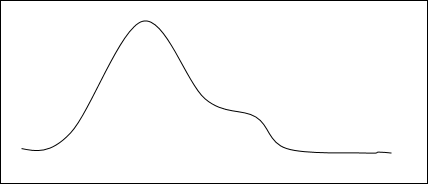
**Hill Climbing Algorithm**

Hill Climbing is a technique to solve certain optimization problems. In this technique, we start with a sub-optimal solution and the solution is improved repeatedly until some condition is maximized.



The idea of starting with a sub-optimal solution is compared to starting from the base of the hill, improving the solution is compared to walking up the hill, and finally maximizing some condition is compared to reaching the top of the hill.

Hence, the hill climbing technique can be considered as the following phases:

* Constructing a sub-optimal solution obeying the constraints of the problem
* Improving the solution step-by-step
* Improving the solution until no more improvement is possible

Hill Climbing technique is mainly used for solving computationally hard problems. It looks only at the current state and immediate future state. Hence, this technique is memory efficient as it does not maintain a search tree.

### Iterative Improvement

In iterative improvement method, the optimal solution is achieved by making progress towards an optimal solution in every iteration. However, this technique may encounter local maxima. In this situation, there is no nearby state for a better solution.

This problem can be avoided by different methods. One of these methods is simulated annealing.

### Random Restart

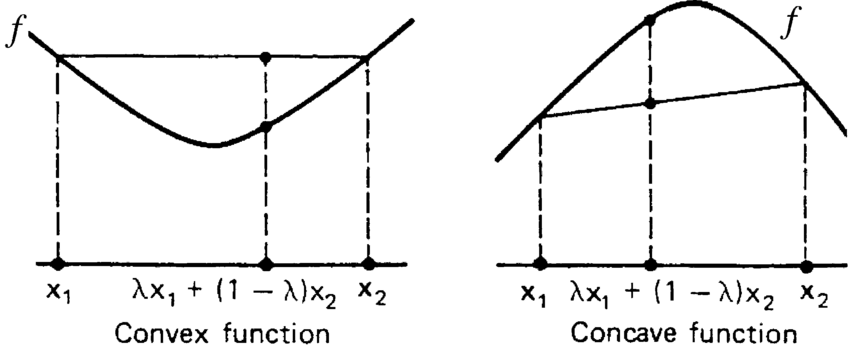
This is another method of solving the problem of local optima. This technique conducts a series of searches. Every time, it starts from a randomly generated initial state. Hence, optima or nearly optimal solution can be obtained comparing the solutions of searches performed.

## Problems of Hill Climbing Technique

### Local Maxima

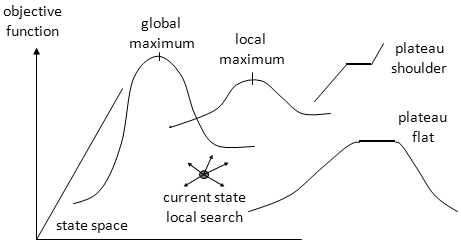
If the heuristic is not convex, Hill Climbing may converge to local maxima, instead of global maxima.

### C:\Users\Kostas\Downloads\concave-function.png



### Plateau

A plateau is encountered when the search space is flat or sufficiently flat that the value returned by the target function is indistinguishable from the value returned for nearby regions, due to the precision used by the machine to represent its value.



## Complexity of Hill Climbing Technique

This technique does not suffer from space related issues, as it looks only at the current state. Previously explored paths are not stored.

For most of the problems in Random-restart Hill Climbing technique, an optimal solution can be achieved in polynomial time. However, for NP-Complete problems, computational time can be exponential based on the number of local maxima.

## Applications of Hill Climbing Technique

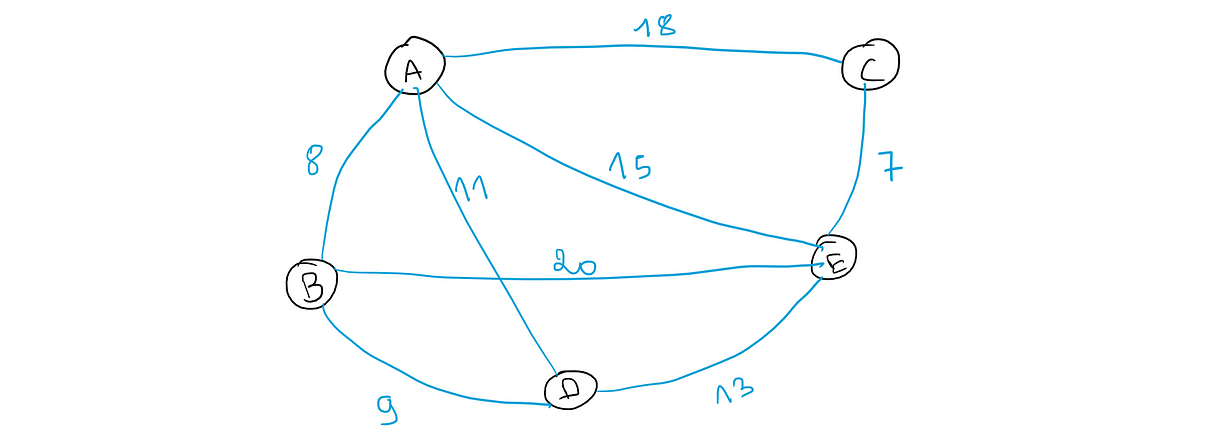
Hill Climbing technique can be used to solve many problems, where the current state allows for an accurate evaluation function, such as Network-Flow, Travelling Salesman problem, 8-Queens problem, Integrated Circuit design, etc.

## Example - travelling salesman problem

This technique can be applied to solve the travelling salesman problem. First an initial solution is determined that visits all the cities exactly once. Hence, this initial solution is not optimal in most of the cases. Even this solution can be very poor. The Hill Climbing algorithm starts with such an initial solution and makes improvements to it in an iterative way. Eventually, a much shorter route is likely to be obtained.

The TSP stands as one of the best known problems when it comes to work with NP-hard problems, which implies that no known algorithm exists to solve it in polynomial time. The problem can be summarized as follows : "Given a set of cities and the cost of travel (or distance) between each possible pairs, the TSP, is to find the best possible way of visiting all the cities and returning to the starting point that minimize the travel cost (or travel distance)." The exact solution to this problem with n cities can only be determined through evaluating (n-1)!/2 possibilities.

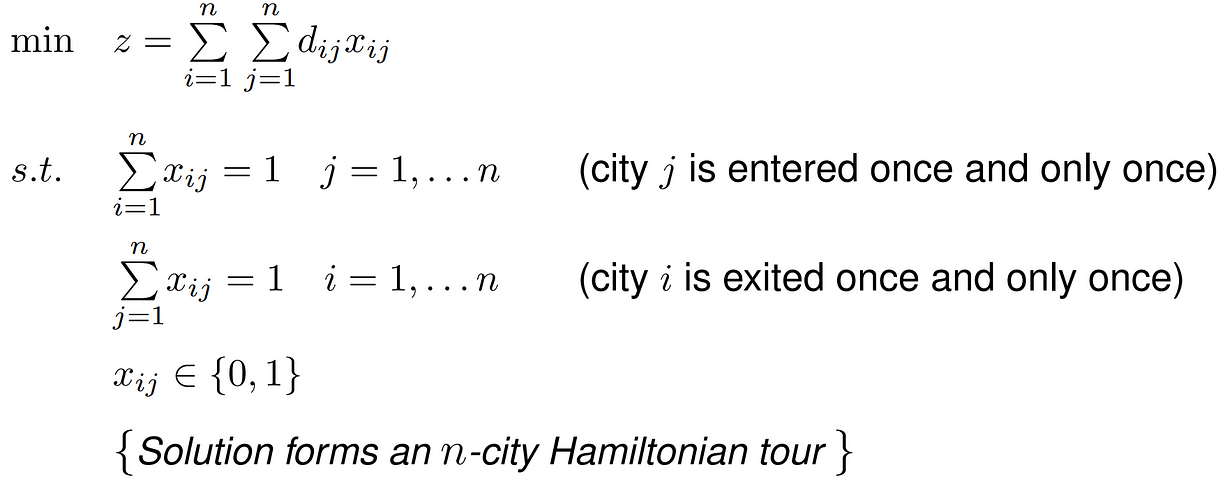
For our exploration, I’ll focus on the symmetric variant of the TSP (**sTSP**), where the cost from *city j* to *city i*equals the cost from *city i*to *city j.*



Example of a Symmetric TSP with 5 nodes (A, B, C, D, E)

**Mathematical formulation of sTSP**

Let *V = {v₁, …, vₙ}* be a set of cities, *A = {(r, s) ; r,s ∈ V}*be the set of edges, *xᵣₛ = 1* if a connection between *r* and *s* exists, otherwise it's *0*, and *dᵣₛ* *=*dₛᵣ be a cost measure associated with *(r, s)∈A.*The sTSP is about finding a Hamiltonian cycle (a cycle visiting every node exactly once) such that its c-length (the sum of the lengths of its edges) is as small as possible.

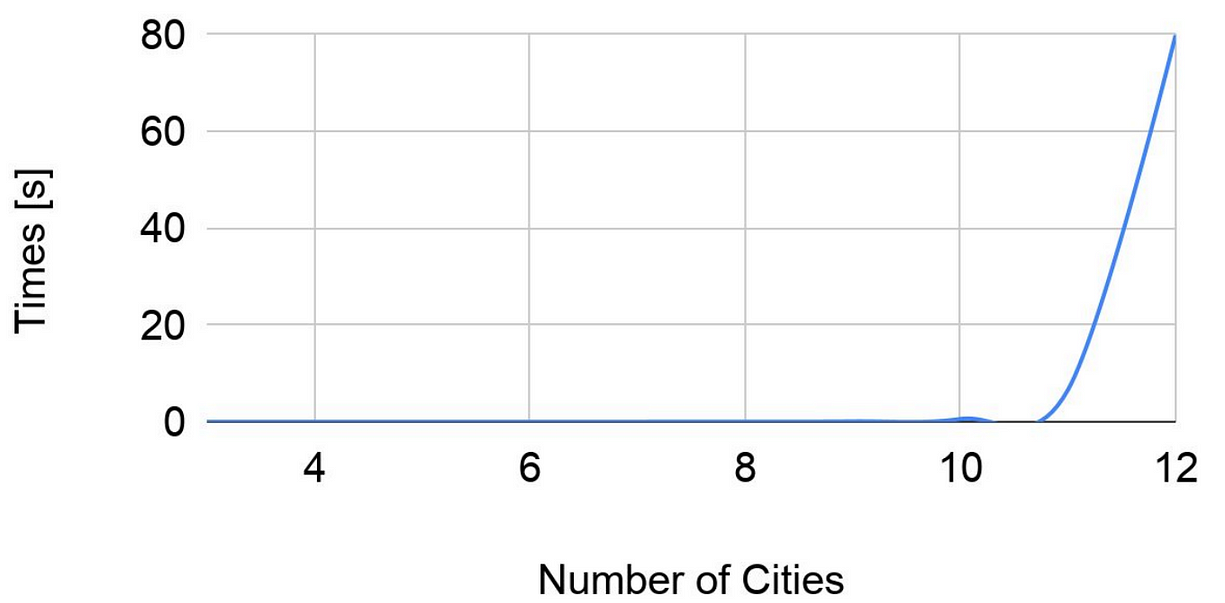


**Other variants of the TSP:**  
- Asymmetric TSP — This variant permits situations where*dᵣₛ****≠****dₛᵣ*for some city pairs*.*  
- Multi TSP— Instead of a single salesman, we have multiple salesmen starting from a common point, collectively visiting all cities.  
- Stochastic TSP — In this variant, the cost measures associated with edges are non-deterministic.

**Some applications of the TSP:**  
The TSP finds practical applications in various domains, including delivery services, vehicle routing problems, logistics, and planning and scheduling.

**Why sacrificing performance to reduce time**An intriguing question arises: Why not exhaustively explore all possibilities and opt for the exact solution, given the advancements in computing performance?

Unfortunately we are still far from achieving that. The following figure illustrates the exponential increase in computation time as the number of cities in the TSP instance grows



Variation of time with the variation of number of cities

**Hill Climbing Algorithm**

In its simplest form, the Hill Climbing algorithm functions as follows: Starting from an initial solution, it explores its immediate neighborhood, making small modifications or steps to the current solution. If a neighboring solution is better (according to the problem’s objective function), the algorithm moves to that solution. This process repeats until a local maximum is reached, meaning that no further improvements can be made by moving to adjacent solutions.

import random

# Distance matrix representing distances between cities

# Replace this with the actual distance matrix for your problem

distance\_matrix = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

def total\_distance(path):

# Calculate the total distance traveled in the given path

total = 0

for i in range(len(path) - 1):

total += distance\_matrix[path[i]][path[i+1]]

total += distance\_matrix[path[-1]][path[0]] # Return to starting city

return total

def hill\_climbing\_tsp(num\_cities, max\_iterations=10000):

current\_path = list(range(num\_cities)) # Initial solution, visiting cities in order

current\_distance = total\_distance(current\_path)

for \_ in range(max\_iterations):

# Generate a neighboring solution by swapping two random cities

neighbor\_path = current\_path.copy()

i, j = random.sample(range(num\_cities), 2)

neighbor\_path[i], neighbor\_path[j] = neighbor\_path[j], neighbor\_path[i]

neighbor\_distance = total\_distance(neighbor\_path)

# If the neighbor solution is better, move to it

if neighbor\_distance < current\_distance:

current\_path = neighbor\_path

current\_distance = neighbor\_distance

return current\_path

def main():

num\_cities = 4 # Number of cities in the TSP

solution = hill\_climbing\_tsp(num\_cities)

print("Optimal path:", solution)

print("Total distance:", total\_distance(solution))

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Travelling salesman problem with Hill-Climbing

The Traveling Salesman Problem (TSP) is given by the following question: “Given is a list of cities and distances between each pair of cities - what is the shortest route that visits each city and returns to the original city?”

The TSP is an **NP-Hard-Problem** which does not mean an instance of the problem will be hard to solve. It means, there does not exist an algorithm that produces the best solution in polynomial time. We can not make predictions about how long it might take to find the best solution.

But, we can find a good solution which might not be the best solution. It is ok to find a route amongst 1000 cities that is only few miles longer than the best route. Particularly, if it would take an inordinate amount amount of computing time to get from our good solution to the best solution.

## https://www.uni-weimar.de/fileadmin/user/fak/medien/professuren/Intelligente_Softwaresysteme/Downloads/Lehre/SBSE/LabClass/SS19/Graph_TSP.pngRepresentation of the Problem

A TSP can be modelled as an undirected weighted graph:

- cities = vertices

- paths between cities = edges

- distance of a path = weight of an edge

This graph can be represented as an **Adjacency matrix**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| \ | A | B | C | D |
| A | 0 | 20 | 42 | 35 |
| B | 20 | 0 | 30 | 34 |
| C | 42 | 30 | 0 | 12 |
| D | 35 | 34 | 12 | 0 |

But how do we get the distances between cities if we only got the koordinates for each city?

Each city is represented by a cartesian koordinate P

P=(px,py)

### Euclidean Distance

Euclidean distance between two points P1 = (x1, y1) and P2 = (x2, y2) is:

d(P1,P2) = √ (x1−x2)2+(y1−y2)2

**EXAMPLE**

from itertools import permutations

def distance(p1, p2):

"""

Returns the Euclidean distance of two points in the Cartesian Plane.

>>> distance([3,4],[0,0])

5.0

"""

return ((p1[0] - p2[0])\*\*2 + (p1[1] - p2[1])\*\*2) \*\* 0.5

print(distance([3,6],[7,6]))

def total\_distance(points):

"""

Returns the length of the path passing throught

all the points in the given order.

>>> total\_distance([[1,2],[4,6]])

5.0

>>> total\_distance([[3,6],[7,6],[12,6]])

9.0

"""

return sum([distance(point, points[index + 1]) for index, point in enumerate(points[:-1])])

def traveling\_salesman(points, start = None):

"""

Finds the shortest route to visit all the cities by bruteforce.

Time complexity is O(N!), so never use on long lists.

>>> travelling\_salesman([[0,0],[10,0],[6,0]])

([0, 0], [6, 0], [10, 0])

>>> travelling\_salesman([[0,0],[6,0],[2,3],[3,7],[0.5,9],[3,5],[9,1]])

([0, 0], [6, 0], [9, 1], [2, 3], [3, 5], [3, 7], [0.5, 9])

"""

if start is None:

start = points[0]

return min([perm for perm in permutations(points) if perm[0] == start], key = total\_distance)

# Example for permutations

test\_list = ["a", "b", "c", "d"]

test\_permutations = permutations(test\_list)

for test\_perm in test\_permutations:

print(test\_perm)

import datetime

def main():

points = [

[0, 0],

[1, 5.7],

[2, 3],

[3, 7],

[0.5, 9],

[3, 5],

[9, 1],

[10, 5],

[20, 5],

[12, 12],

[20, 19],

[25, 6],

[23, 7]

]

points = points[:-5]

#points = points[:-4]

#points = points[:-3]

#points = points[:-2]

then = datetime.datetime.now()

result = traveling\_salesman(points)

distance\_result = total\_distance(result)

now = datetime.datetime.now()

print("calculation time", now - then)

print("""

The minimum distance to visit all

of the following points:\n

{0}

starting at

{1} is {2} and takes this

route:

{3}""".format(

points,

points[0],

distance\_result,

result))

if \_\_name\_\_ == "\_\_main\_\_":

main()

def cartesian\_matrix(coordinates):

'''

Creates a distance matrix for the city coords using straight line distances

computed by the Euclidean distance of two points in the Cartesian Plane.

'''

matrix = {}

for i, p1 in enumerate(coordinates):

for j, p2 in enumerate(coordinates):

matrix[i,j] = distance(p1,p2)

return matrix

m = cartesian\_matrix([(0,0), (1,0), (1,1)])

for k, v in m.items():

print(k, v)

print()

print(m[2,0])

def read\_coords(file\_handle):

coords = []

for line in file\_handle:

x,y = line.strip().split(',')

coords.append((float(x), float(y)))

return coords

with open('/content/drive/MyDrive/class\_material/city100.txt', 'r') as coord\_file:

coords = read\_coords(coord\_file)

matrix = cartesian\_matrix(coords)

def tour\_length(matrix, tour):

"""Sum up the total length of the tour based on the distance matrix"""

result = 0

num\_cities = len(list(tour))

for i in range(num\_cities):

j = (i+1) % num\_cities

city\_i = tour[i]

city\_j = tour[j]

result += matrix[city\_i, city\_j]

return result

import random

def all\_pairs(size, shuffle = random.shuffle):

r1 = list(range(size))

r2 = list(range(size))

if shuffle:

shuffle(r1)

shuffle(r2)

for i in r1:

for j in r2:

yield(i,j) # yield is an iterator function

# for each call of the generator it returns the next value in yield

from copy import deepcopy

# Tweak 1

def swapped\_cities(tour):

"""

Generator to create all possible variations where two

cities have been swapped

"""

ap = all\_pairs(len(tour))

for i,j in ap:

if i < j:

copy = deepcopy(tour)

copy[i], copy[j] = tour[j], tour[i]

yield copy

# Tweak 2

def reversed\_sections(tour):

"""

Generator to return all possible variations where the

section between two cities are swapped.

It preserves entire sections of a route,

yet still affects the ordering of multiple cities in one go.

"""

ap = all\_pairs(len(tour))

for i,j in ap:

if i != j:

#print("indices from:",i, "to", j)

copy = deepcopy(tour)

if i < j:

copy[i:j+1] = reversed(tour[i:j+1])

else:

copy[i+1:] = reversed(tour[:j])

copy[:j] = reversed(tour[i+1:])

if copy != tour: # not returning same tour

yield copy

# usage

print("start tour swap:",[1,2,3,4])

for tour in swapped\_cities([1,2,3,4]):

print(tour)

print()

print("start tour reverse section:",[1,2,3,4])

for tour in reversed\_sections([1,2,3,4]):

print(tour)

def init\_random\_tour(tour\_length):

tour = list(range(tour\_length))

random.shuffle(list(tour))

return tour

init\_function = lambda: init\_random\_tour(len(coords))

objective\_function = lambda tour: tour\_length(matrix, tour)

def hc(init\_function, move\_operator, objective\_function, max\_evaluations):

'''

Hillclimb until either max\_evaluations is

reached or we are at a local optima.

'''

best = init\_function()

best\_score = objective\_function(best)

num\_evaluations = 1

while num\_evaluations < max\_evaluations:

# move around the current position

move\_made = False

for next in move\_operator(best):

if num\_evaluations >= max\_evaluations:

break

next\_score = objective\_function(next)

num\_evaluations += 1

if next\_score < best\_score:

best = next

best\_score = next\_score

move\_made = True

break # depth first search

if not move\_made:

break # couldn't find better move - must be a local max

return (num\_evaluations, best\_score, best)

from PIL import Image, ImageDraw, ImageFont

def write\_tour\_to\_img(coords, tour, title, img\_file):

padding = 20

# shift all coords in a bit

coords = [(x+padding,y+padding) for (x,y) in coords]

maxx, maxy = 0,0

for x,y in coords:

maxx = max(x,maxx)

maxy = max(y,maxy)

maxx += padding

maxy += padding

img = Image.new("RGB",(int(maxx), int(maxy)), color=(255,255,255))

font=ImageFont.load\_default()

d=ImageDraw.Draw(img);

num\_cities = len(tour)

for i in range(num\_cities):

j = (i+1) % num\_cities

city\_i = tour[i]

city\_j = tour[j]

x1,y1 = coords[city\_i]

x2,y2 = coords[city\_j]

d.line((int(x1), int(y1), int(x2), int(y2)), fill=(0,0,0))

d.text((int(x1)+7, int(y1)-5), str(i), font=font, fill=(32,32,32))

for x,y in coords:

x,y = int(x), int(y)

d.ellipse((x-5, y-5, x+5, y+5), outline=(0,0,0), fill=(196,196,196))

d.text((1,1), title, font=font, fill=(0,0,0))

del d

img.save(img\_file, "PNG")

def reload\_image\_for\_jupyter(filename):

# pick a random integer with 1 in 2 billion chance of getting the same

# integer twice

import random

\_\_counter\_\_ = random.randint(0,2e9)

# now use IPython's rich display to display the html image with the

# new argument

from IPython.display import HTML, display

display(HTML('<img src="./'+filename+'?%d" alt="Schema of adaptive filter" height="100">' % \_\_counter\_\_))

def do\_hc\_evaluations(evaluations , move\_operator = swapped\_cities):

max\_evaluations = evaluations

then = datetime.datetime.now()

num\_evaluations, best\_score, best = hc(init\_function, move\_operator, objective\_function, max\_evaluations)

now = datetime.datetime.now()

print("computation time ", now - then)

print("best score:", best\_score)

print("best route:", best)

filename = "test"+str(max\_evaluations)+".PNG"

write\_tour\_to\_img(coords, best, filename, open(filename, "ab"))

reload\_image\_for\_jupyter(filename)

move\_operator = swapped\_cities

#move\_operator = reversed\_sections

max\_evaluations = 500

do\_hc\_evaluations(max\_evaluations,move\_operator)

move\_operator = swapped\_cities

#move\_operator = reversed\_sections

max\_evaluations = 5000

do\_hc\_evaluations(max\_evaluations,move\_operator)

#move\_operator = swapped\_cities

move\_operator = reversed\_sections

max\_evaluations = 50000

do\_hc\_evaluations(max\_evaluations,move\_operator)