Technical Documenation

FH Aachen
Fouad Azar
October 2022
— Version 1

Abstract

Coding language is English, that is variables and documentation must be in English. All code must be documented. All code must be delivered executable. So please provide a command line call to invoke your solution and verify the result. Assume a *nix operating system. Code delivered without a command line call will not be considered a solution of the challenge. You must at least solve two of the three challenges and challenge two must be among them. Please solve the challenges yourself and provide only your original code.

In order to run the code, the following is required:

- MATLAB
- Docker and Docker-Compose
- Git
- Patience

To start, clone the GithHub repository with the URL:

```
git clone https://github.com/FouadAIAzar/interviewChallenge
```

Each challenge has a **quick start** section that you can follow to quickly test the code. Just follow along and it should work fine.

Contents

1 Challenge 1

1.1 Description

Assume a directed acyclic graph (DAG) in which nodes can have multiple parents and there is a single unique root node. The minimum number of edges connecting any node n with the DAG's root node r is considered the depth d (n) of node n . Because a node can have many parents, there might be several routes, of varying number of edges, from the node to the root r . Provide a test (see Test Driven Development) that executes your solution and verifies, it works as expected. Then write the function that returns the depth of a node given the DAG and test it with your test. Assume the DAG, i.e. the input to your function, to be represented as an adjacency matrix . Another input argument is the root node, i.e. consider the root node to be known so that your algorithm does not need to find it. Use any programming language you like to solve this challenge. You will receive extra points if you implement the solution in either Brainfuck or Assembler, but because you will be considered not totally sane we will not invite you to dinner.

1.2 Theory

DAG: A graph G is a finite nonempty set of objects, called vertices, together with a set of unordered pairs of distinct vertices, called edges. The set of vertices of a graph G is called the vertex set of G, denoted by V, and the set of edges is called the edge set of G, denoted by E. The edge $e \in E$ has a set $e = u, v$ and is said to join the vertices u and v.

Adjacency Matrix: An adjacency matrix A is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph.

A DAG G has an adjacency matrix A that is nilpotent. The properties of a nilpotent matrix are:

- $tr(A) = 0$
- $A^n = 0$
- $det(A + I) = 1$

Bellman-Ford Algorithm: Given a directed weighted graph $G = (V, E, w)$ and a source n, the Bellman-Ford algorithm returns the shortest path length from s to every vertex. Bellman–Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution. The approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value and the length of a newly found path.

The time complexity of Bellman-Ford algorithm has a best case scenario of ($|E|$) and worst case of ($|V||E|$)

1.3 Quick Start

Open MATLAB and `cd` to `.../interviewChallenge/c1/` and open up the `main.mlx`.

In the script press `F5` or type `main` in the terminal and wait for results.

1.4 `main`

All solutions to challenge 1 were written in `MATLAB`. No Add-Ons required. MATLABs LiveTex is similar to Jupyter Notebook, but runs much worse.

To start, `cd` to `..\interviewChallenge\c1` and open `main.mlx`. Run the script either by pressing `F5` in the script or by typing `main` in the MATLAB's native terminal.

! → If you executed main within the script, the results will be displayed inside of `main.mlx`. Otherwise, the figures will open up in a separate window and the integer values will dispalyed in the terminal.

1.4.1 User Inputs

`r:` r is used ambiguously, such that it serves to 1) define the number of nodes in a graph, i.e., $|V| = r$ and also serves as the root index, i.e., the end-point of the graph.

You can directly manipulate the value of r using the interactive `main.mlx` field-text-input.

Default value is 5.

`n:` A variable that serves to define the source node n. This number is bounded such that $n \in [1, r]$.

You can select the value of n using the interactive *mlx* slider-input.

The default value is 1.

1.4.2 Function Outputs

`weightedEdges:` A boolean variable that determines whether or not the graph will have weighted edges or not.

You can alter the variable by using the check-box in `main.mlx`.

`A:` An adjacency matrix with the dimension $[r \times r]$. This matrix is the output of the `randomDAGAdjecencyMatrix()`.

→ see 1.5

$$\begin{bmatrix} 0 & A_{12} & A_{13} & A_{14} & A_{15} \\ 0 & 0 & A_{23} & A_{24} & A_{25} \\ 0 & 0 & 0 & A_{34} & A_{35} \\ 0 & 0 & 0 & 0 & A_{45} \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

`G:` A MATLAB native variable known as a digraph. It's only purpose is to simplify figure presentation.

`E:` Is a matrix of all edges and their weights, i.e., $E = \{u, v, w\}$ of the graph `G`. `E` has the dimensions of $[e \times 3]$. e is randomly determined by `randomDAGAdjecnceyMatrix()`, and has an lower-bound of $e \geq r - 1$.

$$\begin{bmatrix} u_1 & v_1 & w_1 \\ u_2 & v_2 & w_2 \\ u_3 & v_3 & w_3 \end{bmatrix}$$

`d:` Represents the minimum cost connecting any node n with the DAG's root node r.

`path:` An array that has the index of all nodes traversed from n to r.

`allItrations:` A Table variable that shows how the depth vector is relaxed after each iteration of the Bellman-Ford algorithm.

`totalCost:` The total cost of the `path` between `n` to `r`.

1.5 `randomDAGAdjecencyMatrix()`

A function that generates a random DAG Adjecency Matrix. It takes two inputs `r` and `weightedEdges` and returns two outputs a graph `G` and an adjacency matrix `A`.

The function breaks the script when $r \leq 1$ or $r \geq 200$. The former has no path to calculate, since it is simply one node and the latter is just to prevent MATLAB from crashing.

Then it checks for variable `weightedEdges` and generates a random $[r \times r]$ matrix with values bounded either to either $\{0, 1\}$ in the case of `weightedEdges = true`, otherwise the values are bounded between $[-1, 1]$. Values are generated using MATLAB's `rand()` function.

The matrix is then transformed into a nilpotnet one. First, the the diagonal of the matrix is nullified, such that `tr(A) = 0`. This is done using `A - daig(diag(A))`.

Second, only the upper traingle of the square matrix is taken using MATLAB's native function `triu(A)`. The upper triangle is considered since we want the direction of the graph to flow, such that $\{\forall(u, v) \in E \mid u < v\}$, ensuring all nodes lead to the root r, Alle Wege fuehren nach Rom!

However, the aforementioned transformations led to an issue. Since the nature of generation is random, sometimes there were rows/ columns of zeros in a matrix. This issue is slightly annoying for the user, since they wish to have exactly r nodes in their graph, but received less than what they demanded. Hence a fix for this problem was to run MATLAB's `nnz()` on each row/column of the matrix, which checks to see which row has any non-zero elements. If it failed, the index of the iteration i and r are taken as the boundary to generate a random array index to store a new value. I know there's probably a better solution, but this will have to do for now.

The matrix `A` is then determined for nilpotency, which has the property: $det(A + I) = 1$, where I is an identity matrix. If the test fails, the script is terminated and error message appears.

Then a matrix `E` is generated, which a list of all starting and terminating nodes of an edge and their respective weights.

1.6 `dagPlotter()`

A function that renders two figures: a digrammatic representation of a graph `G` and a heatmap of an adjacency matrix `A`.

It takes three inputs `A`, `G` and `weightedEdges` and has no returns.

1.7 `bellmanFord()`

The function runs a Bellman-Ford algorithm and finds the depth of every node between the node n and root r.

The function takes 4 input variables – `n`, `G`, `E` and `r`, and returns for output variables – `d`, `allIterations`, `totalCost`, `path`.

The function runs the following pseudo-code:

```
function BellmanFord(list vertices, list edges, vertex source) is

// This implementation takes in a graph, represented as
// lists of vertices (represented as integers [0..n-1]) and edges,
// and fills two arrays (distance and predecessor) holding
// the shortest path from the source to each vertex

distance := list of size n
predecessor := list of size n

// Step 1: initialize graph
for each vertex v in vertices do

    distance[v] := inf
    predecessor[v] := null

distance[source] := 0

// Step 2: relax edges repeatedly

repeat |V|-1 times:
     for each edge (u, v) with weight w in edges do
         if distance[u] + w < distance[v] then
             distance[v] := distance[u] + w
             predecessor[v] := u

return distance, predecessor
```

Then predecessor array is used to trace the path from the root r to the source n.

and finally the results are plotted with the path traced over the graph.

1.8 `brainFuckInterpreter()`

Name ist Programm.

1.9 MATLAB Native-Functions

`clf:` Deletes all children of the current figure that have visible handles. This just ensures that every time the whole script is run again, the figures are reset.

1.10 Additional Files

`script.txt:` A text file containing Brainfuck code. To be interpreted by `brainFuckInterpreter()`

! $\rightarrow$ Do not change the name of `script.txt`

2 Challenge 2

2.1 Description

Write a simple Server in NodeJS using the Express Framework. The server should respond to two routes "$/foo$" and "$/bar$". "foo" should write the response "$Hello$" and "bar" should write the String "$World$" to the response in JSON format, serializing the objects { `response: "Hello"` } or { `response: "World"` } respectively. Make sure the response is exactly as defined here.

The important point in this exercise is to parametrize the server at startup depending on two environment variables `PORT` and `BASE_URL`. `PORT` indicates on which port the express server will listen, and `BASE_URL` defines which will be the base path inserted before the two above routes `foo` and `bar`. For example the shell command:

```
export PORT="3000"
export BASE_URL="/conabio"
npm run express_challenge_two.js
```

Should start a server that responds to `localhost:3000/conabio/foo` with { `response: "Hello"` }.

Your solution should include two Javascript files. The first `index.js` should create and start the Express server, the second `routes.js` shall use an Express Router `express.Router` in which the routes `foo` and `bar` are defined.

2.2 Quick Start

With Docker: Make sure your docker daemon is running and `cd` to `.../interviewChallenge/c2/` and in your bash terminal type `sudo docker-compose up --build` You should get a result that looks like

```
c2-server-1  |
c2-server-1  | > test@1.0.0 start
c2-server-1  | > node index.js
c2-server-1  |
c2-server-1  | PORT/BASE_URL: 3333
```

Using a web browser, navigate to `localhost:3333` There you should read a text that says:

```
Thank you for this oppertunity!
You're now listening to PORT 3333 with BASE_URL:
```

navigate to `localhost:3333/foo`, `localhost:3333/bar` or `localhost:3333/foobar` to find either:

```
{ response: "Hello" }
{ response: "World" }
{ response: "Hello World" }
```

, respectively.

Once you're done, press `CTRL+C` and run

```
sudo docker-compose down -v
```

to remove any danglers. To test for parameterization, add and execute the `change.sh` script:

```
chmod +x ./change.sh
./change.sh
```

Enter a new values for `PORT` and `BASE_URL`, and run

```
sudo docker-compose up
```

Afterwards, verify the results. Once you're done press `CTRL+C` and run:

```
sudo docker-compose down -v
```

2.3 Dockerfile

`FROM node:15`: The Dockerfile gets its image from Dockerhub's `node` image, in particular version 15.

`WORKDIR /app`: It creates a folder `/app` in the root of the container.

`COPY package.json .`: It initially copies only `package.json`, which improves the speed at which docker containers are re-built. The package is copied to the root directory.

`RUN npm install`: Runs the command `npm install` to fetch the node-package manager into the node container.

`COPY . ./`: Grabs the remaining files in the current folder to the root folder. The files that are copied also depend on the `.dockerignore` file as well.

`EXPOSE $PORT`: Honestly, I don't know why this is here. They say it's for documentation purposes, so, yeah, it's here, but I don't know why. It grabs the environment variable $PORT.

`CMD ["npm", "run", "start"]`: This runs the last command line in bash after starting the container. In this case, bash is going to run `npm run start` which is a script defined in `package.json`.

2.4 package.json

`scripts` There are 4 scripts defined:

```
"scripts": {
    "start": "node index.js",
    "change": "./entrypoint.sh & node index.js",
    "dev": "nodemon -L index.js",
    "stop": "kill $(cat .pid)"
  }
```

But, for the purposes of the end-user, please just use either `start`, or have fund with `dev`.

`dependencies`: The dependencies that will be loaded when installing npm into your container will be the following.

```
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "nodeman": "^1.1.2",
    "nodemon": "^2.0.20"
  }
```

2.5 docker-compose.yml

server: The docker-compose is set to version 3, however, it will also work with any other version. The name of the service will be `server` and uses the `node` image. The ports that connect the container to the host are defined in `./.env` as `PORT`. There are two volumes, one is the bind volume `./:/app` which keeps an interface between the current directory and the work directory `/app` open and the other is an anonymous volume `/app/node_modules`, which keeps the volume in the container detached from the current directory.

```
version: "3"
services:
  server:
    build: .
    ports:
      - "${PORT}:${PORT}"
    volumes:
      - ./:/app
      - /app/node_modules
    env_file:
    - ./.env
```

2.6 index.js

require: The main file for the server, `index.js` requires `express` which is stored as `const app`.

parameters: Two constants are required and obtained from the `.env` file, which are `PORT`, set to 3333 by default, and `BASE_URL`, set to a default is an empty string.

router: An `express.Router()` is defined under `routes.js` is stored as `const router`.

listen: The server will listen on `app.listen(port)`

2.7 routes.js

require: The router file for the server, `routes.js` requires `express.Router()` which is stored as `const router`.

routes: Three Routes are defined:

```
router.get("/foo", (req, res) =>{
    res.json({ response: "Hello" })
})

router.get("/bar", (req, res) =>{
    res.json({ response: "World" })
})

router.get("/foobar", (req, res) =>{
    res.json({ response: "Hello World" })
})
```

3 Challenge 3

3.1 Description

Use Docker and docker-compose to set up three containers:

- Database
- Server A
- Server B

The database container shall host a relational database, e.g. MySQL, Postgres, SQLite, etc., and reside inside the same network as Server A . Have a default database installed and running on the database server. Server B shall have no access to the database, but be able to connect to Server A . On startup Server A executes a simple shell-script that checks whether it can connect to the database and logs that to standard-out. Server B executes a similar script on startup and checks, e.g. using ping or nc, whether Server A is available, and logs the result to its standard-out. Consider using simple Debian or Alpine Linux images for A and B, and a standard relational database docker image for the database server. Write a simple Shell-Script that starts the network as described and prints out the respective outputs of the startup scripts.

You will receive extra points if you make Server B respond to requests from the host on port 3000, i.e. 'localhost:3000', with some simple greeting, e.g. "Hello World".

3.2 Quick Start

Make sure your docker daemon is running and `cd` to `.../interviewChallenge/c23/` and in your bash terminal type

```
sudo docker-compose -f docker-compose.yml
-f docker-compose-prod.yml up --build -d
```

Running the following:

```
sudo docker logs c3_server_1
sudo docker logs c3_server2_1
```

You should get a result that looks like

```
c3-server2-1  |
c3-server2-1  | > c3@1.0.0 start2
c3-server2-1  | > node server2.js
c3-server2-1  |
c3-server2-1  | PORT: 3001
c3-server-1   |
c3-server-1   | > c3@1.0.0 start
c3-server-1   | > node server.js
c3-server-1   |
c3-server-1   | PORT: 3000
c3-server-1   | succesfully connected to omics database
```

Using a web browser, navigate to `localhost:3000` There you should read a text that says:

```
I AM SERVER A
```

Navigate to `localhost:3000/omics`. There you should read:

```
{"status":"succes","results":0,"data":{"posts":[]}}
```

Navigate to `localhost:3001`. There you should read a text that says:

```
I AM SERVER B
```

Navigate to `localhost:3001/omics`. There you should read:

```
Cannot GET /omics
```

Unfortunately, I couldn't get server2 to automatically ping server, so this step is a failure on my part. Run:

```
sudo docker exec -it c3_server2_1 bash
```

To access server2's bash. In there, you can run the cmd:

```
ping c3_server_1
```

You should get a result that looks like:

```
64 bytes from c3-server-1.c3_default (172.22.0.4):
icmp_seq=1 ttl=64 time=0.113 ms
```

3.3 Dockerfile

→ 2.3

This `Dockerfile` is essentially the same as the previous one, with a few more additions.

`ARG NODE_ENV:` An argument passed from `docker-compose` that is a conditioner for the following `RUN` function

```
RUN if [ "$NODE _ENV" = "development" ]; \
    then npm install; \
    else npm install --only=production; \
    fi
```

If `NODE_ENV` is in "production", all dependencies, except devDependencies, are installed via `npm`. Otherwise, if "development", then install everything.

`RUN apt-get update` This is needed to install ping in the next step.

`RUN apt-get install iputils-ping -y` Installs ping for server-health test.

3.4 package.json

`scripts` There are 4 scripts defined:

```
"start": "node server.js",
"start2": "node server2.js",
"dev": "nodemon -L server.js",
"dev2": "nodemon -L server2.js"
```

`dependencies:` The dependencies that will be loaded when installing npm into your container will be the following.

```
"dependencies": {
  "cors": "^2.8.5",
  "express": "^4.18.2",
  "express-ping": "^1.4.0",
  "mongoose": "^6.6.5",
  "nodemon": "^2.0.20",
  "request": "^2.88.2"
}
```

3.5 docker-compose.yml

Docker compose will create three containers in using version 3. The containers are:

```
services:
    server:
        image: node
    server2:
        image:node
    omics:
        image: mongo
```

Along with these containers, it will establish three volumes:

```
    server:
        volumes
            - ./:/app
            - /app/node_modules
    omics:
        volumes:
            - omics-db:/data/db
```

3.6 server.js

require: The main file for the server, `index.js` requires `express` which is stored as `const app`, `mongoose` as `const mongoose`, `cors` as `const cors`.

parameters: In order to connect with `MongoDB`, a config file is imported:

```
MONGO_IP,
MONGO_PORT,
MONGO_USER,
MONGO_PASSWORD
```

which are all variables created and passed in from `docker-compose`. The `config` file can define a new parameter:

```
const mongoURL = `mongodb://${MONGO_USER}:${MONGO_PASSWORD}
@${MONGO_IP}:${MONGO_PORT}/?authSource=admin`;
```

tryAgain(): is a function that will continously try to connect to mondoDB and time-out after 5 seconds.

router: An `express.Router()` is defined under `postRoutes.js` is stored as `const router`.

listen: The server will listen on `app.listen(port)`

3.7 server2.js

→ 2.6

The model of this server is exactly the same as the model of `index.js`.

3.8 omicsModel.js

require: The `omicModel.js` requires `mongoose`.

schema: The input schema is as follows:

```
name: {
    type: String,
    require: [true, "Omic must have name"]
},
molecularWeight: {
    type: String,
    required: [true, "Omic must have molecularWeight"]
}
```

3.9 postController.js

require: The `omicModel.js` requires `../models/omicsModel.js`.

controllers: There are five controllers defined:

```
exports.getAllPosts()
exports.getOnePost()
exports.createPost()
exports.updatePost()
exports.deletePost()
```

They are a very simple CRUD, with built in error catchers.

3.10 postRoutes.js

require: The router file for the server, `routes.js` requires `express.Router()` which is stored as `const router` and controllers, which are imported from `../controllers/postController`.

routes: 6 Routes are defined:

```
router.route("/")
      .get(postController.getAllPosts)
      .post(postController.createPost)
      .purge()

router.route("/:id")
      .get(postController.getOnePost)
      .patch(postController.updatePost)
      .delete(postController.deletePost)
```