**INDOOR NAVIGATION SYSTEM FOR HOSPITAL**

A PROJECT REPORT

**DATA STRUCTURES AND ALGORITHMS**

**(CSE2003)**

**SUBMITTED BY: SUBMITTED TO:**

**PRANJAL KARN (18BCE0134) PROF. PADMA PRIYA**

**RITIK GUPTA (18BCE0154)**

**AYUSH SHARMA (18BCE0172)**

**UMANG SINGH (18BCE2202)**



**PROBLEM STATEMENT**

In large size hospitals when a patient arrives alone, they often end up feeling lost. As a result the patient ask nurse and doctors in their way, for directions. This disturbs the doctor’s in their work. Also a large hospital contain various places like Operation Theater, dispensary, washrooms etc. People find it difficult to search all these places.

**ABSTRACT**

For the above stated problem we have created a system to guide patients in large hospitals through indoor navigation. The first approach is to go to the reception bay, from where the patients will be directed to go to the billing counter, available consultant doctor is then decided.

BASIC FRAMEWORK AND MODEL OF OUR PROJECT

Our project is about indoor navigation inside hospital in a new and easy method.

The basic idea and framework of our project is in the following manner:-

* The hospital administration will be having an android, an ios or web app through which the patients and employees can navigate in big puzzled hospital buildings.
* On the entrance door of hospital there will be a QR code which will pin the location of reception (with respect to BLE beacons installed in hospital building at various positions) in the navigation system through which the patient can be directed towards receptionist.
* After that patient will receive a prescription paper from receptionist which will contain QR code which will give location of prescribed doctor’s cabin to the patient on reading it through navigation system’s camera.
* The location of each vertices and various other positions will be considered with respect to BLE beacon network (Since GPS doesn’t work inside the building.)
* After reading the QR code the system will pin the specified location on map. And then the person can easily be given turn to turn direction to reach his desired position.

**INTRODUCTION**

In our indoor navigation system we will be using various technologies to bind up our project.

These technologies include BLE beacon, reading QR code through web cam.

The internal working of our project is as follows:-

* The whole indoor map of hospital can be devided into cluster of points at specific positions such as turning points (vertices), staircase, doctor’s cabin, various other rooms, water coolers, washrooms etc. All these points will be marked on maps with respect to planted BLE beacons network.
* When a patient will scan QR code, the app will create a tree with nodes.
* After that through dijkstra’s algorithm it will find shortest path and will direct people from specific point to point in the given shortest path.

BRIEF INTRODUCTION TO THE TECHNOLOGY USED IN OUR PROJECT:-

**BEACONS**

# Introduction

# What is a beacon?

## A beacon is a small **Bluetooth radio transmitter**. It’s kind of like a lighthouse: it repeatedly transmits a single signal that other devices can see. Instead of emitting visible light, though, it broadcasts a radio signal that is made up of a combination of letters and numbers transmitted on a regular interval of approximately 1/10th of a second. A Bluetooth-equipped device like a smartphone can “see” a beacon once it’s in range, much like sailors looking for a lighthouse to know where they are.

## What is a beacon like on the inside?

What do they look like? Beacons are very small, simple devices. If you crack one open, you won’t find thirty motherboards and oodles of wires. You’ll find a CPU, radio, and batteries. Beacons often use small lithium chip batteries (smaller and more powerful than AA batteries) or run via connected power like USB plugs. They come in different shapes and colors, may include accelerometers, temperature sensors, or unique add-ons but all of them have one thing in common—they transmit a signal.

### **What is a beacon actually transmitting?**

It’s not throwing just any old message into the air. It’s transmitting a unique ID number that tells a listening device which beacon it’s next to.  
Really, it’s just a code name.

How can I interact with beacons?

For example, when a shopping mall installs beacons in their shop, all of the beacons will have certain IDs, registered in their dedicated app. This means a smartphone app can immediately recognize that the incoming ID is important and that it’s from that particular mall. The ID, however, has little meaning on its own; it’s entirely up to an app or other program to recognize what it means.

What happens next? That depends on what the owner has programmed it to do. One code could trigger the app to send a coupon. Another could offer navigation services. The possibilities are nearly endless. All the beacon has to do is connect your exact location to the app, and the rest is up to the program.

## What’s happening behind the computer screen?

Beacons are incredibly misunderstood. They are not tracking you. They’re not interested in that.

They’re just broadcasting a signal. Here’s why this signal can trigger so many different things.  
An online platform (for example, the Kontakt.io dashboard), lets you manage, configure, and update all your beacons. From there, you may develop your own app or use a further program called a Content Management System. These programs allow you to associate links, images, videos, and texts with individual beacons. Many of these platforms are made to be highly user-friendly. **This means they are often sleek and easy-to-use with no coding required.** For example, a program could let a museum owner add brand new capabilities to their gallery app (like quizzes or audio guides) just by typing questions or text. The program then does all the hard work automatically and stores everything in the cloud so your app can easily access it.

## How do beacons connect to the web?

You’ve probably heard of **Bluetooth**. It’s present in 90% of all phones and has been around since the 1990s. So what’s changed? Why is it so important now? While many consumers don’t use Bluetooth on a daily basis, it’s hugely important to the Internet of Things. Being in 90% of the world’s phones, Bluetooth technology means beacons are compatible with devices consumers use on a daily basis around the globe.

Bluetooth provides the **infrastructure** for the entire beacon ecosystem. It’s a standard for sending data over short distances, a wireless technology not so dissimilar from WiFi. This is why beacon hardware can be simple. There is already a web of Bluetooth around you that can connect beacons and smart devices and almost anything else.

**Why do we say “BLE beacons”?**

**BLE** stands for Bluetooth Low Energy. It’s a power-efficient version of Bluetooth originally introduced in 2010. BLE’s low energy needs are vital to beacons, as it allows them to run for years on tiny coin-cell batteries. It also consumes far less energy than the old and clunky Bluetooth. In fact, BLE is a major driver in the IoT, allowing technology to last longer with smaller parts.

The next question is, how do beacons actually enable connecting and transferring data?

### How does a beacon communicate?

Beacon hardware is relatively simple, but the way it triggers actions can get a little complicated. Every system is a little different, but here’s how a beacon communicates, in a nutshell:

The beacon sends out its ID numbers about ten times every second (sometimes more, sometimes less, depending on its settings). A nearby Bluetooth-enabled device, like your phone, picks up that signal. When a dedicated app recognizes it, it links it to an action or piece of content stored in the cloud and displays it to the user. You can “teach” your app how to react to a beacon signal by developing using third-party tools.

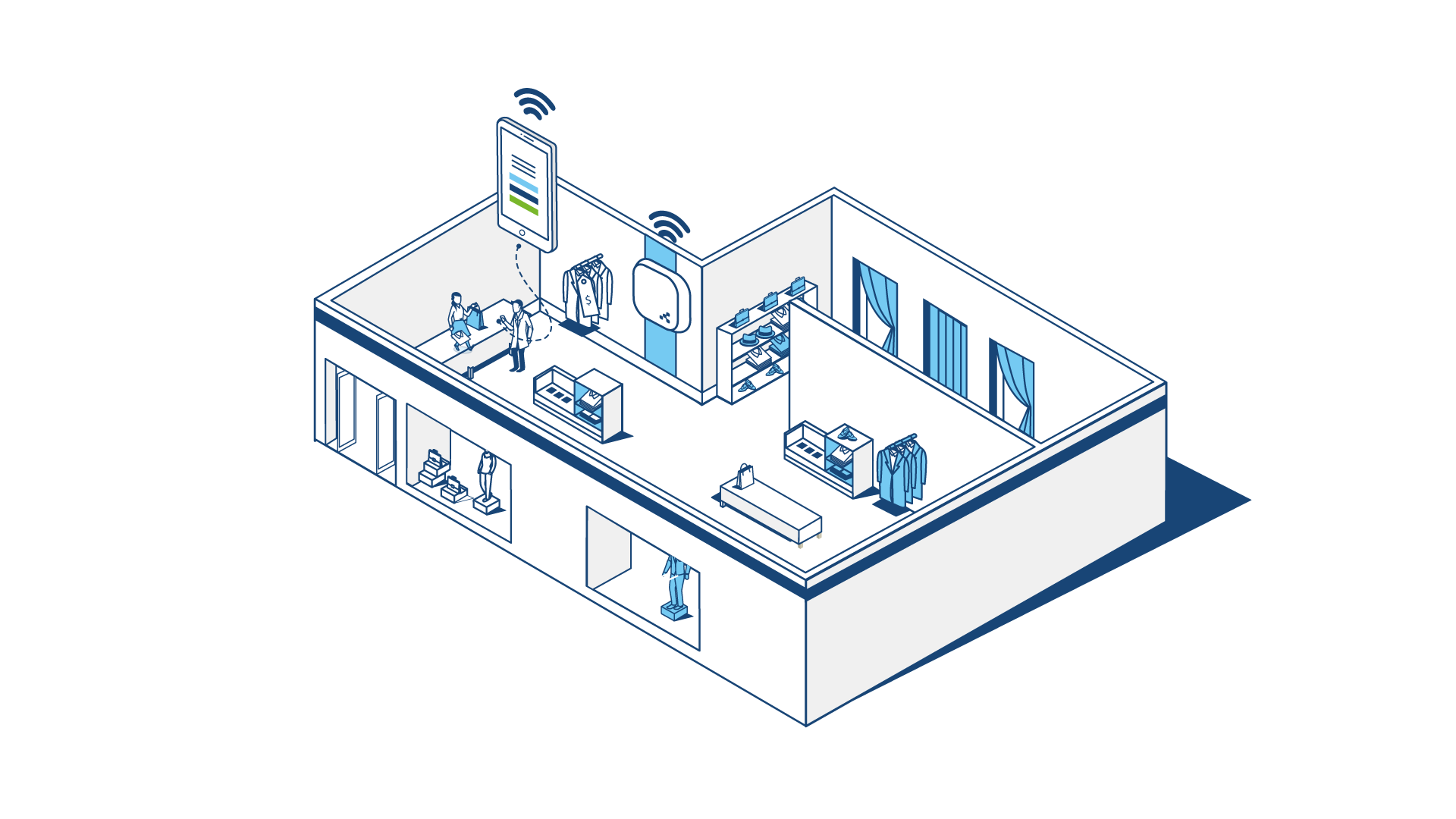


### **Couldn’t the beacon data just be hardcoded into the app?**

Why go through the cloud? It sounds so unnecessary!

The #1 reason you don’t want data hardcoded on your phone is space. Keeping content on the cloud makes your app light and keeps your phone from being bogged down. No one wants to download bulky apps–especially when they’re on the go.

Reason #2 is that content attached to beacons does change. But, remember how beacons only broadcast an ID? That ID doesn’t change too often but the content behind it does. Say you’re a store owner and you want to run a new sales campaign or add a promotion to your existing offerings. If your beacon data is hardcoded, you would have to completely re-release the app. Storing data in the cloud means beacons can be updated almost instantly. It means the app doesn’t have to be altered or re-coded. Once the information is online, it’s ready to go to the beacon.So, what is a beacon going to do to help you? Let’s try an example.



### **So, what is a beacon going to do to help you? Let’s try an example.**

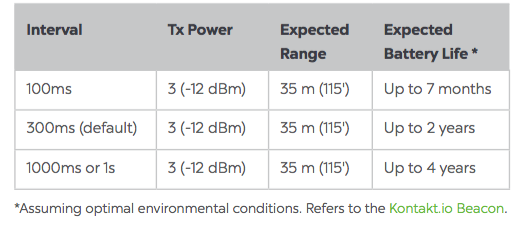
You’re going on a trip to some far-off country—nice! But you don’t speak the language—crap!  
On the way to the hotel, you arrive at a big train station. The station is huge, and you have no idea where to go. How can you get directions that are reliable, clear, and in a language you understand?

Luckily, the train station owner planned for problems just like this. You see a sign indicating that the train station has an app. You download it. A beacon placed on the wall sends out a continuous signal, and, once you activate the app, you’re able to make use out of it. The app takes the beacon ID and checks what information is paired with that particular beacon.

It recognizes that you are standing in front of the donut shop by Gate 14. You enter your destination, and the app generates a clear map to show you the way there. When you turn down the wrong path, it redirects you.  
Since you have plenty of time, the app also lets you know that the coffee shop to your left has a special deal going on. The app tells you all the train schedules and delays for the station.  
You catch your train on time and realize it’s not so stressful after all.

## Where did beacon technology begin?

Today’s beacons began with the introduction of **iBeacon**. iBeacon is simply a protocol that lets Bluetooth devices transmit very small bits of data.  
Then Google entered the scene. In 2015, Google came out with **Eddystone**, their iBeacon alternative. Since then, iBeacon and Eddystone have ruled the proximity market.  
Now, beacon technology is continuing to develop with cooler capabilities, better hardware, and more diverse solutions.



## What are all these numbers?! (reading beacon specs)

What is a beacon packet? Do you need those? Here are some notes on beacon specs and details.

* **Battery life**: Most beacons start with an 18-24 month battery life. However, some beacons with certain requirements and uses last some 6-8 months. Beacons with energy-saving capabilities can last over 5 years.  
  How can beacons last so long with such tiny batteries? Easy! They don’t actually work that hard. They let Bluetooth do all the work, and Bluetooth is incredibly energy efficient.
* **Supported format**: Does your beacon use the iBeacon protocol? Eddystone? Beacons usually support both of these and sometimes the hardware manufacturer’s own format (like AltBeacon).
* **Interval**: How often can the beacon transmit its message? How often you need your beacon to transmit depends on your specific scenario. (ms=millisecond)
* **Tx Power**: The Transmission Power describes how far a beacon can transmit data. This can be as little as 4 meters, but many reach some 50-90 meters. However, it is not necessary that this number be humongous. A 50-meter range beacon can be just as useful as a 90-meter depending on the specific use.
* **Packets**: A beacon’s “packet” is the data it transmits. This just describes the kind of information it is able to transmit. For example, iBeacon contains one packet (iBeacon itself) while Eddystone has three separate ones.
* **Sensors**: Now, beacons are coming out with extra capabilities. They may include accelerometers, light or movement sensors.
* **NFC / RFID**: Beacons are still very new. For some users, it’s highly important that legacy technologies (like NFC and RFID tags) and beacons work together.
* **Price**: Beacons can cost as little as $5. Will such a cheap beacon be worth it? Well, that really depends on what you want, but many users will find that ultra-cheap beacons simply don’t get the job done. Expect one ordinary beacon to run $15-25.

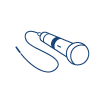
## What is a beacon used for?

    Retail

[](https://webpage-cdn.kontakt.io/uploads/2015/11/10103032/education.png)    Education

   Culture

   Airports

     Events

   Offices

     Hospitality

**What are common use cases?**

Proximity technology is still really popular in retail settings. Pay a visit to Macy’s, American Eagle, or the Tokyo airport, and you may find beacons in place. But beacons are becoming even more popular each year. This means you’re likely to find them just about anywhere you move.



Beacons are already being used for:

* **Tracking**: One of the beacons’ more practical use cases is something many of us would never have guessed. In manufacturing and transport, managers need to know exactly where goods are at any given time. By attaching beacons, they can always have that information. In fact, they can even see the information from previous days or weeks.
* **Navigation**: Creating accurate “GPS for indoor navigation” is a popular beacon use case. What Google Maps does for the outdoors, beacons can do for the indoors. They can tell you where you are and where you’re going in a museum, festival, or train station.
* **Interaction**: Beacons can make reactions automated and trigger events. When you enter a room, the projector starts. It sends notifications or acts as a loyalty card. If you make a purchase at your local cafe, beacons help the app register that you were there. On your tenth entry, you get a free latte—awesome!
* **Security**: Whether it’s making sure patients don’t go in the wrong wing or alerting factory workers to dangerous changes, beacons can automatically send notifications (either to app users or property owners) about a safety issue. Beacons can also be paired with geofencing to add an extra layer to data security.
* **Analysis**: Data is one of the biggest tools at a company’s disposal. Beacons help generate data on where customers are going or where common problems occur on an assembly line. The online platform can store information on which beacons are being triggered and how users are interacting with them.

**Use of Beacons in our project**

**We are looking to:-**

* Support visitors’ navigation in tortuous venues and huge open spaces.
* Enable visitors easily to find a specific point / room / object in a venue .
* Make indoors navigation accurate, directed, and real-time.
* Adjust navigation to dynamically changing spaces like exhibition halls or markets, where booths constantly change their location and ownership.
* Attract people to cities, landmarks, or historical sites through context-rich navigation solutions.
* Help visually impaired people navigate your venue.

**We use beacons to:-**

* Implement an indoor navigation app that will work similarly to GPS for the end user.
* Introduce path and time estimation features .
* Link navigation with the real-time or offline schedule of your event.
* Enable beacon guided tours, which will be available exclusively through in-app purchase or subscription.
* Add environmental information for visually impaired people .
* Integrate beacon and GPS navigation for outdoors / indoors spaces such as universities,

**EXPLANATION OF MODULES USED FOR SCANNING QR CODE THROUGH PYTHON CODE:**

**EXPLANATION OF IMUTILS AND OPENCV**

**IMUTILS**

A series of convenience functions to make basic image processing functions such as translation,

rotation, resizing, skeletonization, and displaying Matplotlib images easier with OpenCV

and **both** Python 2.7 and Python 3.

**Finding function OpenCV functions by name**

OpenCV can be a big, hard to navigate library, especially if you are just getting started learning

computer vision and image processing. The find\_function method allows you to quickly search

function names across modules (and optionally sub-modules) to find the function you are looking

for.

**Translation**

Translation is the shifting of an image in either the x or y direction. To translate an image in

OpenCV you would need to supply the (x, y)-shift, denoted as (tx, ty) to construct the translation

matrix M:

And from there, you would need to apply the cv2.warpAffine function.

Instead of manually constructing the translation matrix M and calling cv2.warpAffine, you can

simply make a call to the translate function of imutils.

**Rotation**

Rotating an image in OpenCV is accomplished by making a call

to cv2.getRotationMatrix2D and cv2.warpAffine. Further care has to be taken to supply the (x,

y)-coordinate of the point the image is to be rotated about. These calculation calls can quickly add

up and make your code bulky and less readable. The rotate function in imutils helps resolve this

problem.

**Example:**

# loop over the angles to rotate the image

for angle in xrange(0, 360, 90):

# rotate the image and display it

rotated = imutils.rotate(bridge, angle=angle)

cv2.imshow("Angle=%d" % (angle), rotated)

**Resizing**

Resizing an image in OpenCV is accomplished by calling the cv2.resize function. However, special

care needs to be taken to ensure that the aspect ratio is maintained. This resize function

of imutils maintains the aspect ratio and provides the keyword arguments width and height so the

image can be resized to the intended width/height while (1) maintaining aspect ratio and (2)

ensuring the dimensions of the image do not have to be explicitly computed by the developer.

Another optional keyword argument, inter, can be used to specify interpolation method as well.

**Example:**

# loop over varying widths to resize the image to

for width in (400, 300, 200, 100):

# resize the image and display it

resized = imutils.resize(workspace, width=width)

cv2.imshow("Width=%dpx" % (width), resized)

**Skeletonization**

Skeletonization is the process of constructing the "topological skeleton" of an object in an image,

where the object is presumed to be white on a black background. OpenCV does not provide a

function to explicitly construct the skeleton, but does provide the morphological and binary

functions to do so.

For convenience, the skeletonize function of imutils can be used to construct the topological

skeleton of the image.

The first argument, size is the size of the structuring element kernel. An optional

argument, structuring, can be used to control the structuring element -- it defaults

to cv2.MORPH\_RECT , but can be any valid structuring element.

**Example:**

# skeletonize the image

gray = cv2.cvtColor(logo, cv2.COLOR\_BGR2GRAY)

skeleton = imutils.skeletonize(gray, size=(3, 3))

cv2.imshow("Skeleton", skeleton)

**USE OF OPENCV AND IMUTILS IN OUR PROJECT**

 We used the opencv library to read the image cv.imread to read the

barcode image that is being displayed using the camera of the laptop.

We generate a rectangular frame that detects the presence of qr code in

the picture and surrounds the QR code with the rectangular frame and

Then fetches the desired data that the qr code displays.

 Then we used the imutils to steam the video so that it would be easier for

us to see scan the QR code and store its data as in the form of integer in

the form of coordinates of the location. We scanned two code the first one

is the locations qr code and the second one is the destination qr code. The

data from these qr code are fetched and then fitted into our algorithms

that find the shortest path to travel to that location using the dijkstra's

algorithm.

**CAPTURE VIDEO FROM CAMERA**

Often, we have to capture live stream with camera. OpenCV provides a very simple interface to this. Let’s capture a video from the camera (I am using the in-built webcam of my laptop), convert it into grayscale video and display it. Just a simple task to get started.

To capture a video, you need to create a VideoCapture object. Its argument can be either the device index or the name of a video file. Device index is just the number to specify which camera. Normally one camera will be connected (as in my case). So I simply pass 0 (or -1). You can select the second camera by passing 1 and so on. After that, you can capture frame-by-frame. But at the end, don’t forget to release the capture.

import numpy as np

import cv2

cap = cv2.VideoCapture(0)

while(True):

# Capture frame-by-frame

ret, frame = cap.read()

# Our operations on the frame come here

gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

# Display the resulting frame

cv2.imshow('frame',gray)

if cv2.waitKey(1) & 0xFF == ord('q'):

break

# When everything done, release the capture

cap.release()

cv2.destroyAllWindows()

cap.read() returns a bool (True/False). If frame is read correctly, it will be True. So you can check end of the video by checking this return value.

Sometimes, cap may not have initialized the capture. In that case, this code shows error. You can check whether it is initialized or not by the method cap.isOpened(). If it is True, OK. Otherwise open it using cap.open().

You can also access some of the features of this video using cap.get(propId) method where propId is a number from 0 to 18. Each number denotes a property of the video (if it is applicable to that video) and full details can be seen here: Property Identifier. Some of these values can be modified using cap.set(propId, value). Value is the new value you want.

For example, I can check the frame width and height by cap.get(3) and cap.get(4). It gives me 640x480 by default. But I want to modify it to 320x240. Just use ret = cap.set(3,320) and ret = cap.set(4,240).

**IMPLEMENTATION**

For demo purposes we have considered the following map of hospital:



In this map all the nodes represent turning point, doctor’s cabin, washroom and other important position.

Python code:-

from imutils.video import VideoStream

from pyzbar import pyzbar

import argparse

import datetime

import imutils

import time

import cv2

from collections import defaultdict

def scanqr():

barcodeData=''

# construct the argument parser and parse the arguments

ap = argparse.ArgumentParser()

ap.add\_argument("-o", "--output", type=str, default="barcodes.csv",

help="path to output CSV file containing barcodes")

args = vars(ap.parse\_args())

# initialize the video stream and allow the camera sensor to warm up

vs = VideoStream(src=0).start()

time.sleep(2.0)

# open the output CSV file for writing and initialize the set of

# barcodes found thus far

csv = open(args["output"], "w")

found = set()

# loop over the frames from the video stream

while True:

# grab the frame from the threaded video stream and resize it to

# have a maximum width of 400 pixels

frame = vs.read()

frame = imutils.resize(frame, width=400)

# find the barcodes in the frame and decode each of the barcodes

barcodes = pyzbar.decode(frame)

# loop over the detected barcodes

for barcode in barcodes:

# extract the bounding box location of the barcode and draw

# the bounding box surrounding the barcode on the image

(x, y, w, h) = barcode.rect

cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 0, 255), 2)

# the barcode data is a bytes object so if we want to draw it

# on our output image we need to convert it to a string first

barcodeData = barcode.data.decode("utf-8")

barcodeType = barcode.type

# draw the barcode data and barcode type on the image

text = "{} ({})".format(barcodeData, barcodeType)

cv2.putText(frame, text, (x, y - 10),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 0, 255), 2)

# if the barcode text is currently not in our CSV file, write

# the timestamp + barcode to disk and update the set

if barcodeData not in found:

csv.write("{},{}\n".format(datetime.datetime.now(),

barcodeData))

csv.flush()

found.add(barcodeData)

# show the output frame

cv2.imshow("Barcode Scanner", frame)

key = cv2.waitKey(1) & 0xFF

# if the `q` key was pressed, break from the loop

if barcodeData!='':

break

# close the output CSV file do a bit of cleanup

#print("[INFO] cleaning up...")

return(barcodeData)

csv.close()

cv2.destroyAllWindows()

vs.stop()

#Class to represent a graph

class Graph:

# A utility function to find the

# vertex with minimum dist value, from

# the set of vertices still in queue

def minDistance(self,dist,queue):

# Initialize min value and min\_index as -1

minimum = float("Inf")

min\_index = -1

# from the dist array,pick one which

# has min value and is till in queue

for i in range(len(dist)):

if dist[i] < minimum and i in queue:

minimum = dist[i]

min\_index = i

return min\_index

# Function to print shortest path

# from source to j

# using parent array

def printPath(self, parent, j):

#Base Case : If j is source

if parent[j] == -1 :

print (j,end='')

return

self.printPath(parent , parent[j])

print ('->',j,end='')

# A utility function to print

# the constructed distance

# array

def printSolution(self, dist, parent,src,des):

i=des

print("%d --> %d \ndistance:%d \t\t\t\t\t" % (src, i, dist[i]))

print('PATH: ',end='')

self.printPath(parent,i)

'''Function that implements Dijkstra's single source shortest path

algorithm for a graph represented using adjacency matrix

representation'''

def dijkstra(self, graph, src,des):

row = len(graph)

col = len(graph[0])

# The output array. dist[i] will hold

# the shortest distance from src to i

# Initialize all distances as INFINITE

dist = [float("Inf")] \* row

#Parent array to store

# shortest path tree

parent = [-1] \* row

# Distance of source vertex

# from itself is always 0

dist[src] = 0

# Add all vertices in queue

queue = []

for i in range(row):

queue.append(i)

#Find shortest path for all vertices

while queue:

# Pick the minimum dist vertex

# from the set of vertices

# still in queue

u = self.minDistance(dist,queue)

# remove min element

queue.remove(u)

# Update dist value and parent

# index of the adjacent vertices of

# the picked vertex. Consider only

# those vertices which are still in

# queue

for i in range(col):

'''Update dist[i] only if it is in queue, there is

an edge from u to i, and total weight of path from

src to i through u is smaller than current value of

dist[i]'''

if graph[u][i] and i in queue:

if dist[u] + graph[u][i] < dist[i]:

dist[i] = dist[u] + graph[u][i]

parent[i] = u

# print the constructed distance array

self.printSolution(dist,parent,src,des)

g= Graph()

graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],

[4, 0, 8, 0, 0, 0, 0, 11, 0],

[0, 8, 0, 7, 0, 4, 0, 0, 2],

[0, 0, 7, 0, 9, 14, 0, 0, 0],

[0, 0, 0, 9, 0, 10, 0, 0, 0],

[0, 0, 4, 14, 10, 0, 2, 0, 0],

[0, 0, 0, 0, 0, 2, 0, 1, 6],

[8, 11, 0, 0, 0, 0, 1, 0, 7],

[0, 0, 2, 0, 0, 0, 6, 7, 0]

]

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Scan the source QR CODE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

s=int(scanqr())

print('Source node pinned successfully to node number : ',s)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Scan the destination QR CODE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

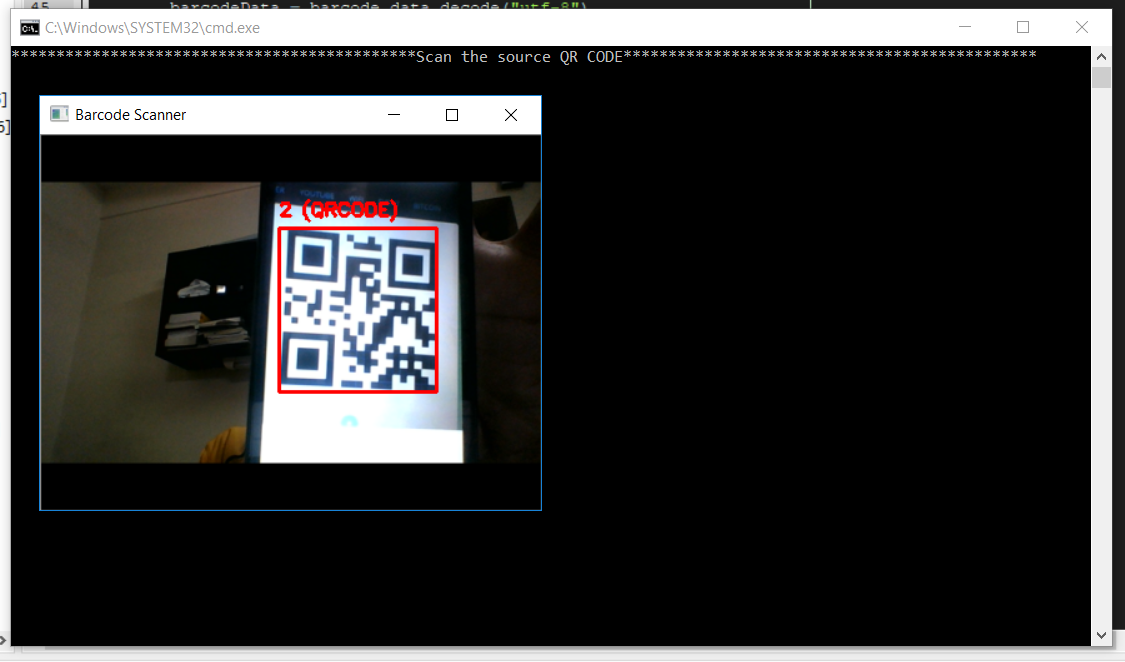
des=int(scanqr())

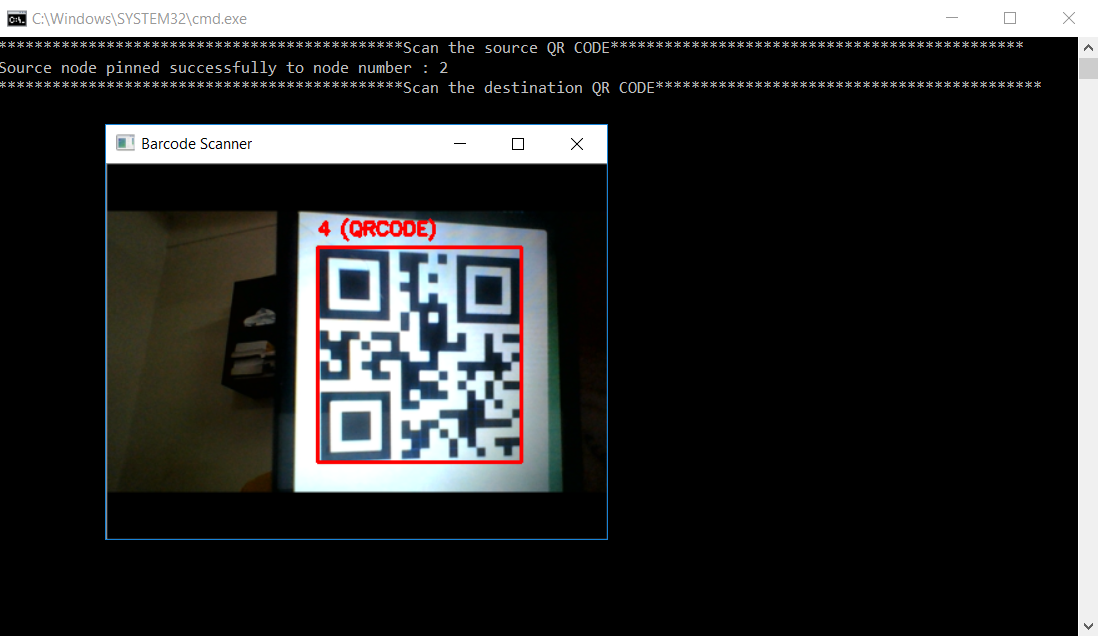
print('Destination node pinned successfully to node number : ',des)

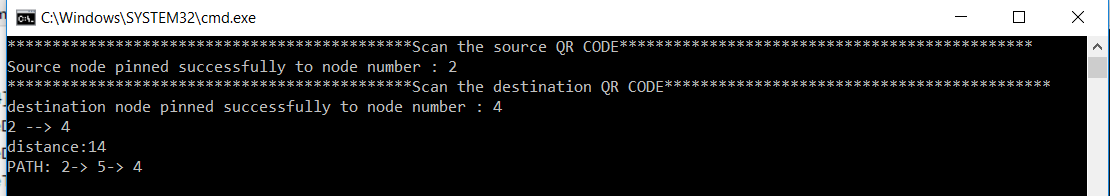
# Print the solution

g.dijkstra(graph,s,des)

**OUTPUT**

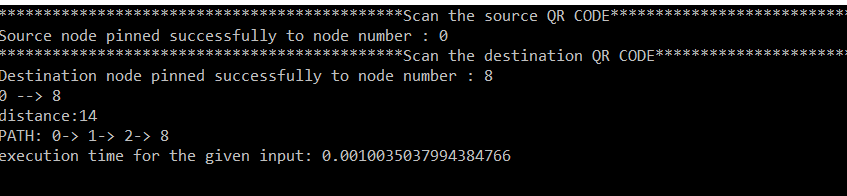
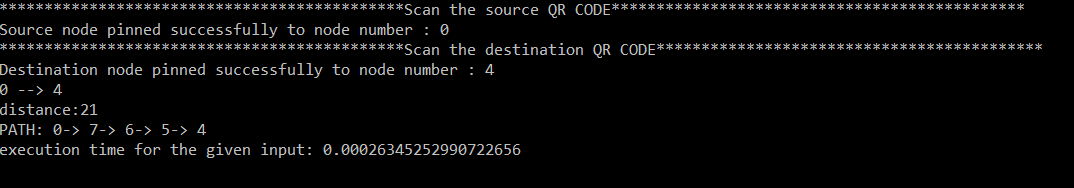
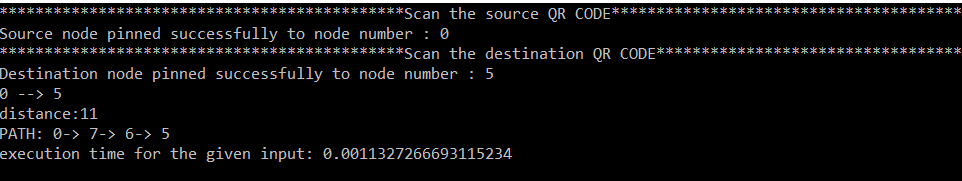
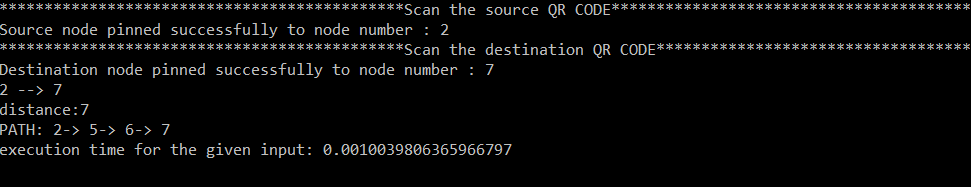
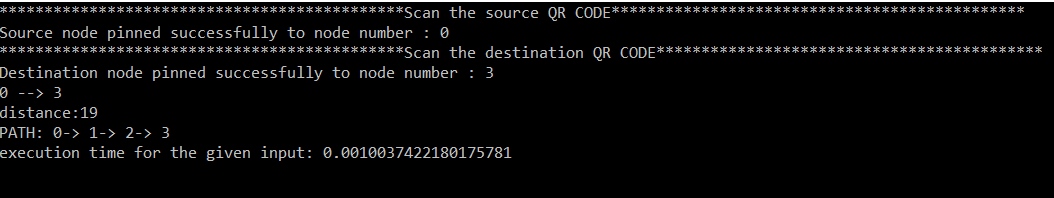
1. Scan QR code for source node(current location)
2. Scan QR code for destination node.



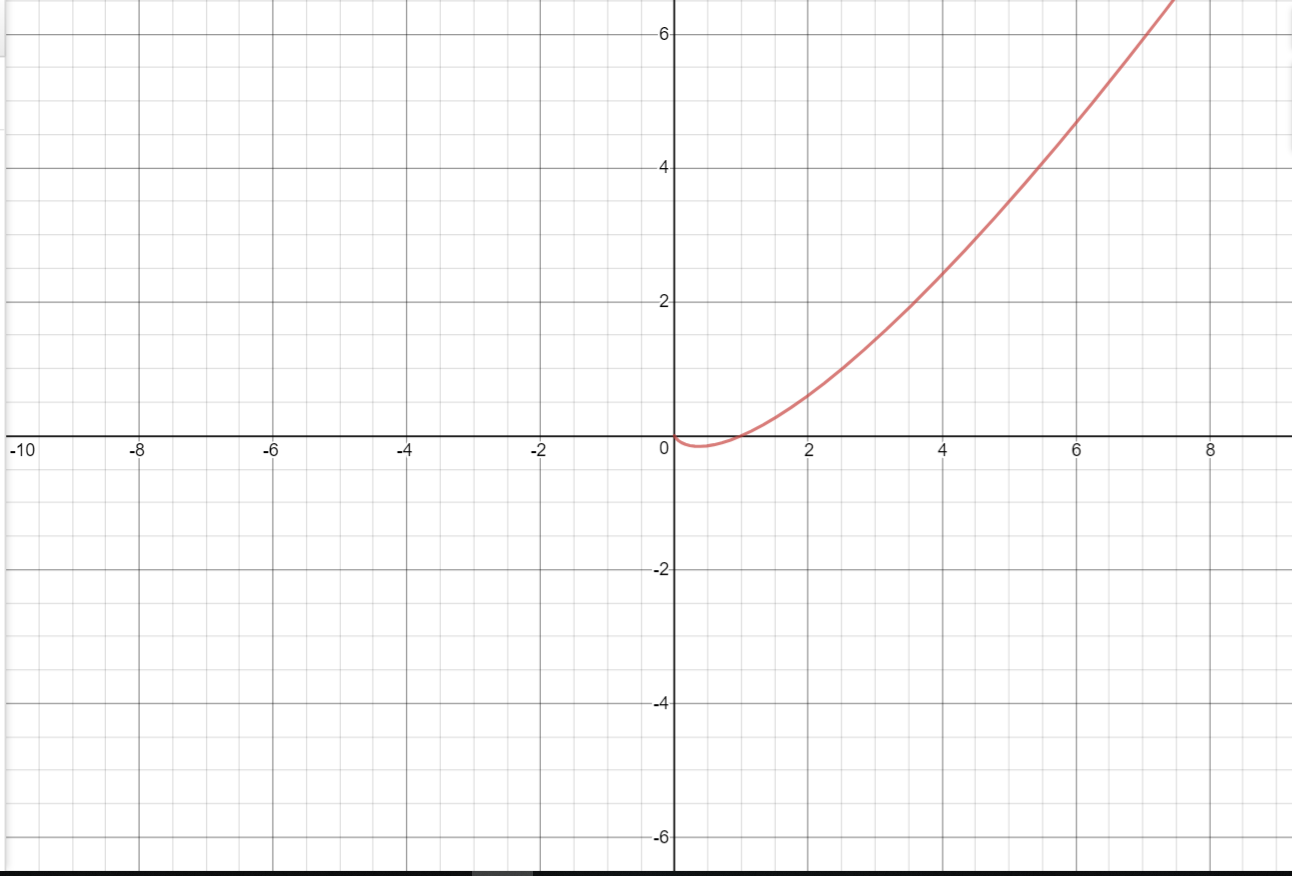
1. System prints the shortest path from source node to destination node.

**RESULT**

**Execution time analysis for various inputs:**

1. 
2. 
3. 
4. 
5. 

**Graph for input size(in byte) vs. execution time:-**

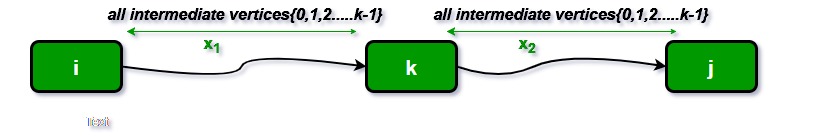


**We used the following algorithms in our project:-**

**(Analysis of every algorithm is given below)**

**Floyd Warshall Algorithm**  
In this algorithm, we initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, .. k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.  
**1)** k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.  
**2)** k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j] if dist[i][j] > dist[i][k] + dist[k][j].

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/dpFloyd-Warshall-.jpg)

Code:

// C Program for Floyd Warshall Algorithm

#include<stdio.h>

// Number of vertices in the graph

#define V 4

/\* Define Infinite as a large enough value. This value will be used

  for vertices not connected to each other \*/

#define INF 99999

// A function to print the solution matrix

**void** printSolution(**int** dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm

**void** floydWarshall (**int** graph[][V])

{

    /\* dist[][] will be the output matrix that will finally have the shortest

      distances between every pair of vertices \*/

**int** dist[V][V], i, j, k;

    /\* Initialize the solution matrix same as input graph matrix. Or

       we can say the initial values of shortest distances are based

       on shortest paths considering no intermediate vertex. \*/

**for** (i = 0; i < V; i++)

**for** (j = 0; j < V; j++)

            dist[i][j] = graph[i][j];

    /\* Add all vertices one by one to the set of intermediate vertices.

      ---> Before start of an iteration, we have shortest distances between all

      pairs of vertices such that the shortest distances consider only the

      vertices in set {0, 1, 2, .. k-1} as intermediate vertices.

      ----> After the end of an iteration, vertex no. k is added to the set of

      intermediate vertices and the set becomes {0, 1, 2, .. k} \*/

**for** (k = 0; k < V; k++)

    {

        // Pick all vertices as source one by one

**for** (i = 0; i < V; i++)

        {

            // Pick all vertices as destination for the

            // above picked source

**for** (j = 0; j < V; j++)

            {

                // If vertex k is on the shortest path from

                // i to j, then update the value of dist[i][j]

**if** (dist[i][k] + dist[k][j] < dist[i][j])

                    dist[i][j] = dist[i][k] + dist[k][j];

            }

        }

    }

    // Print the shortest distance matrix

    printSolution(dist);

}

/\* A utility function to print solution \*/

**void** printSolution(**int** dist[][V])

{

**printf** ("The following matrix shows the shortest distances"

            " between every pair of vertices \n");

**for** (**int** i = 0; i < V; i++)

    {

**for** (**int** j = 0; j < V; j++)

        {

**if** (dist[i][j] == INF)

**printf**("%7s", "INF");

**else**

**printf** ("%7d", dist[i][j]);

        }

**printf**("\n");

    }

}

// driver program to test above function

**int** main()

{

    /\* Let us create the following weighted graph

            10

       (0)------->(3)

        |         /|\

      5 |          |

        |          | 1

       \|/         |

       (1)------->(2)

            3           \*/

**int** graph[V][V] = { {0,   5,  INF, 10},

                        {INF, 0,   3, INF},

                        {INF, INF, 0,   1},

                        {INF, INF, INF, 0}

                      };

    // Print the solution

    floydWarshall(graph);

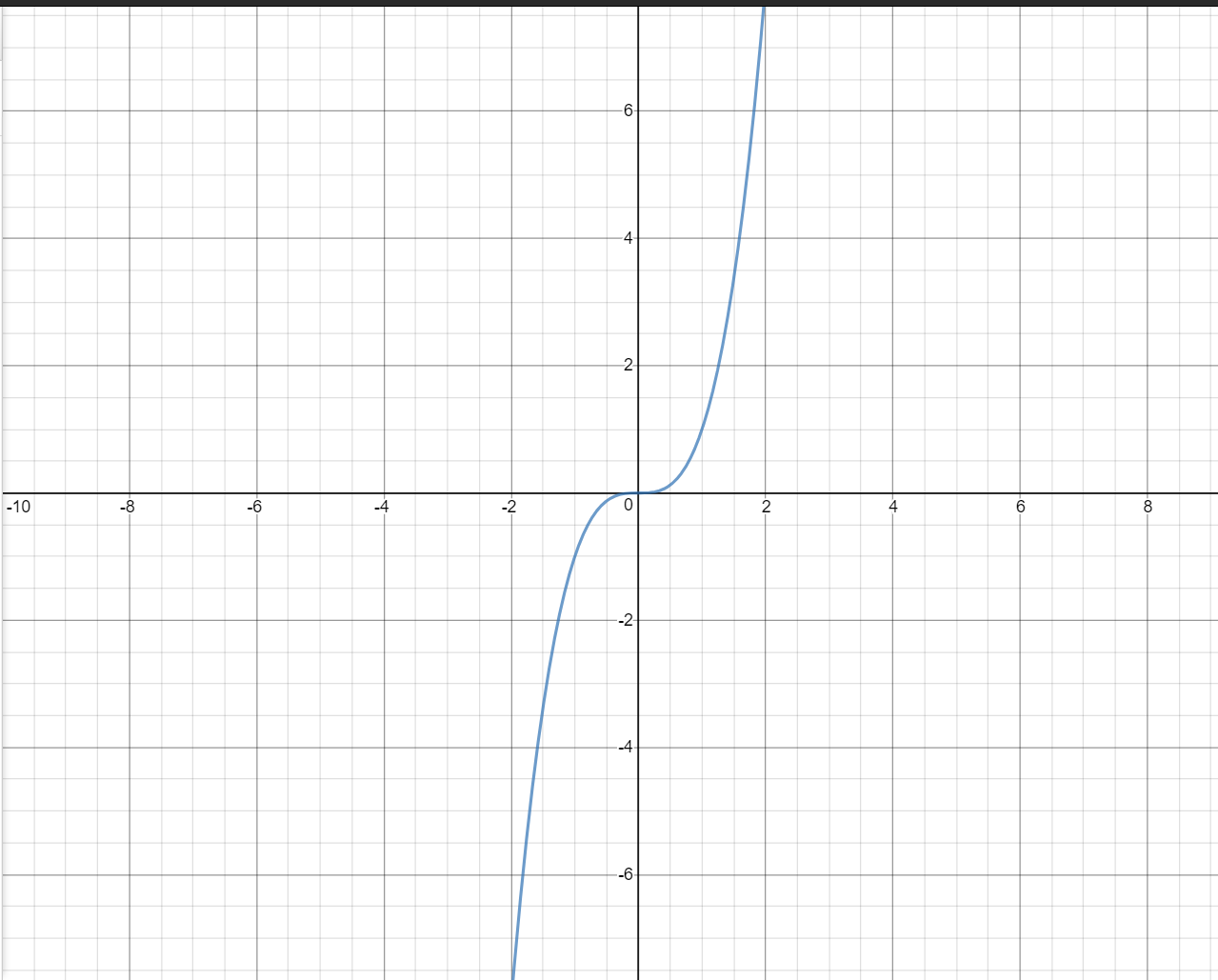
**return** 0;

}

**Time complexity:**

* Floyd Warshall Algorithm consists of three loops over all nodes.
* The inner most loop consists of only operations of a constant complexity.
* Hence, the asymptotic complexity of Floyd-Warshall algorithm is O(n3), where n is the number of nodes in the given graph.

**Graph of time complexity:**



**BFS Shortest Path**

In this algorithm,we first initialize an array dist[0, 1, …., v-1] such that dist[i] stores the distance of vertex i from the source vertex and array pred[0, 1, ….., v-1] such that pred[i] represents the immediate predecessor of the vertex i in the breadth first search starting from the source.  
Now we get the length of the path from source to any other vertex in O(1) time from array d, and for printing the path from source to any vertex we can use array p and that will take O(V) time in worst case as V is the size of array P. So most of the time of algorithm is spent in doing the Breadth first search from given source which we know takes O(V+E) time. Thus time complexity of our algorithm is O(V+E).

**Code:**

// CPP code for printing shortest path between

// two vertices of unweighted graph

#include <bits/stdc++.h>

**using** **namespace** std;

// utility function to form edge between two vertices

// source and dest

**void** add\_edge(vector<**int**> adj[], **int** src, **int** dest)

{

    adj[src].push\_back(dest);

    adj[dest].push\_back(src);

}

// a modified version of BFS that stores predecessor

// of each vertex in array p

// and its distance from source in array d

**bool** BFS(vector<**int**> adj[], **int** src, **int** dest, **int** v,

**int** pred[], **int** dist[])

{

    // a queue to maintain queue of vertices whose

    // adjacency list is to be scanned as per normal

    // DFS algorithm

    list<**int**> queue;

    // boolean array visited[] which stores the

    // information whether ith vertex is reached

    // at least once in the Breadth first search

**bool** visited[v];

    // initially all vertices are unvisited

    // so v[i] for all i is false

    // and as no path is yet constructed

    // dist[i] for all i set to infinity

**for** (**int** i = 0; i < v; i++) {

        visited[i] = **false**;

        dist[i] = INT\_MAX;

        pred[i] = -1;

    }

    // now source is first to be visited and

    // distance from source to itself should be 0

    visited[src] = **true**;

    dist[src] = 0;

    queue.push\_back(src);

    // standard BFS algorithm

**while** (!queue.empty()) {

**int** u = queue.front();

        queue.pop\_front();

**for** (**int** i = 0; i < adj[u].size(); i++) {

**if** (visited[adj[u][i]] == **false**) {

                visited[adj[u][i]] = **true**;

                dist[adj[u][i]] = dist[u] + 1;

                pred[adj[u][i]] = u;

                queue.push\_back(adj[u][i]);

                // We stop BFS when we find

                // destination.

**if** (adj[u][i] == dest)

**return** **true**;

            }

        }

    }

**return** **false**;

}

// utility function to print the shortest distance

// between source vertex and destination vertex

**void** printShortestDistance(vector<**int**> adj[], **int** s,

**int** dest, **int** v)

{

    // predecessor[i] array stores predecessor of

    // i and distance array stores distance of i

    // from s

**int** pred[v], dist[v];

**if** (BFS(adj, s, dest, v, pred, dist) == **false**)

    {

        cout << "Given source and destination"

             << " are not connected";

**return**;

    }

    // vector path stores the shortest path

    vector<**int**> path;

**int** crawl = dest;

    path.push\_back(crawl);

**while** (pred[crawl] != -1) {

        path.push\_back(pred[crawl]);

        crawl = pred[crawl];

    }

    // distance from source is in distance array

    cout << "Shortest path length is : "

        << dist[dest];

    // printing path from source to destination

    cout << "\nPath is::\n";

**for** (**int** i = path.size() - 1; i >= 0; i--)

        cout << path[i] << " ";

}

// Driver program to test above functions

**int** main()

{

    // no. of vertices

**int** v = 8;

    // array of vectors is used to store the graph

    // in the form of an adjacency list

    vector<**int**> adj[v];

    // Creating graph given in the above diagram.

    // add\_edge function takes adjacency list, source

    // and destination vertex as argument and forms

    // an edge between them.

    add\_edge(adj, 0, 1);

    add\_edge(adj, 0, 3);

    add\_edge(adj, 1, 2);

    add\_edge(adj, 3, 4);

    add\_edge(adj, 3, 7);

    add\_edge(adj, 4, 5);

    add\_edge(adj, 4, 6);

    add\_edge(adj, 4, 7);

    add\_edge(adj, 5, 6);

    add\_edge(adj, 6, 7);

**int** source = 0, dest = 7;

    printShortestDistance(adj, source, dest, v);

**return** 0;

}

**Time Complexity:**

The Time complexity of both BFS and DFS will be O(V + E), where V is the number of vertices, and E is the number of Edges. This again **depends on the data strucure**that we user to represent the graph. If it is an adjacency matrix, it will be O(V^2) . If we use an adjacency list, it will be O(V+E). Now , to explain how, lets us consider the difference between a sparsely connected graph and a densely connected graph.

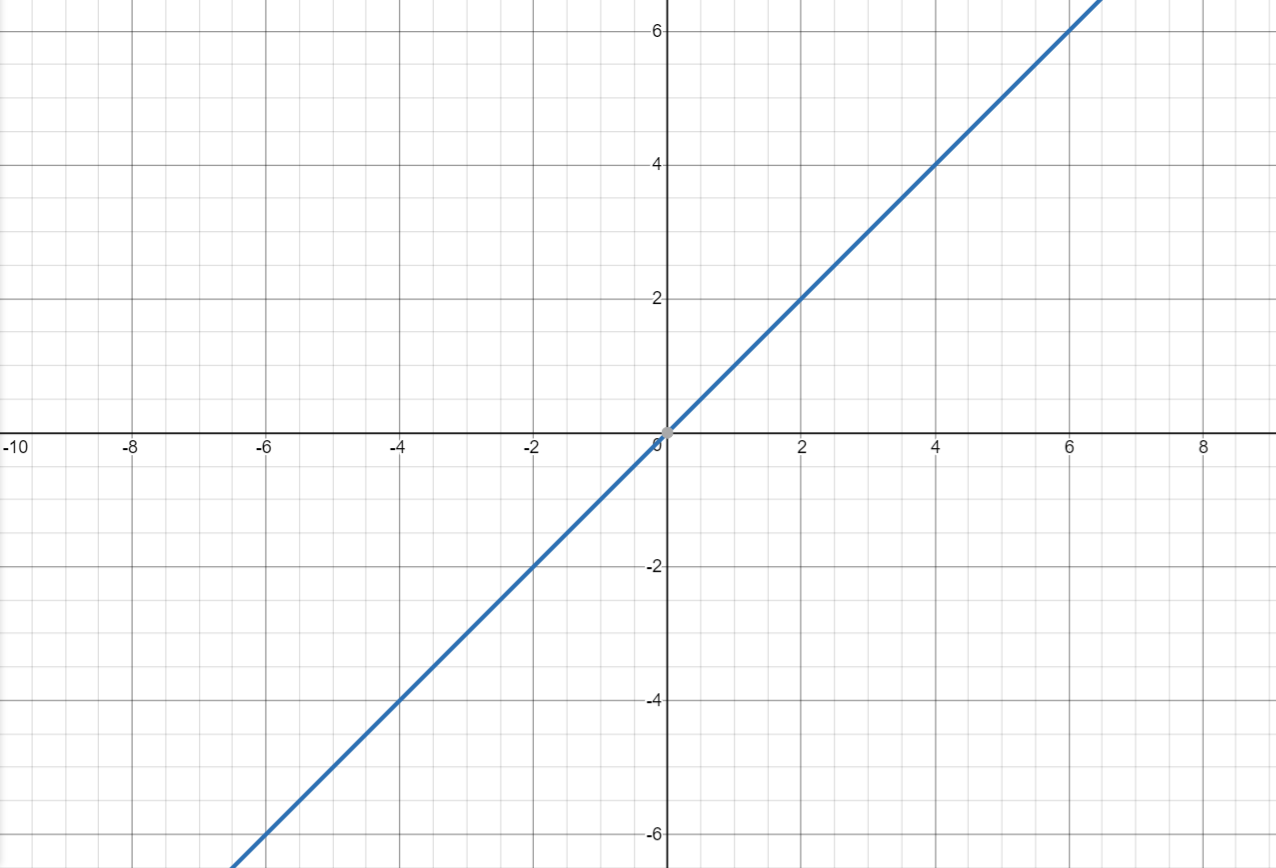
Dense graph is a graph in which the number of edges is close to the maximal number of edges. Sparse graph is a graph in which the number of edges is close to the minimal number of edges. Sparse graph can be disconnected.

Now, what this means is that, if the graph has so many edges and is still a small graph with only few vertices(for example, a graph with 100 vertices but with 4950 edges since it is totally connected( **n\*(n-1)/2** ), then the number of iterations or recursive calls is **dominated by the number of edges**, because 4950 > 100. Now as n -> very large , this difference matters. This kind of graph is a dense graph. Time complexity will be O(E)

Consider an other example , where the number of vertices is very much larger than the number of edges. This kind of graph is a sparsely connected graph. And it may or may not be a disconnected graph. Now consider if the graph has 1000 vertices, but only 100 edges. Here, some of the vertices are disconnected. So, the BFS in this case will be purely **dominated by the number of Vertices**, rather than by the number of Edges. So, the time complexity will be O(V).

So, what does it mean by O(V + E) ? It means , whichever term is bigger will dominate the time complexity. That is why the time complexity of BFS is O(V+E).

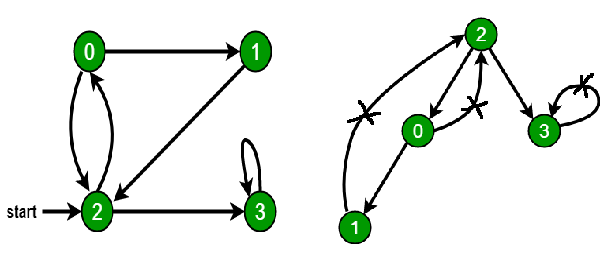
**Graph of Time Complexity:**



**DFS Shortest Path**

[Depth First Traversal (or Search)](http://en.wikipedia.org/wiki/Depth-first_search) for a graph is similar to [Depth First Traversal of a tree](https://www.geeksforgeeks.org/archives/618). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/cycle.png)

See [this post](https://www.geeksforgeeks.org/archives/11644) for all applications of Depth First Traversal.  
Following are implementations of simple Depth First Traversal. The C++ implementation uses [adjacency list representation](https://www.geeksforgeeks.org/graph-and-its-representations/) of graphs. [STL](https://www.geeksforgeeks.org/the-c-standard-template-library-stl/)‘s [list container](http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#LIST) is used to store lists of adjacent nodes.

**Code:**

// C++ program to print DFS traversal from

// a given vertex in a  given graph

#include<iostream>

#include<list>

**using** **namespace** std;

// Graph class represents a directed graph

// using adjacency list representation

**class** Graph

{

**int** V;    // No. of vertices

    // Pointer to an array containing

    // adjacency lists

    list<**int**> \*adj;

    // A recursive function used by DFS

**void** DFSUtil(**int** v, **bool** visited[]);

**public**:

    Graph(**int** V);   // Constructor

    // function to add an edge to graph

**void** addEdge(**int** v, **int** w);

    // DFS traversal of the vertices

    // reachable from v

**void** DFS(**int** v);

};

Graph::Graph(**int** V)

{

**this**->V = V;

    adj = **new** list<**int**>[V];

}

**void** Graph::addEdge(**int** v, **int** w)

{

    adj[v].push\_back(w); // Add w to v’s list.

}

**void** Graph::DFSUtil(**int** v, **bool** visited[])

{

    // Mark the current node as visited and

    // print it

    visited[v] = **true**;

    cout << v << " ";

    // Recur for all the vertices adjacent

    // to this vertex

    list<**int**>::iterator i;

**for** (i = adj[v].begin(); i != adj[v].end(); ++i)

**if** (!visited[\*i])

            DFSUtil(\*i, visited);

}

// DFS traversal of the vertices reachable from v.

// It uses recursive DFSUtil()

**void** Graph::DFS(**int** v)

{

    // Mark all the vertices as not visited

**bool** \*visited = **new** **bool**[V];

**for** (**int** i = 0; i < V; i++)

        visited[i] = **false**;

    // Call the recursive helper function

    // to print DFS traversal

    DFSUtil(v, visited);

}

**int** main()

{

    // Create a graph given in the above diagram

    Graph g(4);

    g.addEdge(0, 1);

    g.addEdge(0, 2);

    g.addEdge(1, 2);

    g.addEdge(2, 0);

    g.addEdge(2, 3);

    g.addEdge(3, 3);

    cout << "Following is Depth First Traversal"

            " (starting from vertex 2) \n";

    g.DFS(2);

**return** 0;

}

**Time Complexity For DFS Algorithm:**

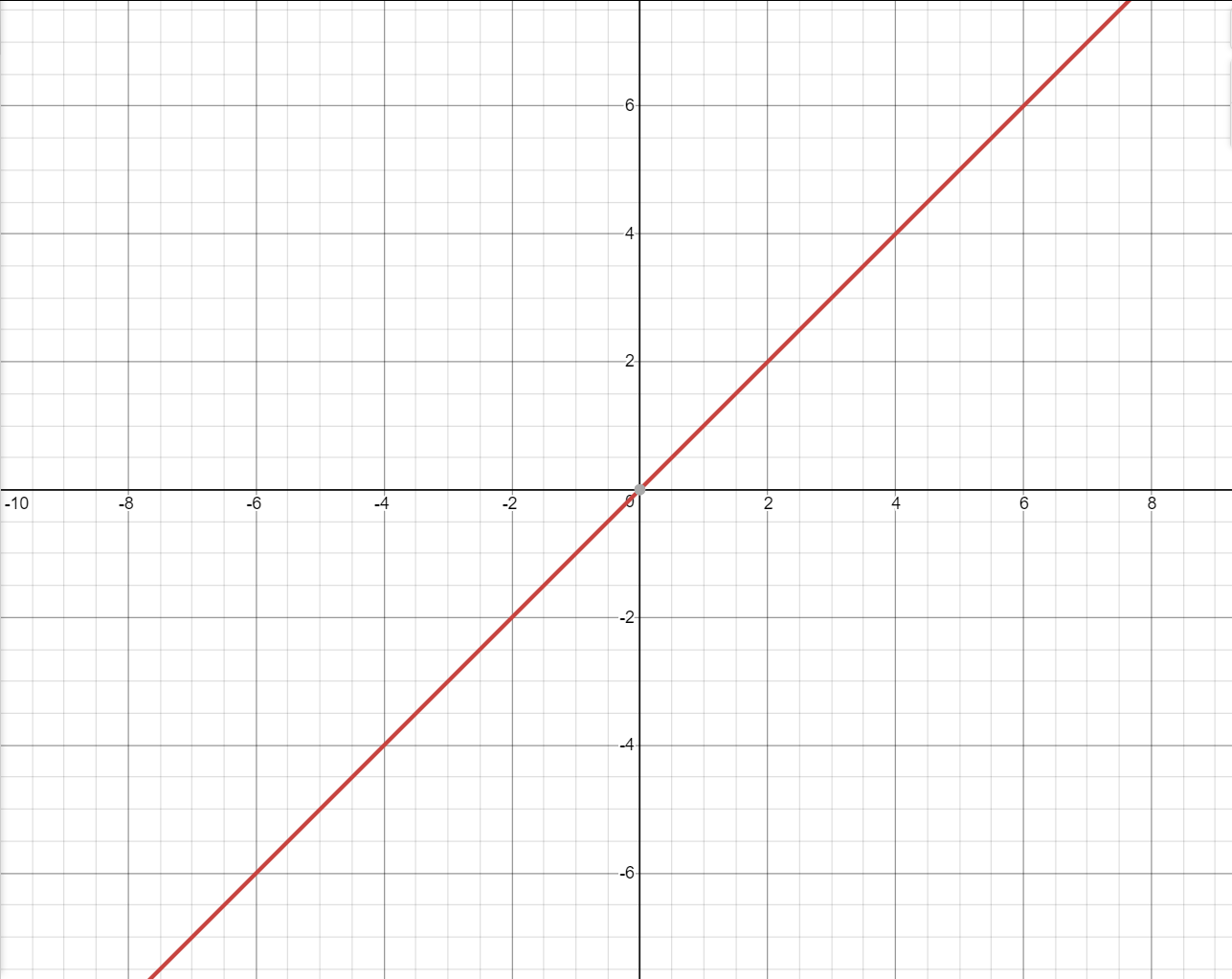
Say, you have a connected graph with V nodes and E edges.

In DFS, you traverse each node exactly once. Therefore, the time complexity of DFS is at least **O(V)**.

Now, any additional complexity comes from how you discover all the outgoing paths or edges for each node which, in turn, is dependent on the way your graph is implemented. If an edge leads you to a node that has already been traversed, you skip it and check the next. Typical DFS implementations use a hash table to maintain the list of traversed nodes so that you could find out if a node has been encountered before in O(1) time (constant time).

* If your graph is implemented as an adjacency matrix (a V x V array), then, for each node, you have to traverse an entire row of length V in the matrix to discover all its outgoing edges. Please note that each row in an adjacency matrix corresponds to a node in the graph, and the said row stores information about edges stemming from the node. So, the complexity of DFS is **O(V \* V) = O(V^2)**.
* If your graph is implemented using adjacency lists, wherein each node maintains a list of all its adjacent edges, then, for each node, you could discover all its neighbors by traversing its adjacency list just once in linear time. For a directed graph, the sum of the sizes of the adjacency lists of all the nodes is E (total number of edges). So, the complexity of DFS is **O(V) + O(E) = O(V + E)**.
  + For an undirected graph, each edge will appear twice. Once in the adjacency list of either end of the edge. So, the overall complexity will be **O(V) + O (2E) ~ O(V + E)**.
* There are different other ways to implement a graph. You can reason the complexity accordingly.

**Graph of Time Complexity:**



**Dijkstra's algorithm**

Dijkstra’s algorithm is very similar to [Prim’s algorithm for minimum spanning tree](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/). Like Prim’s MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.  
Algorithm  
**1)** Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.  
**2)** Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.  
**3)** While sptSet doesn’t include all vertices  
….**a)** Pick a vertex u which is not there in sptSet and has minimum distance value.  
….**b)** Include u to sptSet.  
….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Let us understand with the following example:  
[](https://www.geeksforgeeks.org/wp-content/uploads/Fig-11.jpg)

The set sptSet is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}. After including 0 to sptSet, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green colour.

[](https://www.geeksforgeeks.org/wp-content/uploads/MST1.jpg)

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.

[](https://www.geeksforgeeks.org/wp-content/uploads/DIJ2.jpg)

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).  
[](https://www.geeksforgeeks.org/wp-content/uploads/DIJ3.jpg)

Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.

[](https://www.geeksforgeeks.org/wp-content/uploads/DIJ4.jpg)

We repeat the above steps until sptSet doesn’t include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

[](https://www.geeksforgeeks.org/wp-content/uploads/DIJ5.jpg)

**Code:**

// A C++ program for Dijkstra's single source shortest path algorithm.

// The program is for adjacency matrix representation of the graph

#include <stdio.h>

#include <limits.h>

// Number of vertices in the graph

#define V 9

// A utility function to find the vertex with minimum distance value, from

// the set of vertices not yet included in shortest path tree

**int** minDistance(**int** dist[], **bool** sptSet[])

{

   // Initialize min value

**int** min = INT\_MAX, min\_index;

**for** (**int** v = 0; v < V; v++)

**if** (sptSet[v] == **false** && dist[v] <= min)

         min = dist[v], min\_index = v;

**return** min\_index;

}

// A utility function to print the constructed distance array

**int** printSolution(**int** dist[], **int** n)

{

**printf**("Vertex   Distance from Source\n");

**for** (**int** i = 0; i < V; i++)

**printf**("%d tt %d\n", i, dist[i]);

}

// Function that implements Dijkstra's single source shortest path algorithm

// for a graph represented using adjacency matrix representation

**void** dijkstra(**int** graph[V][V], **int** src)

{

**int** dist[V];     // The output array.  dist[i] will hold the shortest

                      // distance from src to i

**bool** sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest

                     // path tree or shortest distance from src to i is finalized

     // Initialize all distances as INFINITE and stpSet[] as false

**for** (**int** i = 0; i < V; i++)

        dist[i] = INT\_MAX, sptSet[i] = **false**;

     // Distance of source vertex from itself is always 0

     dist[src] = 0;

     // Find shortest path for all vertices

**for** (**int** count = 0; count < V-1; count++)

     {

       // Pick the minimum distance vertex from the set of vertices not

       // yet processed. u is always equal to src in the first iteration.

**int** u = minDistance(dist, sptSet);

       // Mark the picked vertex as processed

       sptSet[u] = **true**;

       // Update dist value of the adjacent vertices of the picked vertex.

**for** (**int** v = 0; v < V; v++)

         // Update dist[v] only if is not in sptSet, there is an edge from

         // u to v, and total weight of path from src to  v through u is

         // smaller than current value of dist[v]

**if** (!sptSet[v] && graph[u][v] && dist[u] != INT\_MAX

                                       && dist[u]+graph[u][v] < dist[v])

            dist[v] = dist[u] + graph[u][v];

     }

     // print the constructed distance array

     printSolution(dist, V);

}

// driver program to test above function

**int** main()

{

   /\* Let us create the example graph discussed above \*/

**int** graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},

                      {4, 0, 8, 0, 0, 0, 0, 11, 0},

                      {0, 8, 0, 7, 0, 4, 0, 0, 2},

                      {0, 0, 7, 0, 9, 14, 0, 0, 0},

                      {0, 0, 0, 9, 0, 10, 0, 0, 0},

                      {0, 0, 4, 14, 10, 0, 2, 0, 0},

                      {0, 0, 0, 0, 0, 2, 0, 1, 6},

                      {8, 11, 0, 0, 0, 0, 1, 0, 7},

                      {0, 0, 2, 0, 0, 0, 6, 7, 0}

                     };

    dijkstra(graph, 0);

**return** 0;

}

**Time Complexity:**

In worst case graph will be a complete graph i.e total edges= v(v-1)/2 where v is no of vertices.

That is : e>>v and e ~ v^2

Time Complexity of Dijkstra's algorithms is:  
1. With Adjacency List and Priority queue:

O((v+e) log v)

-> in worst case: e>>v so O( e log v)

2. With matrix and Priority queue:

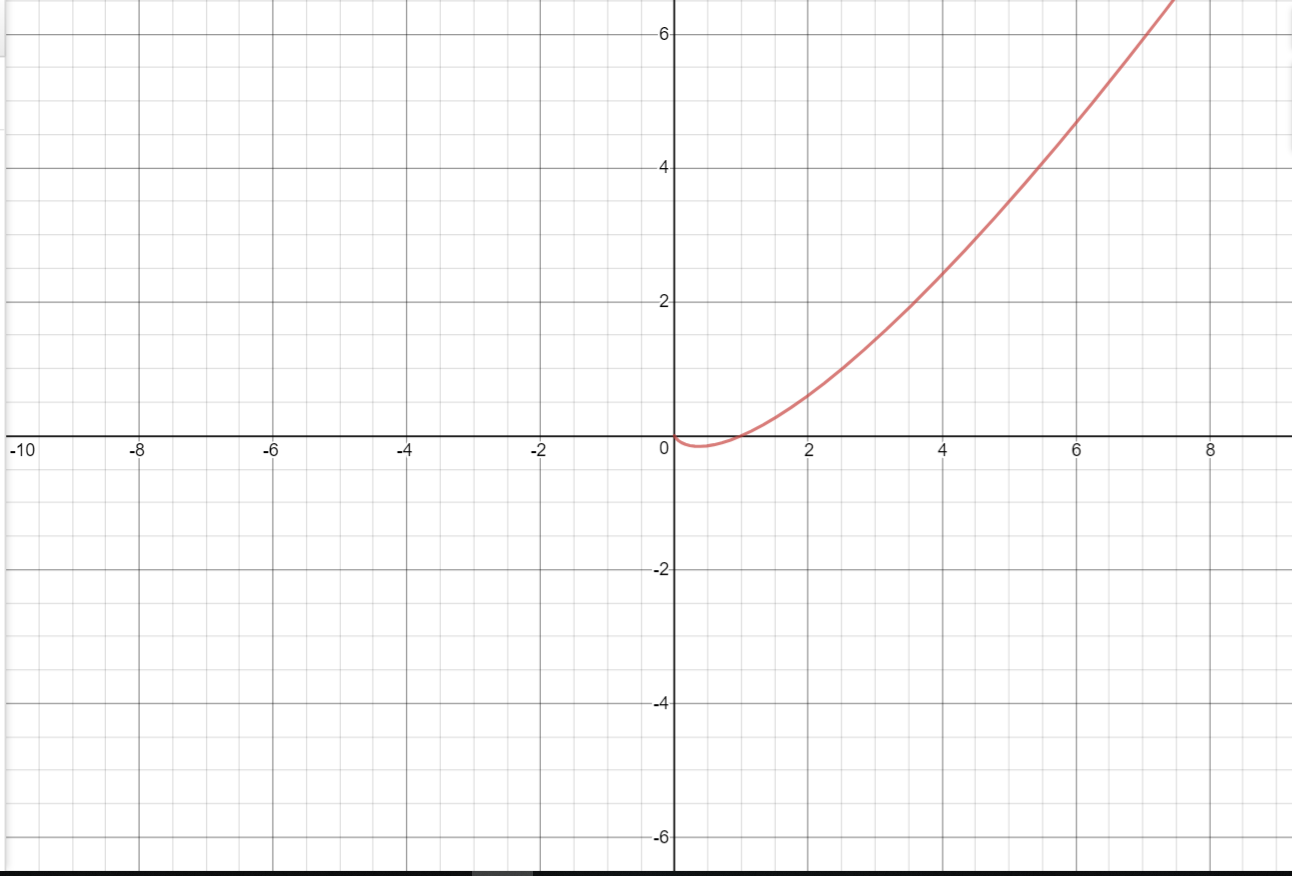
O(v^2 + e log v)

-> in Worst case e ~ v^2  
So O(v^2 + e log v) ~ O(e + e log v) ~ O(e log v).

3. With Fibonacci Heap and adjacency list :

it will be O( e + v log v)

**Graph for Time Complexity:**



**A\* Algorithm**

A\* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A\* Search Algorithm comes to the rescue.

What A\* Search Algorithm does is that at each step it picks the node according to a value-‘**f**’ which is a parameter equal to the sum of two other parameters – ‘**g**’ and ‘**h**’. At each step it picks the node/cell having the lowest ‘**f**’, and process that node/cell.

We define ‘**g**’ and ‘**h**’ as simply as possible below

**g** = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.  
**h** = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.).

We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

// A\* Search Algorithm

1. Initialize the open list

2. Initialize the closed list

put the starting node on the open

list (you can leave its **f** at zero)

3. while the open list is not empty

a) find the node with the least **f** on

the open list, call it "q"

b) pop q off the open list

c) generate q's 8 successors and set their

parents to q

d) for each successor

i) if successor is the goal, stop search

successor.**g** = q.**g** + distance between

successor and q

successor.**h** = distance from goal to

successor (This can be done using many

ways, we will discuss three heuristics-

Manhattan, Diagonal and Euclidean

Heuristics)

successor.**f** = successor.**g** + successor.**h**

ii) if a node with the same position as

successor is in the OPEN list which has a

lower **f** than successor, skip this successor

iii) if a node with the same position as

successor is in the CLOSED list which has

a lower **f** than successor, skip this successor

otherwise, add the node to the open list

end (for loop)

e) push q on the closed list

end (while loop)

So suppose as in the below figure if we want to reach the target cell from the source cell, then the A\* Search algorithm would follow path as shown below. Note that the below figure is made by considering Euclidean Distance as a heuristics.

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/a_-search-algorithm-2.png)

**Heuristics**  
We can calculate **g** but how to calculate h ?

We can do things.  
A) Either calculate the exact value of h (which is certainly time consuming).  
             OR  
B ) Approximate the value of h using some heuristics (less time consuming).

We will discuss both of the methods.

A) **Exact Heuristics** –

We can find exact values of h, but that is generally very time consuming.

Below are some of the methods to calculate the exact value of h.

1) Pre-compute the distance between each pair of cells before running the A\* Search Algorithm.

2) If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the [distance formula/Euclidean Distance](https://en.wikipedia.org/wiki/Euclidean_distance)

**B) Approximation Heuristics –**

There are generally three approximation heuristics to calculate h –

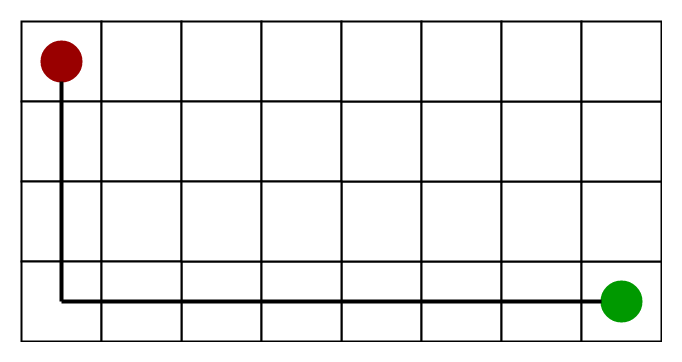
**1) Manhattan Distance –**

* It is nothing but the sum of absolute values of differences in the goal’s x and y coordinates and the current cell’s x and y coordinates respectively, i.e.,

**h** = abs (current\_cell.x – goal.x) +

abs (current\_cell.y – goal.y)

* When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

The Manhattan Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/a_-search-algorithm-3.png)

**2) Diagonal Distance-**

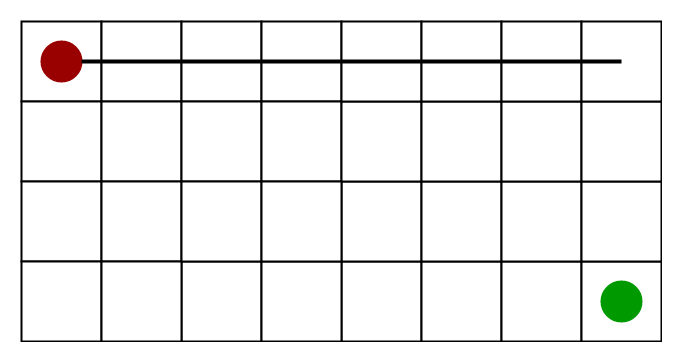
* It is nothing but the maximum of absolute values of differences in the goal’s x and y coordinates and the current cell’s x and y coordinates respectively, i.e.,

**h** = max { abs(current\_cell.x – goal.x),

abs(current\_cell.y – goal.y) }

* When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

The Diagonal Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/a_-search-algorithm-4.png)

**3) Euclidean Distance-**

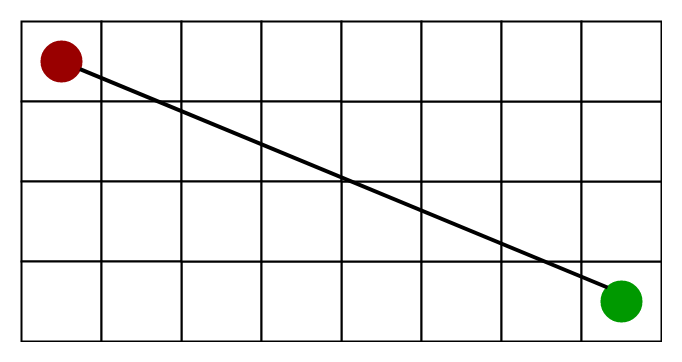
* As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula

**h** = sqrt ( (current\_cell.x – goal.x)2 +

(current\_cell.y – goal.y)2 )

* When to use this heuristic? – When we are allowed to move in any directions.

The Euclidean Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).

[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/a_-search-algorithm-5.png)

**Relation (Similarity and Differences) with other algorithms-**  
Dijkstra is a special case of A\* Search Algorithm, where h = 0 for all nodes.

**Implementation**  
We can use any data structure to implement open list and closed list but for best performance we use a **set** data structure of C++ STL(implemented as Red-Black Tree) and a boolean hash table for a closed list.

The implementations are similar to Dijsktra’s algorithm. If we use a Fibonacci heap to implement the open list instead of a binary heap/self-balancing tree, then the performance will become better (as Fibonacci heap takes O(1) average time to insert into open list and to decrease key)

**Code:**

// A C++ Program to implement A\* Search Algorithm

#include<bits/stdc++.h>

**using** **namespace** std;

#define ROW 9

#define COL 10

// Creating a shortcut for int, int pair type

**typedef** pair<**int**, **int**> Pair;

// Creating a shortcut for pair<int, pair<int, int>> type

**typedef** pair<**double**, pair<**int**, **int**>> pPair;

// A structure to hold the neccesary parameters

**struct** cell

{

    // Row and Column index of its parent

    // Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1

**int** parent\_i, parent\_j;

    // f = g + h

**double** f, g, h;

};

// A Utility Function to check whether given cell (row, col)

// is a valid cell or not.

**bool** isValid(**int** row, **int** col)

{

    // Returns true if row number and column number

    // is in range

**return** (row >= 0) && (row < ROW) &&

           (col >= 0) && (col < COL);

}

// A Utility Function to check whether the given cell is

// blocked or not

**bool** isUnBlocked(**int** grid[][COL], **int** row, **int** col)

{

    // Returns true if the cell is not blocked else false

**if** (grid[row][col] == 1)

**return** (**true**);

**else**

**return** (**false**);

}

// A Utility Function to check whether destination cell has

// been reached or not

**bool** isDestination(**int** row, **int** col, Pair dest)

{

**if** (row == dest.first && col == dest.second)

**return** (**true**);

**else**

**return** (**false**);

}

// A Utility Function to calculate the 'h' heuristics.

**double** calculateHValue(**int** row, **int** col, Pair dest)

{

    // Return using the distance formula

**return** ((**double**)**sqrt** ((row-dest.first)\*(row-dest.first)

                          + (col-dest.second)\*(col-dest.second)));

}

// A Utility Function to trace the path from the source

// to destination

**void** tracePath(cell cellDetails[][COL], Pair dest)

{

**printf** ("\nThe Path is ");

**int** row = dest.first;

**int** col = dest.second;

    stack<Pair> Path;

**while** (!(cellDetails[row][col].parent\_i == row

             && cellDetails[row][col].parent\_j == col ))

    {

        Path.push (make\_pair (row, col));

**int** temp\_row = cellDetails[row][col].parent\_i;

**int** temp\_col = cellDetails[row][col].parent\_j;

        row = temp\_row;

        col = temp\_col;

    }

    Path.push (make\_pair (row, col));

**while** (!Path.empty())

    {

        pair<**int**,**int**> p = Path.top();

        Path.pop();

**printf**("-> (%d,%d) ",p.first,p.second);

    }

**return**;

}

// A Function to find the shortest path between

// a given source cell to a destination cell according

// to A\* Search Algorithm

**void** aStarSearch(**int** grid[][COL], Pair src, Pair dest)

{

    // If the source is out of range

**if** (isValid (src.first, src.second) == **false**)

    {

**printf** ("Source is invalid\n");

**return**;

    }

    // If the destination is out of range

**if** (isValid (dest.first, dest.second) == **false**)

    {

**printf** ("Destination is invalid\n");

**return**;

    }

    // Either the source or the destination is blocked

**if** (isUnBlocked(grid, src.first, src.second) == **false** ||

            isUnBlocked(grid, dest.first, dest.second) == **false**)

    {

**printf** ("Source or the destination is blocked\n");

**return**;

    }

    // If the destination cell is the same as source cell

**if** (isDestination(src.first, src.second, dest) == **true**)

    {

**printf** ("We are already at the destination\n");

**return**;

    }

    // Create a closed list and initialise it to false which means

    // that no cell has been included yet

    // This closed list is implemented as a boolean 2D array

**bool** closedList[ROW][COL];

**memset**(closedList, **false**, **sizeof** (closedList));

    // Declare a 2D array of structure to hold the details

    //of that cell

    cell cellDetails[ROW][COL];

**int** i, j;

**for** (i=0; i<ROW; i++)

    {

**for** (j=0; j<COL; j++)

        {

            cellDetails[i][j].f = FLT\_MAX;

            cellDetails[i][j].g = FLT\_MAX;

            cellDetails[i][j].h = FLT\_MAX;

            cellDetails[i][j].parent\_i = -1;

            cellDetails[i][j].parent\_j = -1;

        }

    }

    // Initialising the parameters of the starting node

    i = src.first, j = src.second;

    cellDetails[i][j].f = 0.0;

    cellDetails[i][j].g = 0.0;

    cellDetails[i][j].h = 0.0;

    cellDetails[i][j].parent\_i = i;

    cellDetails[i][j].parent\_j = j;

    /\*

     Create an open list having information as-

     <f, <i, j>>

     where f = g + h,

     and i, j are the row and column index of that cell

     Note that 0 <= i <= ROW-1 & 0 <= j <= COL-1

     This open list is implenented as a set of pair of pair.\*/

    set<pPair> openList;

    // Put the starting cell on the open list and set its

    // 'f' as 0

    openList.insert(make\_pair (0.0, make\_pair (i, j)));

    // We set this boolean value as false as initially

    // the destination is not reached.

**bool** foundDest = **false**;

**while** (!openList.empty())

    {

        pPair p = \*openList.begin();

        // Remove this vertex from the open list

        openList.erase(openList.begin());

        // Add this vertex to the closed list

        i = p.second.first;

        j = p.second.second;

        closedList[i][j] = **true**;

       /\*

        Generating all the 8 successor of this cell

            N.W   N   N.E

              \   |   /

               \  |  /

            W----Cell----E

                 / | \

               /   |  \

            S.W    S   S.E

        Cell-->Popped Cell (i, j)

        N -->  North       (i-1, j)

        S -->  South       (i+1, j)

        E -->  East        (i, j+1)

        W -->  West           (i, j-1)

        N.E--> North-East  (i-1, j+1)

        N.W--> North-West  (i-1, j-1)

        S.E--> South-East  (i+1, j+1)

        S.W--> South-West  (i+1, j-1)\*/

        // To store the 'g', 'h' and 'f' of the 8 successors

**double** gNew, hNew, fNew;

        //----------- 1st Successor (North) ------------

        // Only process this cell if this is a valid one

**if** (isValid(i-1, j) == **true**)

        {

            // If the destination cell is the same as the

            // current successor

**if** (isDestination(i-1, j, dest) == **true**)

            {

                // Set the Parent of the destination cell

                cellDetails[i-1][j].parent\_i = i;

                cellDetails[i-1][j].parent\_j = j;

**printf** ("The destination cell is found\n");

                tracePath (cellDetails, dest);

                foundDest = **true**;

**return**;

            }

            // If the successor is already on the closed

            // list or if it is blocked, then ignore it.

            // Else do the following

**else** **if** (closedList[i-1][j] == **false** &&

                     isUnBlocked(grid, i-1, j) == **true**)

            {

                gNew = cellDetails[i][j].g + 1.0;

                hNew = calculateHValue (i-1, j, dest);

                fNew = gNew + hNew;

                // If it isn’t on the open list, add it to

                // the open list. Make the current square

                // the parent of this square. Record the

                // f, g, and h costs of the square cell

                //                OR

                // If it is on the open list already, check

                // to see if this path to that square is better,

                // using 'f' cost as the measure.

**if** (cellDetails[i-1][j].f == FLT\_MAX ||

                        cellDetails[i-1][j].f > fNew)

                {

                    openList.insert( make\_pair(fNew,

                                               make\_pair(i-1, j)));

                    // Update the details of this cell

                    cellDetails[i-1][j].f = fNew;

                    cellDetails[i-1][j].g = gNew;

                    cellDetails[i-1][j].h = hNew;

                    cellDetails[i-1][j].parent\_i = i;

                    cellDetails[i-1][j].parent\_j = j;

                }

            }

        }

        //----------- 2nd Successor (South) ------------

        // Only process this cell if this is a valid one

**if** (isValid(i+1, j) == **true**)

        {

            // If the destination cell is the same as the

            // current successor

**if** (isDestination(i+1, j, dest) == **true**)

            {

                // Set the Parent of the destination cell

                cellDetails[i+1][j].parent\_i = i;

                cellDetails[i+1][j].parent\_j = j;

**printf**("The destination cell is found\n");

                tracePath(cellDetails, dest);

                foundDest = **true**;

**return**;

            }

            // If the successor is already on the closed

            // list or if it is blocked, then ignore it.

            // Else do the following

**else** **if** (closedList[i+1][j] == **false** &&

                     isUnBlocked(grid, i+1, j) == **true**)

            {

                gNew = cellDetails[i][j].g + 1.0;

                hNew = calculateHValue(i+1, j, dest);

                fNew = gNew + hNew;

                // If it isn’t on the open list, add it to

                // the open list. Make the current square

                // the parent of this square. Record the

                // f, g, and h costs of the square cell

                //                OR

                // If it is on the open list already, check

                // to see if this path to that square is better,

                // using 'f' cost as the measure.

**if** (cellDetails[i+1][j].f == FLT\_MAX ||

                        cellDetails[i+1][j].f > fNew)

                {

                    openList.insert( make\_pair (fNew, make\_pair (i+1, j)));

                    // Update the details of this cell

                    cellDetails[i+1][j].f = fNew;

                    cellDetails[i+1][j].g = gNew;

                    cellDetails[i+1][j].h = hNew;

                    cellDetails[i+1][j].parent\_i = i;

                    cellDetails[i+1][j].parent\_j = j;

                }

            }

        }

        //----------- 3rd Successor (East) ------------

        // Only process this cell if this is a valid one

**if** (isValid (i, j+1) == **true**)

        {

            // If the destination cell is the same as the

            // current successor

**if** (isDestination(i, j+1, dest) == **true**)

            {

                // Set the Parent of the destination cell

                cellDetails[i][j+1].parent\_i = i;

                cellDetails[i][j+1].parent\_j = j;

**printf**("The destination cell is found\n");

                tracePath(cellDetails, dest);

                foundDest = **true**;

**return**;

            }

            // If the successor is already on the closed

            // list or if it is blocked, then ignore it.

            // Else do the following

**else** **if** (closedList[i][j+1] == **false** &&

                     isUnBlocked (grid, i, j+1) == **true**)

            {

                gNew = cellDetails[i][j].g + 1.0;

                hNew = calculateHValue (i, j+1, dest);

                fNew = gNew + hNew;

                // If it isn’t on the open list, add it to

                // the open list. Make the current square

                // the parent of this square. Record the

                // f, g, and h costs of the square cell

                //                OR

                // If it is on the open list already, check

                // to see if this path to that square is better,

                // using 'f' cost as the measure.

**if** (cellDetails[i][j+1].f == FLT\_MAX ||

                        cellDetails[i][j+1].f > fNew)

                {

                    openList.insert( make\_pair(fNew,

                                        make\_pair (i, j+1)));

                    // Update the details of this cell

                    cellDetails[i][j+1].f = fNew;

                    cellDetails[i][j+1].g = gNew;

                    cellDetails[i][j+1].h = hNew;

                    cellDetails[i][j+1].parent\_i = i;

                    cellDetails[i][j+1].parent\_j = j;

                }

            }

        }

        //----------- 4th Successor (West) ------------

        // Only process this cell if this is a valid one

**if** (isValid(i, j-1) == **true**)

        {

            // If the destination cell is the same as the

            // current successor

**if** (isDestination(i, j-1, dest) == **true**)

            {

                // Set the Parent of the destination cell

                cellDetails[i][j-1].parent\_i = i;

                cellDetails[i][j-1].parent\_j = j;

**printf**("The destination cell is found\n");

                tracePath(cellDetails, dest);

                foundDest = **true**;

**return**;

            }

            // If the successor is already on the closed

            // list or if it is blocked, then ignore it.

            // Else do the following

**else** **if** (closedList[i][j-1] == **false** &&

                     isUnBlocked(grid, i, j-1) == **true**)

            {

                gNew = cellDetails[i][j].g + 1.0;

                hNew = calculateHValue(i, j-1, dest);

                fNew = gNew + hNew;

                // If it isn’t on the open list, add it to

                // the open list. Make the current square

                // the parent of this square. Record the

                // f, g, and h costs of the square cell

                //                OR

                // If it is on the open list already, check

                // to see if this path to that square is better,

                // using 'f' cost as the measure.

**if** (cellDetails[i][j-1].f == FLT\_MAX ||

                        cellDetails[i][j-1].f > fNew)

                {

                    openList.insert( make\_pair (fNew,

                                          make\_pair (i, j-1)));

                    // Update the details of this cell

                    cellDetails[i][j-1].f = fNew;

                    cellDetails[i][j-1].g = gNew;

                    cellDetails[i][j-1].h = hNew;

                    cellDetails[i][j-1].parent\_i = i;

                    cellDetails[i][j-1].parent\_j = j;

                }

            }

        }

        //----------- 5th Successor (North-East) ------------

        // Only process this cell if this is a valid one

**if** (isValid(i-1, j+1) == **true**)

        {

            // If the destination cell is the same as the

            // current successor

**if** (isDestination(i-1, j+1, dest) == **true**)

            {

                // Set the Parent of the destination cell

                cellDetails[i-1][j+1].parent\_i = i;

                cellDetails[i-1][j+1].parent\_j = j;

**printf** ("The destination cell is found\n");

                tracePath (cellDetails, dest);

                foundDest = **true**;

**return**;

            }

            // If the successor is already on the closed

            // list or if it is blocked, then ignore it.

            // Else do the following

**else** **if** (closedList[i-1][j+1] == **false** &&

                     isUnBlocked(grid, i-1, j+1) == **true**)

            {

                gNew = cellDetails[i][j].g + 1.414;

                hNew = calculateHValue(i-1, j+1, dest);

                fNew = gNew + hNew;

                // If it isn’t on the open list, add it to

                // the open list. Make the current square

                // the parent of this square. Record the

                // f, g, and h costs of the square cell

                //                OR

                // If it is on the open list already, check

                // to see if this path to that square is better,

                // using 'f' cost as the measure.

**if** (cellDetails[i-1][j+1].f == FLT\_MAX ||

                        cellDetails[i-1][j+1].f > fNew)

                {

                    openList.insert( make\_pair (fNew,

                                    make\_pair(i-1, j+1)));

                    // Update the details of this cell

                    cellDetails[i-1][j+1].f = fNew;

                    cellDetails[i-1][j+1].g = gNew;

                    cellDetails[i-1][j+1].h = hNew;

                    cellDetails[i-1][j+1].parent\_i = i;

                    cellDetails[i-1][j+1].parent\_j = j;

                }

            }

        }

        //----------- 6th Successor (North-West) ------------

        // Only process this cell if this is a valid one

**if** (isValid (i-1, j-1) == **true**)

        {

            // If the destination cell is the same as the

            // current successor

**if** (isDestination (i-1, j-1, dest) == **true**)

            {

                // Set the Parent of the destination cell

                cellDetails[i-1][j-1].parent\_i = i;

                cellDetails[i-1][j-1].parent\_j = j;

**printf** ("The destination cell is found\n");

                tracePath (cellDetails, dest);

                foundDest = **true**;

**return**;

            }

            // If the successor is already on the closed

            // list or if it is blocked, then ignore it.

            // Else do the following

**else** **if** (closedList[i-1][j-1] == **false** &&

                     isUnBlocked(grid, i-1, j-1) == **true**)

            {

                gNew = cellDetails[i][j].g + 1.414;

                hNew = calculateHValue(i-1, j-1, dest);

                fNew = gNew + hNew;

                // If it isn’t on the open list, add it to

                // the open list. Make the current square

                // the parent of this square. Record the

                // f, g, and h costs of the square cell

                //                OR

                // If it is on the open list already, check

                // to see if this path to that square is better,

                // using 'f' cost as the measure.

**if** (cellDetails[i-1][j-1].f == FLT\_MAX ||

                        cellDetails[i-1][j-1].f > fNew)

                {

                    openList.insert( make\_pair (fNew, make\_pair (i-1, j-1)));

                    // Update the details of this cell

                    cellDetails[i-1][j-1].f = fNew;

                    cellDetails[i-1][j-1].g = gNew;

                    cellDetails[i-1][j-1].h = hNew;

                    cellDetails[i-1][j-1].parent\_i = i;

                    cellDetails[i-1][j-1].parent\_j = j;

                }

            }

        }

        //----------- 7th Successor (South-East) ------------

        // Only process this cell if this is a valid one

**if** (isValid(i+1, j+1) == **true**)

        {

            // If the destination cell is the same as the

            // current successor

**if** (isDestination(i+1, j+1, dest) == **true**)

            {

                // Set the Parent of the destination cell

                cellDetails[i+1][j+1].parent\_i = i;

                cellDetails[i+1][j+1].parent\_j = j;

**printf** ("The destination cell is found\n");

                tracePath (cellDetails, dest);

                foundDest = **true**;

**return**;

            }

            // If the successor is already on the closed

            // list or if it is blocked, then ignore it.

            // Else do the following

**else** **if** (closedList[i+1][j+1] == **false** &&

                     isUnBlocked(grid, i+1, j+1) == **true**)

            {

                gNew = cellDetails[i][j].g + 1.414;

                hNew = calculateHValue(i+1, j+1, dest);

                fNew = gNew + hNew;

                // If it isn’t on the open list, add it to

                // the open list. Make the current square

                // the parent of this square. Record the

                // f, g, and h costs of the square cell

                //                OR

                // If it is on the open list already, check

                // to see if this path to that square is better,

                // using 'f' cost as the measure.

**if** (cellDetails[i+1][j+1].f == FLT\_MAX ||

                        cellDetails[i+1][j+1].f > fNew)

                {

                    openList.insert(make\_pair(fNew,

                                        make\_pair (i+1, j+1)));

                    // Update the details of this cell

                    cellDetails[i+1][j+1].f = fNew;

                    cellDetails[i+1][j+1].g = gNew;

                    cellDetails[i+1][j+1].h = hNew;

                    cellDetails[i+1][j+1].parent\_i = i;

                    cellDetails[i+1][j+1].parent\_j = j;

                }

            }

        }

        //----------- 8th Successor (South-West) ------------

        // Only process this cell if this is a valid one

**if** (isValid (i+1, j-1) == **true**)

        {

            // If the destination cell is the same as the

            // current successor

**if** (isDestination(i+1, j-1, dest) == **true**)

            {

                // Set the Parent of the destination cell

                cellDetails[i+1][j-1].parent\_i = i;

                cellDetails[i+1][j-1].parent\_j = j;

**printf**("The destination cell is found\n");

                tracePath(cellDetails, dest);

                foundDest = **true**;

**return**;

            }

            // If the successor is already on the closed

            // list or if it is blocked, then ignore it.

            // Else do the following

**else** **if** (closedList[i+1][j-1] == **false** &&

                     isUnBlocked(grid, i+1, j-1) == **true**)

            {

                gNew = cellDetails[i][j].g + 1.414;

                hNew = calculateHValue(i+1, j-1, dest);

                fNew = gNew + hNew;

                // If it isn’t on the open list, add it to

                // the open list. Make the current square

                // the parent of this square. Record the

                // f, g, and h costs of the square cell

                //                OR

                // If it is on the open list already, check

                // to see if this path to that square is better,

                // using 'f' cost as the measure.

**if** (cellDetails[i+1][j-1].f == FLT\_MAX ||

                        cellDetails[i+1][j-1].f > fNew)

                {

                    openList.insert(make\_pair(fNew,

                                        make\_pair(i+1, j-1)));

                    // Update the details of this cell

                    cellDetails[i+1][j-1].f = fNew;

                    cellDetails[i+1][j-1].g = gNew;

                    cellDetails[i+1][j-1].h = hNew;

                    cellDetails[i+1][j-1].parent\_i = i;

                    cellDetails[i+1][j-1].parent\_j = j;

                }

            }

        }

    }

    // When the destination cell is not found and the open

    // list is empty, then we conclude that we failed to

    // reach the destiantion cell. This may happen when the

    // there is no way to destination cell (due to blockages)

**if** (foundDest == **false**)

**printf**("Failed to find the Destination Cell\n");

**return**;

}

// Driver program to test above function

**int** main()

{

    /\* Description of the Grid-

     1--> The cell is not blocked

     0--> The cell is blocked    \*/

**int** grid[ROW][COL] =

    {

        { 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },

        { 1, 1, 1, 0, 1, 1, 1, 0, 1, 1 },

        { 1, 1, 1, 0, 1, 1, 0, 1, 0, 1 },

        { 0, 0, 1, 0, 1, 0, 0, 0, 0, 1 },

        { 1, 1, 1, 0, 1, 1, 1, 0, 1, 0 },

        { 1, 0, 1, 1, 1, 1, 0, 1, 0, 0 },

        { 1, 0, 0, 0, 0, 1, 0, 0, 0, 1 },

        { 1, 0, 1, 1, 1, 1, 0, 1, 1, 1 },

        { 1, 1, 1, 0, 0, 0, 1, 0, 0, 1 }

    };

    // Source is the left-most bottom-most corner

    Pair src = make\_pair(8, 0);

    // Destination is the left-most top-most corner

    Pair dest = make\_pair(0, 0);

    aStarSearch(grid, src, dest);

**return**(0);

}

**Time Complexity:**

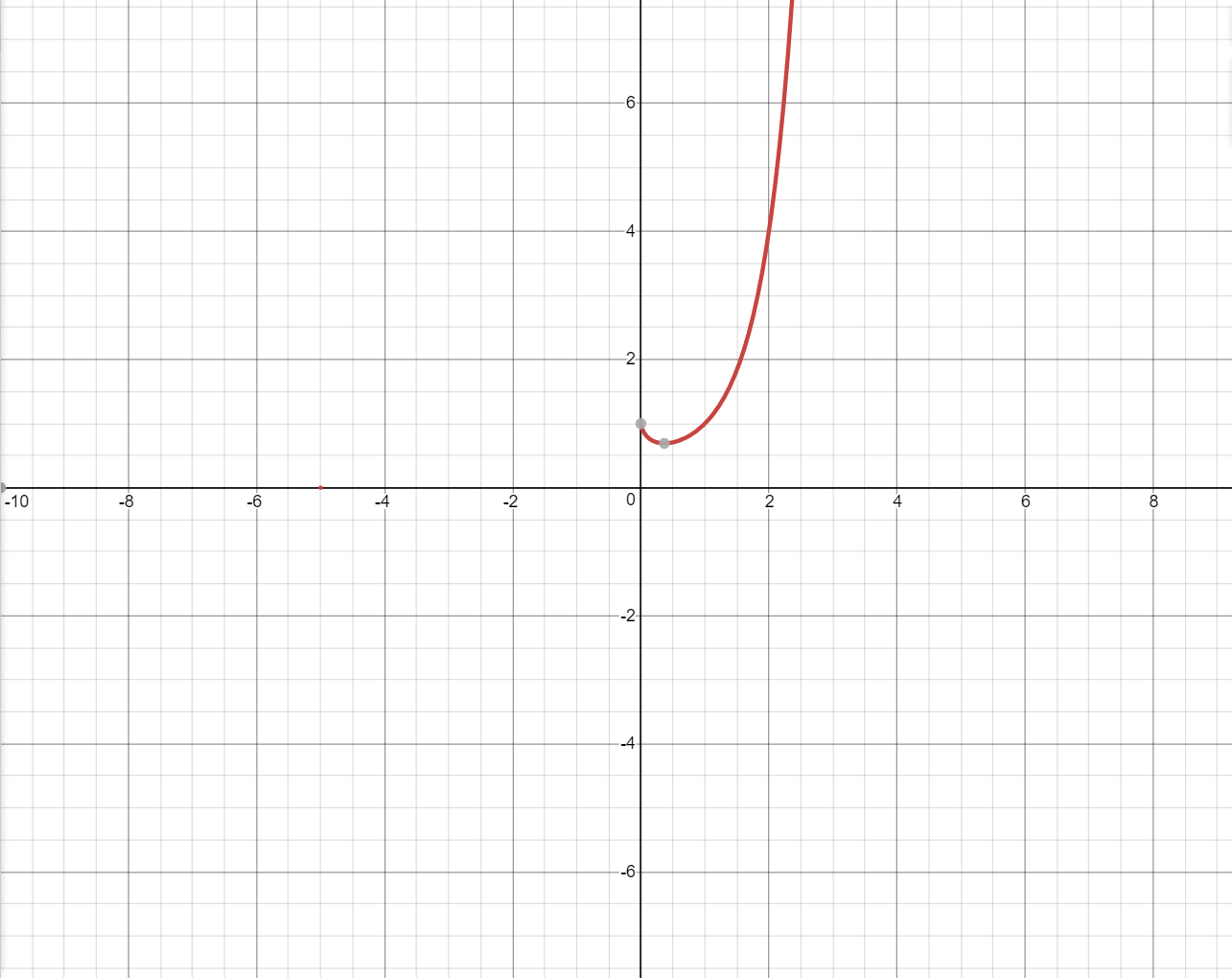
These are basically two different perspectives or two different ways of viewing the running time. Both are valid (neither is incorrect), but O(b^d) is arguably more useful in the settings that typically arise in AI.

In the algorithms community and CS theory community, folks there tend to like to count the running time as a function of the number of vertices and edges in the graph. Why? In that context, worst-case running time is what makes most sense; also, in the problems that are typically considered in that community, in the worst case we need to examine the entire graph, so you typically can't hope to do better than O(|V|+|E|).

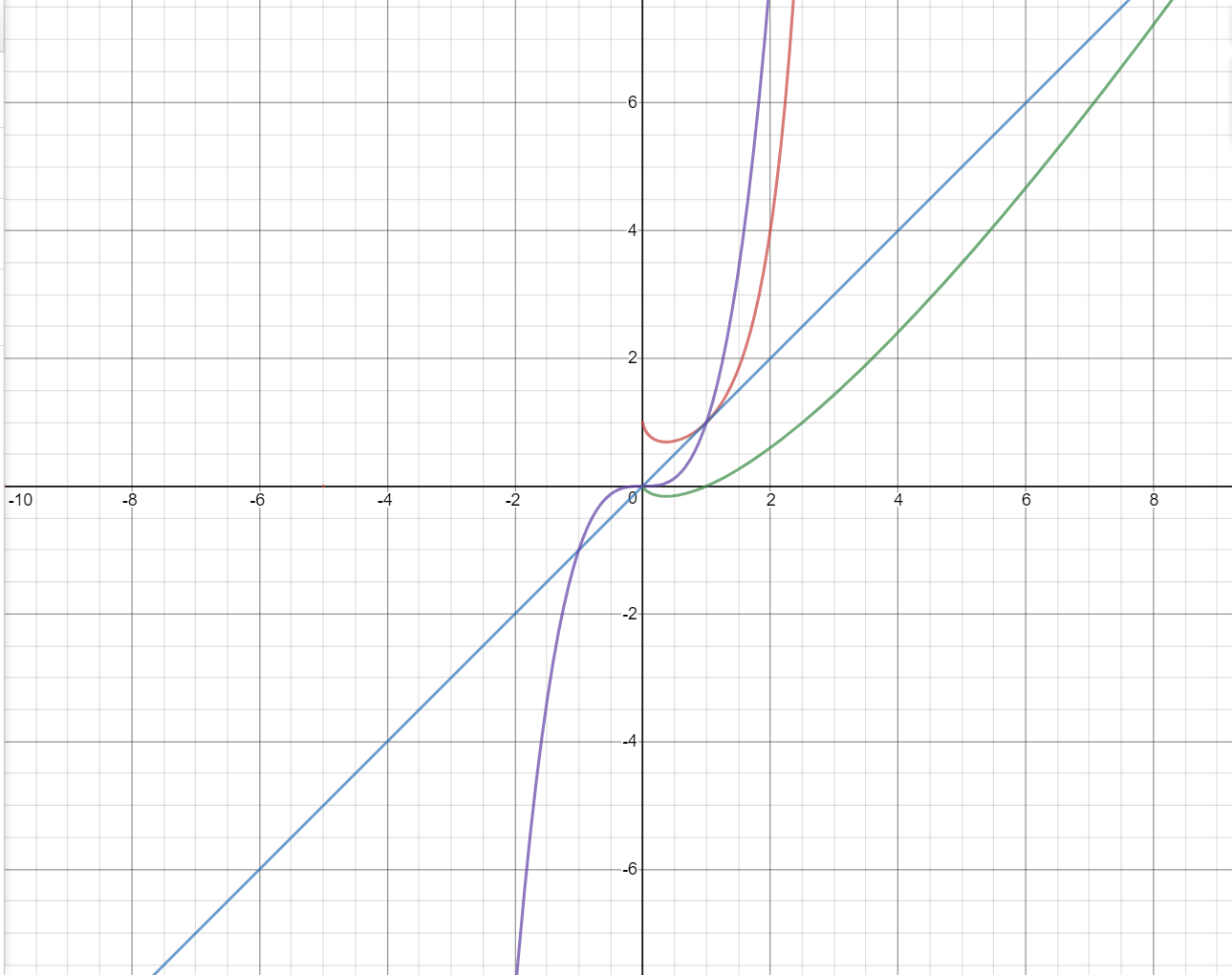
In the AI community, folks tend to count the running time differently. They often consider a specific kind of graph: a tree with branching factor b. Also, in the situations that arise there, the graph is often infinite or very large. Typically we try hard to avoid examining all of the graph -- that's often one of the major goals of the algorithms. Thus, counting complexity in terms of |V| and |E| doesn't make sense: |V|may be infinite, and in any case, we don't plan on examining all of the graph, so all that matters is the number of vertices we actually visit, not the number that may exist elsewhere but that we don't care about.

So, for the situations that often arise in the AI community, it's often more meaningful to measure the running time in terms of the branching factor of the tree (b) and the depth of the goal node (d). Typically, once we find the goal node, the algorithm stops. In such a tree, if we examine every vertex at depth ≤d≤d before we find the goal node, we'll end up visiting O(b^d) vertices before we stop. Thus, if you like, you could think of this as visiting a subset of the graph with |V|=O(b^d) (where now V includes only the vertices we visit) and |E|=O(bd) (E includes only the edges we look at), and you could think of an O(b^d)-time algorithm as one whose running time is O(|V|+|E|).. though this is a bit of an abuse of the |V|,|E| notation. Anyway, hopefully you can see why O(b^d) is more informative than O(|V|+|E|)in this context.

**Graph for time complexity:**



**Comparison of Time complexities of all Search algorithm**



**Graph in following colour represent the corresponding time complexity of following algorithms:**

1. **Purple- Floyd Warshall algorithm**
2. **Blue- BFS and DFS algorithm**
3. **Green- Dijkstra’s algorithm**
4. **Red- A\* algorithm**

**From the above graph we can figure out that the time complexity of Dijkstra’s algorithm is the lowest and most efficient from all other algorithms and therefore, we have used Dijkstra’s algorithm in our project (code).**

**REFRENCES:**

1. Research paper on:

**Indoor Positioning Based on Bluetooth Low-Energy Beacons Adopting Graph Optimization by-**

**Zheng Zuo 1, Liang Liu 2,\*, Lei Zhang 2 and Yong Fang 2**

1 College of Electronics and Information Engineering, Sichuan University, Chengdu 610064, China;

b110434@stu.scu.edu.cn

2 College of Cybersecurity, Sichuan University, Chengdu 610064, China; zhanglei2018@scu.edu.cn (L.Z.);

yfang@scu.edu.cn (Y.F.)

**\*** Correspondence: liangzhai118@scu.edu.cn

Received: 6 October 2018; Accepted: 30 October 2018; Published: 2 November 2018

1. Research Article on Improving Indoor Localization Using Bluetooth Low Energy Beacons by-

**Pavel Kriz, FilipMaly, and Tomas Kozel**

*Department of Informatics and Quantitative Methods, Faculty of Informatics and Management, University of Hradec Kralove,*

*Rokitanskeho 62, 500 03 Hradec Kralove, Czech Republic*

1. <https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_gui/py_video_display/py_video_display.html>
2. <https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_tutorials.html>
3. <https://docs.python.org/3/>
4. <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>
5. <https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>
6. <https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/>
7. <https://www.geeksforgeeks.org/a-search-algorithm/>
8. <https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/>