặc thủy tinh đến nhiệt độ cao sau đó làm nguội dần dần, do đó, điều này cho phép kim loại đạt đến trạng thái tinh thể năng lượng thấp. Quá trình tương tự được sử dụng trong ủ mô phỏng, trong đó thuật toán chọn một nước đi ngẫu nhiên, thay vì chọn nước đi tốt nhất. Nếu di chuyển ngẫu nhiên cải thiện trạng thái, thì nó đi theo con đường tương tự. Nếu không, thuật toán đi theo con đường có xác suất nhỏ hơn 1 hoặc nó di chuyển xuống dốc và chọn một con đường khác.

8. Tài liệu tham khảo:

https://thanhthao94blog.wordpress.com/2016/08/07/giai-thuat-leo-doi-hill-climbing/

https://morioh.com/p/7c35b905c9a4

https://websitehcm.com/hill-climbing-algorithm-trong-tri-tue-nhan-tao/

Giáo trình, Lý thuyết môn Trí Tuệ Nhân Tạo Đại Học Công Nghệ TP.HCM – HUTECH (20DTHB5) (2022).

Chương II. Demo Thuật toán Leo đồi

1. Triển khai Thuật toán Leo đồi với bài toán cụ thể

Bài toán lựa chọn: **Người bán hàng** (Travelling Salesman Problem)

1.1 Ý tưởng:

Trong bài toán Nhân viên bán hàng đi du lịch, chúng ta có một nhân viên bán hàng cần đến thăm một số thành phố chính xác một lần, sau đó anh ta quay trở lại thành phố đầu tiên. Khoảng cách giữa từng cặp thành phố đã được biết trước và chúng ta cần tìm con đường ngắn nhất. Như bạn có thể tưởng tượng, có (thường là) một số lượng lớn các giải pháp (lộ trình) khả thi cho một vấn đề nhân viên bán hàng Du lịch cụ thể; mục tiêu là tìm ra giải pháp tốt nhất (tức là ngắn nhất).

Thuật toán Leo đồi cố gắng tìm ra giải pháp tốt nhất cho vấn đề này bằng cách bắt đầu với một giải pháp ngẫu nhiên, và sau đó tạo ra các giải pháp lân cận: các giải pháp chỉ khác một chút so với giải pháp hiện tại. Nếu giải pháp tốt nhất trong số những người hàng xóm (neighbours) đó tốt hơn (tức là ngắn hơn) so với giải pháp hiện tại, nó sẽ thay thế giải pháp hiện tại bằng bằng giải pháp tốt hơn này. Sau đó, nó lặp lại mô hình bằng cách tạo ra các hàng xóm một lần nữa. Nếu tại một thời điểm nào đó không có hàng xóm nào tốt hơn giải pháp hiện tại, nó sẽ trả về giải pháp hiện tại. Thuật toán này khá đơn giản, nhưng cần phải nói rằng không phải lúc nào nó cũng tìm ra giải pháp tốt nhất. Nó có thể bị mắc kẹt ở mức tối đa cục bộ: nơi mà giải pháp hiện tại không phải là giải pháp tốt nhất cho vấn đề, nhưng không có người hàng xóm trực tiếp nào của giải pháp hiện tại tốt hơn giải pháp hiện tại. Như đã mô tả, thuật toán sẽ dừng lại ở điểm như vậy, rất tiếc là không trả về giải pháp tốt nhất. Các thuật toán phức tạp hơn tồn tại có cơ hội cao hơn để tìm ra giải pháp tốt nhất, nhưng chúng thường chiếm nhiều tài nguyên tính toán hơn.

1.2 Tạo vấn đề cho bài toán Người bán hàng:

Đầu tiên chúng ta sẽ tạo danh sách các thành phố, trong đó mỗi thành phố có thông tin về khoảng cách từ đó đến các thành phố khác. Tất nhiên, khoảng cách từ mỗi thành phố đến chính nó bằng 0, và khoảng cách từ thành phố A đến thành phố B cũng giống như khoảng cách từ thành phố B đến thành phố A. Điều đó cho chúng ta một danh sách có kích thước n (trong demo này thì n bằng 4):

```
tsp
= [
        [0, 400, 500, 300],
        [400, 0, 300, 500],
        [500, 300, 0, 400],
        [300, 500, 400, 0]
    ]
```

Như chúng ta có thể thấy, khoảng cách của thành phố đầu tiên với chính nó, tất nhiên là 0 và ví dụ khoảng cách của nó đến thành phố thứ ba là 500. Người ta có thể thấy rằng khoảng cách của thành phố thứ ba với thành phố đầu tiên cũng là 500.

1.3 Tạo một trình tạo giải pháp ngẫu nhiên:

Để hoạt động leo đồi hoạt động, nó phải bắt đầu với một giải pháp ngẫu nhiên cho bài toán người bán hàng. Từ đó, nó có thể tạo ra các giải pháp lân cận và bắt đầu quá trình tối ưu hóa. Vậy để giải quyết vấn đề này đơn giản là một danh sách các số nhận dạng của tất cả các thành phố, theo thứ tự người bán hàng nên đến các thành phố đó. Mỗi thành phố phải được đến đúng một lần. Hàm phạm vi của Python rất tốt để tạo thành phố, vì nó tạo ra một phạm vi gồm tất cả các số từ 0 đến đối số đã cho, trong trường hợp này là độ dài của chính bài toán, vì bản thân bài toán chứa một mục nhập cho mỗi thành phố.

Vì mỗi thành phố chỉ được ghé một lần, sau khi số nhận của nó được thêm vào giải pháp, thì chúng ta sẽ xóa số nhận dạng của thành phố đó khỏi danh sách số nhận dạng thành phố. Lưu ý bạn sẽ cần nhập ngẫu nhiên để code này hoạt động:

```
def randomSolution(tsp):
    cities = list(range(len(tsp)))
    solution = []


    for i in range(len(tsp)):
        randomCity = cities[random.randint(0, len(cities) - 1)]
        solution.append(randomCity)
        cities.remove(randomCity)


    return solution
```

1.4 Tạo một hàm tính toán độ dài của một tuyến đường:

Vì chúng ta muốn Thuật toán tìm ra lời giải ngắn nhất, nên phải cần một hàm tính độ dài của một giải pháp cụ thể. Vì một giải pháp là danh sách tất cả các thành phố theo một thứ tự cụ thể, chúng ta chỉ có thể lặp lại một giải pháp và sử dụng đối số tsp để thêm khoảng cách đến từng thành phố mới vào tổng chiều dài tuyến đường của chúng ta. Trình lặp i "visit" từng thành phố, vì vậy i – 1 là "at" thành phố trước đó hoặc thành phố cuối cùng khi i bằng 0 (đó chính xác là những gì chúng ta muốn, vì chúng ta muốn kết thúc ở thành phố đầu tiên một lần nữa). Do

đó giải pháp [i] cho thành phố hiện tại và giải pháp [i-1] cho thành phố trước đó. Sau đó ta cần sử dụng tsp để lấy khoảng cách giữa các thành phố này, chúng ta thêm vào tổng chiều dài của tuyến đường (routeLength).

```
def routeLength(tsp, solution):
    routeLength = 0
    for i in range(len(solution)):
        routeLength += tsp[solution[i - 1]][solution[i]]
    return routeLength
```

1.5 Tạo một hàm tạo ra tất cả hàng xóm (neighbours) của một giải pháp

Như đã giải thích trước đây, việc leo đồi hoạt động bằng cách tạo ra tất cả các giải pháp lân cận với giải pháp hiện tại. Hãy tạo ra một chức năng làm chính xác điều đó. Một giải pháp lân cận là một giải pháp chỉ khác một chút so với giải pháp hiện tại. Cũng lưu ý rằng một hàng xóm vẫn cần phải là một giải pháp chính xác: mỗi thành phố vẫn cần được đến thăm chính xác một lần. Chúng ta có thể thực hiện cả hai điều này bằng cách tạo một hàng xóm như sau: sao chép giải pháp hiện tại, sau đó tạo hai thành phố hoán đổi vị trí cho nhau! Bằng cách này, chúng ta tạo tất các hàng xóm cho một giải pháp và cận tạo các thành phố hoán đổi vị trí, chúng ta cần tạo hai vòng lặp for, một vòng lặp lồng vào nhau, cả hai đều lặp lại giải pháp hiện tại. Vì hoán đổi thành phố A với thành phố B cũng giống như hoán đổi thành phố B với thành phố A, nên vòng lặp thứ hai của chúng ta chỉ cần lặp qua các thành phố đó mà vòng lặp đầu tiên chưa lặp lại. Bên trong vòng lặp thứ hai, chúng ta tạo hàng xóm của mình với các thành phố được hoán đổi và thêm nó vào danhh sách hàng xóm của chúng ta.

```
def
getNeighbours(solution):
                                neighbours = []
                                for i in range(len(solution)):
                                    for j in range(i + 1, len(solution)):
                                        neighbour = solution.copy()
                                        neighbour[i] = solution[j]
                                        neighbour[j] = solution[i]
```

```
                neighbours.append(neighbour)
        return neighbours
```

1.6 Tạo một hàm tìm hàng xóm tốt nhất

Chúng ta đã có một chức năng tạo ra tất cả hàng xóm cho một giải pháp, tiếp theo chúng ta tạo ra một chức năng tìm ra giải pháp tốt nhất trong số các hàng xóm này. Đây là một hàm khá đơn giản: trước tiên nó đặt bestNeighbour thành hàng xóm đầu tiên trong danh sách các hàng xóm (và bestRouteLength theo độ dài tuyến đường của nó), và sau đó lặp lại trên tất cả các hàng xóm; khi một hàng xóm có độ dài tuyến đường ngắn hơn, cả bestNeighbour và bestRouteLength đều được cập nhật. Bằng cách này, hàng xóm tốt nhất được tìm thấy:

```
def
getBestNeighbour(tsp,
neighbours):
                bestRouteLength = routeLength(tsp, neighbours[0])
                bestNeighbour = neighbours[0]
                for neighbour in neighbours:
                    currentRouteLength = routeLength(tsp, neighbour)
                    if currentRouteLength < bestRouteLength:
                        bestRouteLength = currentRouteLength
                        bestNeighbour = neighbour
                return bestNeighbour, bestRouteLength
```

1.7 Tạo Thuật toán Leo đồi:

Đây là chính năng cốt lõi! Sau khi tạo các chức năng trước đó, bước này trở nên khá dễ dàng:

```
def
hillClimbing(tsp):
                currentSolution = randomSolution(tsp)
                currentRouteLength = routeLength(tsp, currentSolution)
                neighbours = getNeighbours(currentSolution)
                bestNeighbour, bestNeighbourRouteLength =
            getBestNeighbour(tsp, neighbours)
```

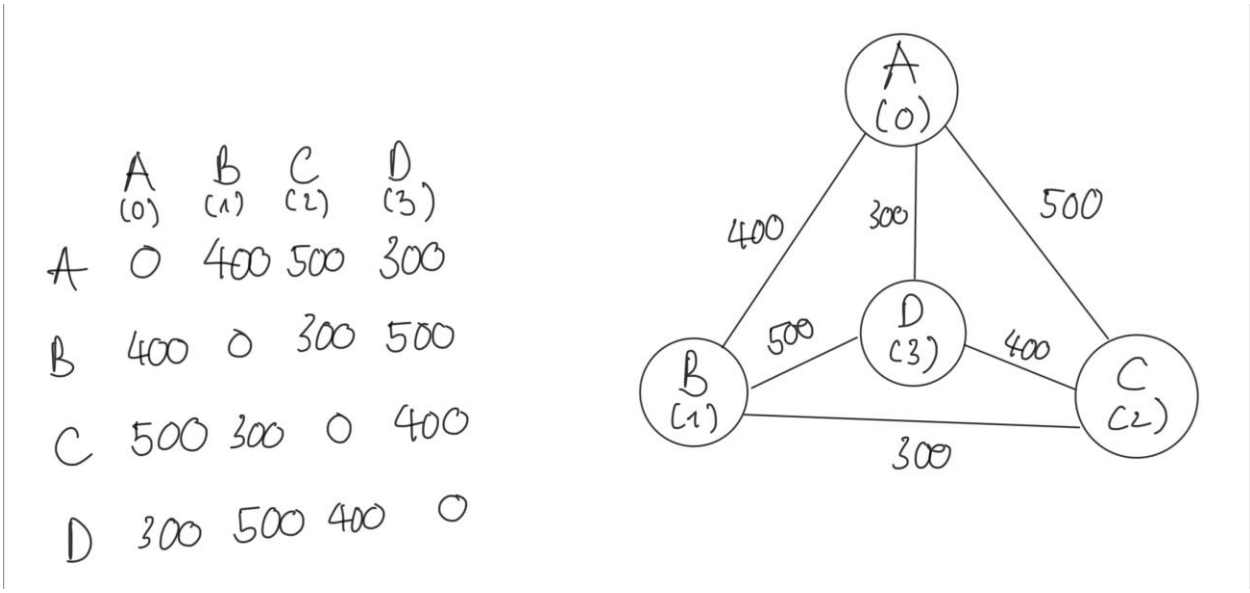
```
        while bestNeighbourRouteLength < currentRouteLength:
            currentSolution = bestNeighbour
            currentRouteLength = bestNeighbourRouteLength
            neighbours = getNeighbours(currentSolution)
            bestNeighbour, bestNeighbourRouteLength =
    getBestNeighbour(tsp, neighbours)


        return currentSolution, currentRouteLength
```

Đầu tiên, chúng ta thực hiện một giải pháp ngẫu nhiên và tính toán độ dài tuyến đường của nó. Sau đó, chúng ta tạo ra các giải pháp lân cận và tìm ra giải pháp tốt nhất. Từ đó trở đi, miễn là hàng xóm tốt nhất tốt hơn giải pháp hiện tại, chúng ta lặp lại mô hình tương tự với giải pháp hiện tại mỗi khi được cập nhật với hàng xóm tốt nhất. Khi quá trình này dừng lại, chúng ta trả về giải pháp hiện tại (và độ dài tuyến đường của nó).

2. Chương trình Demo chạy bằng Python:

Vd cụ thể 1:



Kết quả:

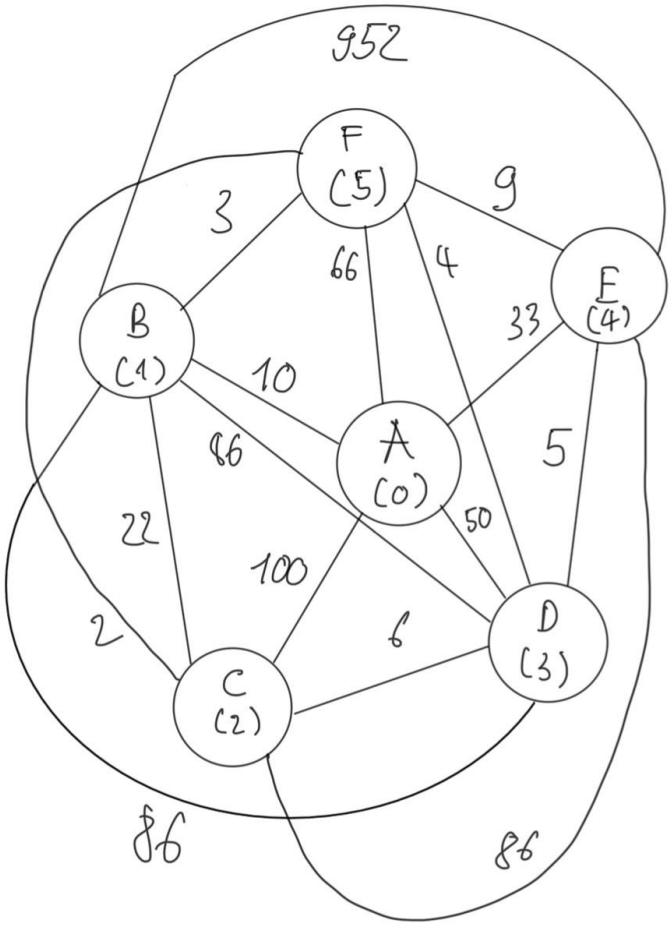
```
D:\Máy cũ ổ 1\Tài liệu ĐH\Trí Tuệ Nhân Tạo>python demoHillClimbingTravellingSalesman.py
([3, 0, 1, 2], 1400)

D:\Máy cũ ổ 1\Tài liệu ĐH\Trí Tuệ Nhân Tạo>python demoHillClimbingTravellingSalesman.py
([2, 3, 0, 1], 1400)

D:\Máy cũ ổ 1\Tài liệu ĐH\Trí Tuệ Nhân Tạo>python demoHillClimbingTravellingSalesman.py
([2, 1, 0, 3], 1400)
```

Vd cụ thể 2:

	A (0)	B (1)	C (2)	D (3)	E (4)	F (5)
A	0	10	100	50	33	66
B	10	0	22	86	952	3
C	100	22	0	6	86	2
D	50	86	6	0	5	4
E	33	952	86	5	0	9
F	66	3	2	4	9	0



Kết quả:

```
D:\Máy cũ ổ 1\Tài liệu ĐH\Trí Tuệ Nhân Tạo>python demoHillClimbingTravellingSalesman.py
([1, 0, 4, 3, 2, 5], 59)

D:\Máy cũ ổ 1\Tài liệu ĐH\Trí Tuệ Nhân Tạo>python demoHillClimbingTravellingSalesman.py
([0, 4, 3, 2, 5, 1], 59)

D:\Máy cũ ổ 1\Tài liệu ĐH\Trí Tuệ Nhân Tạo>python demoHillClimbingTravellingSalesman.py
([3, 4, 0, 1, 5, 2], 59)
```

demoHillClimbingTravellingSalesman.py