Project 3: Page Rank

The deadline for this project is **Monday, 10th of June**, end of day, anywhere on earth (AoE)[1]. You can find more information about the project submission in appendix A.

1 Overview

In this project, you will implement a C program that can analyze directed graphs and compute the *page rank* of its vertices. The directed graphs that we are interested in can be considered as consisting of websites and the hyperlinks connecting two websites.

Section 2 introduces the random surfer model as well as the Markov chain model which will both be used to determine the page rank. The functionalities your program needs to implement is documented in section 3. Section 4 indicates which implemented functionalities will score what amount of points. The appendix gives more technical information, a theory background on graphs, and examples on how to manually work with Markov chains. Furthermore, it answers several frequently asked questions.

2 PageRank

In the early days of search engines, PageRank was the distinguishing algorithm that effectively was key in making Google the giant it nowadays is, by giving it a perceivable advantage over other search engines. It was developed by the co-founders of Google, Sergey Brin and Larry Page[2]. PageRank assigns to every website a score, the page rank, that ranks its relevance among all search results. In the case of Google, the pages with the highest page rank would be the search results displayed to you.

In this project, we will first use the random surfer model to simulate and approximate the page ranks of the websites of a given network. Then, we will look at the mathematical foundations needed to compute the page rank.

2.1 Random Surfer

Intuitively, the page rank of a website is the probability that a *random surfer* will eventually end up on that website after surfing the web for an extended period of time. The term random surfer deserves some explanation: A random surfer is a fictional internet user whose normal behaviour is to look at the current website and to select a link to click at random, and then to follow that link to the next website. This normal behaviour is exhibited with probability $1 - p$ (for some number p between 0 and 1). With probability p, or if there are no links to follow, the random surfer gets bored, and instead moves on to an entirely random website. Lacking any other information, we can assume that all links on a website are equally likely to be clicked, and similarly for websites: The probability of the user moving to one of the N available websites after getting bored is $1/N$, and the probability of clicking one of the L available links on one website is $1/L$.

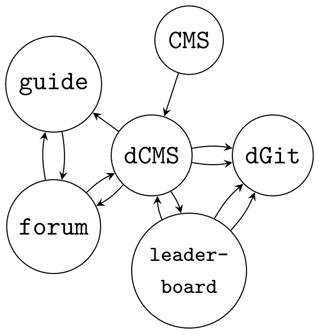


Figure 1: An Example Network made up of Websites and Hyperlinks between them

[1] https://en.wikipedia.org/wiki/Anywhere_on_Earth
[2] Sergey Brin, Lawrence Page, The anatomy of a large-scale hypertextual Web search engine, Computer Networks and ISDN Systems, Volume 30, Issues 1-7, 1998, pages 107-117, ISSN 0169-7552, https://doi.org/10.1016/S0169-7552(98)00110-X.

In the upcoming examples it may help to think of the probability of boredom as $p \approx 15\%$ and $(1-p) \approx 85\%$, but your program must handle all values of $p \in [0,1]$.

Figure 1 illustrates an example network with six pages and 12 links between the pages. A random surfer might visit the pages in the following order, starting in `dCMS`.

```
dCMS → guide → forum → dGit → dGit → leaderboard → dGit →
leaderboard → guide → dGit → dCMS → CMS → dGit → guide → forum
→ guide → dCMS → leaderboard → dCMS → dGit → dGit → dGit → CMS
→ dCMS → forum → leaderboard → dCMS → CMS → guide → forum →
guide → dGit → dGit → dGit → CMS → dCMS → forum → dGit → dGit →
leaderboard → CMS
```

Figure 2: The Browsing History of our Random Surfer after visiting 40 Pages

We observe that, starting in `dCMS`, the random surfer visits the installation `guide`, then the `forum`, then `dGit`, where they stay for a while, then go to the `leaderboard`, and so on. Notice that when visiting the `dGit`, the random surfer will instantly get bored, as there are no links to follow, and moves to one of the six available pages with equal probability of 1/6.

Page Rank Approximation

Before diving into the mathematical foundations of computing the page rank, we will first approximate the page rank of a site by using the browsing history h of a random surfer. In the previous example, our random surfer visited $|h| = 40$ websites (excluding the first), that is, they decided 40 times whether they get bored and move on, or whether they stay interested and click a random link. We can use these measurements to approximate the page rank pr of a website w as

$$pr_{\approx}(w) = \frac{\text{visitsTo}(w,h)}{|h|},$$

where visitsTo(w,h) is the function that counts how many times the site w was visited within the history h. Applying this definition to our example from earlier, we obtain the following counts and page ranks:

site w	`CMS`	`dCMS`	`dGit`	`leaderboard`	`guide`	`forum`
visitsTo(w,h)	4	7	13	5	6	5
$pr_{\approx}(w)$	$\frac{4}{40} = 0.1$	$\frac{7}{40} = 0.175$	$\frac{13}{40} = 0.325$	$\frac{5}{40} = 0.125$	$\frac{6}{40} = 0.15$	$\frac{5}{40} = 0.125$

Figure 3: Approximating the page rank using random surfer statistics

Your implementation will simulate a random surfer with parameter $p \in [0,1]$ for a duration of $|h|$ website visits and use the above formula to compute an approximated page rank $pr_{\approx}(w)$.

2.2 Markovian Approach

Although the random surfer can be used as an approximation for the page rank of a website, in reality it is much more precise to use what is called a Markov chain to compute a website's page rank.

Essentially, a Markov chain is a special kind of a graph[3]. The vertices of the graph are the states S the system can be in and the edges E are the transitions from one state to another. Every edge $s \to s' \in E$ is assigned a number $\mathbb{P}(s \to s')$, called edge weight, that determines how likely it is to go from state s to s'.

In our scenario, the Markov chain is to be understood as follows: The vertices S of the Graph are the individual websites and the edges E are the ways our random surfer can jump from one site to another. These jumps are, of course, echoeing the hyperlinks on the websites we consider. But notice that, even though there is no hyperlink between every pair of websites, since our random surfer can get bored, they can visit *any* website in the step with a positive probability. Thus, in our Markov Chains, there is an edge between any two websites. The edge weight $\mathbb{P}(s \to s')$ of each such edge can be considered as the probability jumping to state s' next, if currently in state s. By construction, our random surfer can never get stuck in a website without links (like

[3]If you need an introduction/refresher to/on graphs, see appendix C

`dGit` in the example), as that website will instantly make them bored. We can mathematically express that the probability of going from the current state s to any of its neighbor states s' is 1 as:

$$\sum_{s' \in S} \mathbb{P}(s \to s') = 1.$$

We will later enumerate the steps by an index, which for simplicity we refer to as *time*. Looking back at the example statistic, our time line started in state $s_0 = \texttt{dCMS}$ at timestamp 0 and our last observation of the random surfer was at timestamp 40 in state $s_{40} = \texttt{leaderboard}$. We could have simulated the random surfer for an arbitary amount of time, but decided stop after 40 steps.

To find out what the probability $\mathbb{P}(s \to s')$ is, for some states $s, s' \in S$, we need to look back at the network structure in fig. 1. From `dCMS`, it is slightly more likely to go to `dGit` over `leaderboard`, because there are two links from `dCMS` to `dGit`, but only one link to `leaderboard`. To express this observation, we will define $links(s, s')$ to be the number of links that lead within one step from s to s' in the network. Moreover, we will define the out-degree $out(s)$ of a website s to be number of links leaving the website, or formally:

$$out(s) = \sum_{s' \in S} links(s, s')$$

Using these definitions, we can then express the transition probabilities of our Markov chain as follows:

$$\mathbb{P}(s \to s') = \begin{cases} \frac{1}{|S|} & \text{if } out(s) = 0 \\ \underbrace{p \cdot \frac{1}{|S|}}_{\text{gets bored}} + \underbrace{(1-p) \cdot \frac{links(s,s')}{out(s)}}_{\text{stays interested}} & \text{otherwise} \end{cases}$$

We can use this Markov chain to study the probability of a random surfer being in each of the S states of our system at a given time point. This is will be a vector π of length $|S|$, where every component π_s corresponds to the probability of the random surfer being in state s. By fixing an ordering on the vertices such as in fig. 3, we can denote the fact that our random surfer is *certainly* in state `dCMS` as the vector $\pi = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$.

When simulating a Markov chain, your implementation must assume that the random surfer starts in any of the states $s \in S$ with *equal probability*, or formally

$$\pi_{initial} = \underbrace{\begin{pmatrix} \frac{1}{|S|} & \frac{1}{|S|} & \cdots & \frac{1}{|S|} & \frac{1}{|S|} \end{pmatrix}}_{|S| \text{ entries}}.$$

Similar to storing the probabilities of being in a state in a vector, we can collect all transition probabilities $\mathbb{P}(s \to s')$ in a matrix, which is a two-dimensional table of numeric entries. In order to store these probabilities in a matrix M, we need to assign a fixed order to all states $s \in S$, for instance the order in which we earlier computed $pr_{\approx}$. The entry m_{ij} of the of the i-th row and j-th colmn of the matrix M will contain the probability of going from state s_i to state s_j, e.g. $\mathbb{P}(s_i \to s_j)$.

For our example with $p = 0.15$, we obtain

$$M = \begin{pmatrix} \frac{1}{40} & \frac{7}{8} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} \\ \frac{1}{40} & \frac{1}{40} & \frac{73}{200} & \frac{39}{200} & \frac{39}{200} & \frac{39}{200} \\ \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{1}{40} & \frac{37}{120} & \frac{71}{120} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} \\ \frac{1}{40} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} & \frac{1}{40} & \frac{7}{8} \\ \frac{1}{40} & \frac{9}{20} & \frac{1}{40} & \frac{1}{40} & \frac{9}{20} & \frac{1}{40} \end{pmatrix}$$

Given that we know the probabilities π where our random surfer is *now*, we can find a nice formula to compute the probabilities where our random surfer will be in the next time step π': Let us consider the probability of arriving in state s' in the next time step, i.e. $\pi'_{s'}$. Currently, we may be in any of the states s, each with probability π_s. To arrive at s' after one time step, we must have followed the edge $s \to s'$ of the Markov chain, which happens with probability $\mathbb{P}(s \to s')$. To find the probability of being in s' in the next time step, we must add up the probabilities of all ways of getting there, multiplied with the probabilities that we are in state s right before the step, which is π_s:

$$\pi'_{s'} = \sum_{s \in S} \pi_s \cdot \mathbb{P}(s \to s')$$

Recalling that we order (and hence enumerate) the states $S = \{s_1, \dots, s_{|S|}\}$, we can consider s' as some s_k for some number k allowing us to write π'_{s_k} as

$$\pi'_{s_k} = \sum_{i=1}^{|S|} \pi_{s_i} \cdot \mathbb{P}(s_i \to s_k),$$

which is by definition of the matrix' entries m_{ij} the inner product

$$= \sum_{i=1}^{|S|} \pi_{s_i} \cdot m_{ik}$$

If you find this equation hard to grasp, consult appendix C for more examples.

Given a matrix M, computing the entries π'_s for all states $s \in S$ produces a new probability vector π', whose i-th entry is the probability of being in the state s_i. In mathematics, this operation is called *matrix multiplication* and is so commonly used that it deserved its own notation

$$\pi' = \pi \cdot M,$$

which you should read as: “in order to obtain the probabilities π', we must matrix-multiply the vector π times M”, for which you can use the above formulas.

Page Rank Approximation

We have now seen how we can use matrix multiplication to compute the state probabilities π' in the next time step, given the state probabilities at the current time step π. By repeating this procedure, we can simulate the Markov chain starting with probabilities $\pi_{initial}$ at some future point in time i:

$$\begin{aligned} \pi_0 &= \pi_{initial} \\ \pi_{i+1} &= \pi_i \cdot M \end{aligned}$$

By repeating this procedure many times, the vector π_N converges towards the (so-called) stationary distribution of the Markov Chain, whose entries are the associated page ranks.[4] For example, repeatedly matrix-multiplying our “we start in `CMS`” vector with the transition matrix M, we obtain

$$\begin{array}{rcccccc}
 & \texttt{CMS} & \texttt{dCMS} & \texttt{dGit} & \texttt{leaderboard} & \texttt{guide} & \texttt{forum} \\
\pi_0 \approx & (0.0000 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000) \\
\pi_1 \approx & (0.0250 & 0.0250 & 0.3650 & 0.1950 & 0.1950 & 0.1950) \\
\vdots & & & & & & \\
\pi_{20} \approx & (0.0502 & 0.2274 & 0.1779 & 0.0889 & 0.1982 & 0.2574) \\
\pi_{21} \approx & (0.0502 & 0.2274 & 0.1779 & 0.0889 & 0.1982 & 0.2574)
\end{array}$$

and observe that the values slowly approach a fixed value – they arrive at the stationary distribution. Within 20 iterations of matrix multiplication, we have computed each page’s page rank, correct up to four decimal places.

[4]The term “stationary” characterises that “nothing changes if doing one more step”.

3 Program Specification

Your task in this project will be to implement the random surfer method as well as the matrix multiplication method to compute the page rank values in C.

Instead of providing empty function implementations for you to fill, this section will specify the expected behavior of your program. You are invited to use standard libraries, such as `stdio.h`, `stdlib.h`, `string.h` or `getopt.h`, but you may not use external libraries. Use the latest C language standard (C11). Your submission may be structured into several `.c` and `.h` files in the `src/` directory, and executing `make` must generate an executable binary named `pagerank`.

3.1 Command Line Parameters

Your program must understand and correctly interprete the following command line options:

```
$ ./pagerank -h
Usage: ./pagerank [OPTIONS] ... [FILENAME]
Perform pagerank computations for a given file in the DOT format

  -h    Print a brief overview of the available command line parameters
  -r N  Simulate N steps of the random surfer and output the result
  -m N  Simulate N steps of the Markov chain and output the result
  -s    Compute and print the statistics of the graph as defined
          in section 3.4
  -p P  Set the parameter p to P%. (Default: P = 10)
```

Correct usage of your program assumes that

- `N` is an integer larger than 0
- `P` is an integer between 1 and 100 (inclusive).
- at least one of the optional parameters `-h`, `-r`, `-m` and `-s` is used at a time
- if `-h` is not specified, an extra argument `FILENAME` is expected. It contains the relative or absolute path to the input file

Incorrect usage of your program only needs to be handled for the bonus task (see section 4).

Please note that `(Default: P = 10)` describes the parameter `-p P` to be optional, and if it is not provided, defaults to $p = 0.1$. Moreover, what happens if multiple of the flags `-r`, `-m` and `-s` are provided, is not defined.

3.2 Input format

The graphs your program needs to work with are restricted to a *subset* of the DOT graph language[5]. We will define the minimal format your program needs to understand below.

```
digraph Prog2Graph {
CMS -> dCMS;
dCMS -> dGit;
dCMS -> dGit;
dCMS -> guide;
dCMS -> forum;
dCMS -> leaderboard;
leaderboard -> dCMS;
leaderboard -> dGit;
leaderboard -> dGit;
guide -> forum;
forum -> guide;
forum -> dCMS;
}
```

Every file must start with the keyword `digraph` (for directed graph), followed by a space, an identifier for the graph, a space, and an opening brace. For every edge of the grah, one line follows, containing the identifier of the state where the edge originates, followed by a space, an arrow made up of a minus and a greater-than sign, a space, the identifier of the target state, and a semicolon. Every line is terminated with a newline character (`\n`). Identifiers start with a letter from the latin alphabet, followed by letters of the latin alphabet or numbers. You must support identifier lengths up to 256 characters.

[5] `https://graphviz.org/`

It is sufficient for your program to *only* understand this exact representation, you may however, for example, allow more liberal placement of whitespace.

There are many available tools that let you vizualize graphs in the DOT format you may want to use[6].

3.3 Output format

After approximating the page ranks, your program should print the computed values to the standard output. Per state of the graph, print a line containing the identifier of the state, at least one space or tabulator character, and the computed page rank up to a precision of at least 10 decimals.

The order in which you output the states does not matter. Computing the page rank for the `Prog2Graph` from the input format example (with $p = 0.15$ and $N = 100$, using the Markov approach) may output the following:

```
$ ./pagerank -m 100 -p 15 prog2graph.dot
CMS         0.0502000920
dCMS        0.2274320656
dGit        0.1778830020
guide       0.1982474347
forum       0.2573738626
leaderboard 0.0888635431
```

Note: In order to achieve a high enough precision, we recommend using the floating point type `double`.

3.4 Statistics

If instructed to print statistics instead of computing the page rank, your program should print the following information about the input graph.

(a) The identifier of the graph

(b) The number of vertices of the graph

(c) The number of edges of the graph

(d) The minimal and maximal in-degree among all vertices of the graph

(e) The minimal and maximal out-degree among all vertices of the graph

So far, we only defined the out-degree, but observe how the in-degree $in(s)$ of a state s is defined analogously:

$$out(s) = \sum_{s' \in S} links(s, s')$$
$$in(s) = \sum_{s' \in S} links(s', s)$$

The following execution of the tool using our example graph defines the exact output format to be used by your implementation:

```
$ ./pagerank -s prog2graph.dot
Prog2Graph:
- num nodes: 6
- num edges: 12
- indegree: 0-4
- outdegree: 0-5
```

[6]For example `https://edotor.net/`

4 Assignments

Though we do not instruct you how to implement the required functionality, we provide some hints among the assignments relevant for grading.

Moreover, this project might be your first medium-sized programming project. We recommend solving the following assignments in order, as they slowly guide you through the challenges of this project.

Lastly, to better assess the complexity of the project, we give you several *optional* milestones, at which you should have finished implementing individual parts of the project.

Assignment 1: Help Page (1 Points)

Implement the commandline flag `-h`.

We recommend you use the `getopt` standard library function to implement parsing of command line parameters. Although the related manual page[7] and code in `argparser.c` in project 2 provide some examples on how to use `getopt`, the section in the `libc` manual[8] gives particularly useful ones.

Assignment 2: Statistics (5 Points)

Implement the command line flag `-s`.

To compute the statistics, you should implement parsing of the input graph and storing the parsed graph in a data structure. Once you have constructed a graph representation in memory, computing the statistics is rather simple.

First, decide how you want to represent graphs in memory. Then, parse the input files into your graph data structure. Dynamic memory management will prove to be very useful. Observe which kind of information becomes available at which time during the parsing of the input file. To create a simple parser, you can use the `fscanf`[9] function.

– Optional Milestone α: Solve all above exercises until June 02 –

Assignment 3: Random Surfer (7 Points)

Implement the command line flag `-r`. Make sure it respects the optional flag `-p`. Recall that simulating the random surfer for N steps means picking a random starting state and executing N steps, where you track the visited states. The initial state should not be counted.

Make sure that your random surfer shows random behavior and visits different vertices upon every start of your program. For your implementation it will prove useful to randomly generate numbers in the interval $[0, N-1]$ with equal probabilities of $1/N$. The pre-defined functions in `utils.c` will be helpful.

– Optional Milestone ω: Solve all above exercises until June 06 –

Assignment 4: Markov Chain (5 Points)

Implement the command line flag `-m`. Make sure it respects the optional flag `-p`. Recall that simulating N steps of the Markov chain means starting with an initial vector that is equally likely in every state, to which you multiply the transition matrix n times. It does not matter if you actually perform matrix multiplication in memory or perform the computation based on a graph. Clearly, the final result should be the same.

Assignment 5: Error Detection (2 Bonus Points)

The signature of the `int main();` states that the main function should return an integer. This integer is commonly referred to as status code. If the program exists normally, 0 should be returned. However, if the program ran into an error, it should exit with exit code 1. Error handling in this project is optional and you may assume that for all non-bonus assignments, error handling is voluntary.

Implement error handling. Possible errors include:

- Invalid command line arguments are provided
- The input file does not exist
- The input file has an invalid format

[7] `https://man7.org/linux/man-pages/man3/getopt.3.html`
[8] `https://www.gnu.org/software/libc/manual/html_node/Using-Getopt.html`
[9] `https://man7.org/linux/man-pages/man3/fscanf.3p.html`

A Infrastructure, How to pass, Assorted Hints

Using dGit

Use git to clone your personal project repository from dGit, for example by entering

```
git clone ssh://git@dgit.cs.uni-saarland.de:2222/prog2/2024/students/project-3-{your matriculation number}
```

Submit your project using `git push` until Monday, 10th of June, end of day, anywhere on earth (AoE). If you need a refresher on how to use git, consult the materials from the Programming 2 Pre-Course or additional material online[10].

Grading

Similar to the way you obtain your project, you will submit the project through git. The last commit pushed to our server before the deadline will be considered for grading. Please ensure that the last commit actually represents the version of your project that you wish to hand in.

Once you `git push`ed your project to our server, we will automatically grade your implementation on our server and give you feedback about its correctness. You can view your latest test results by following the link to the Prog2 Leaderboard in dCMS Personal Status page. We use three types of tests: **public** tests, **regular** tests and **eval** tests. The public tests are available to you from the beginning of the project in your project repository. You should use them to test your implementation locally on your system. Once you *pass a sufficient number of public tests*, we will run the regular tests for your implementation, disclosing to you the names of the regular tests and whether or not you successfully pass them.

Eval tests are similar to regular tests, but run only once after the project deadline. They directly determine the number of points your implementation scored.

Notes

(a) Attempts at tampering with our grading system, trying to deny service to, purposefully crash, or extract grading related data from it will (if detected) result in your immediate expulsion from the lecture and a notification to the examination board. In the past, similar attempts have led to us filing criminal complaints with the police.

(b) Ensure that your program frees dynamically allocated memory after use and is free of undefined behavior.

(c) Use double precision floating points (double) for your page rank computations.

(d) Test your implementation with varying values for p and different scenarios of graphs. Do your tests consider edge cases such as disconnected graphs, sources, sinks and self-loops?

[10] Such as `https://git-scm.com/book/en/v2`

B Building, Running and Testing

Compiling the Project

You may build the project by executing the provided `Makefile`, e.g. by running `make` in your WSL, or terminal. Upon successful build, a binary executable `pagerank` will be created in your project's root directory. Alternatively, you may compile and link the files in the `src/` directory manually.

Testing

The provided make file lets you compile your program and execute the tests by running `make tests` from the root directory of your project, upon which the provided python script `tests/run-tests.py` is launched. Each test consists of a python script located in the `tests/pub/` directory and typically refers to one of the graphs inside `tests/graphs/`. The `run-tests.py` script executes your `pagerank` binary in the given test scenario and compares the values printed by your implementation against the expected values.

To debug individual tests, you can list the available tests with

```
$ python3 tests/run-tests.py -l
pub.invoke-help
pub.markov-1-empty
pub.markov-1-simple
pub.random-1-empty
pub.stats-empty
pub.stats-prog2graph
pub.stats-simple
```

and specify the tests to execute with the filter flag (`-f`) before passing the path to your executable like so

```
$ python3 tests/run-tests.py -f pub.markov-1-simple ./pagerank -d
```

The debug flag (`-d`) gives more verbose output regarding the expected output of your program.

Note: `make tests` will compile your project, if necessary, before testing. When running tests using `run-tests.py`, you must manually recompile your project before testing.

C Theory Corner

C.1 Graphs

Graphs are an important concept in computer science, but you may have escaped their definition or terminology for some reason or another.

Common Definition Graphs come in numerous varieties and flavors, but besides the general definition, we will focus on the types of graphs you need to understand for this project. An (unweighted, directed) graph is formally defined as a tuple $G = (V, E)$ consisting of the *vertices* and *edges* of the graph. "Vertex" is one of the names we use (depending on the context) for the graph's nodes, states, or places. The vertices play the role of the distinct locations inside the graph where something can be. "Edge" is one of the names we use for the graph's links, connections, or arrows. Generally, edges connect vertices of a graph.

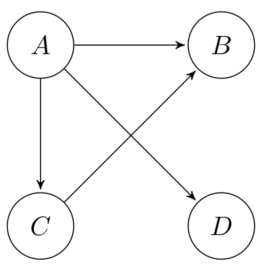
Typically, V is the *set* of vertices and $E \subseteq V \times V$ is the set of edges. An edge $(s,t) \in E$ thus consists of a starting vertex s and a target vertex t, often written $s \to t$. When the direction of the arrow in the graph does not matter, edges are not defined as tuples of starting and ending vertex (s,t), but rather as sets of starting and ending vertex s, t. Such graphs are called undirected graphs.

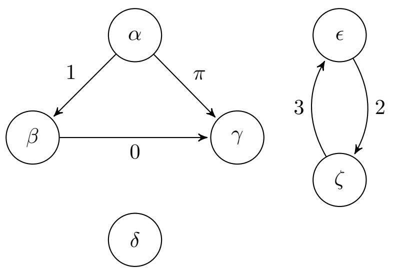
Some graphs, for example *weighted graphs* may consist of more components. In a weighted graph $G_w = (V, E, w)$, we include a weight function, that assigns every edge $e \in E$ a number $w(e) \in \mathbb{R}$. Weighted graphs play an important role in graphs representing road maps, where one, for example, is interested in shortest paths between cities.

Multigraphs The network graphs serving as inputs to our `pagerank` program slightly deviate from the standard definition of directed, unweighted graphs. Since one webpage may have multiple links to another, there may exist duplicate edges in the same direction between two vertices. We could either interprete the graphs as multigraphs, whose edges E are not sets, but multisets of edges. Mutisets are essentially sets, into which you can insert the same element multiple times, and the multiset keeps track of how many times it is contained. To tell the notation of multisets apart from that of sets, one sometimes writes $E = \{\{\texttt{dCMS} \to \texttt{guide}, \texttt{dCMS} \to \texttt{dGit}, \texttt{dCMS} \to \texttt{dGit}, \ldots\}\}$

Alternatively, one can understand mutigraphs as regular (directed or undirected) *weighted* graphs, where the weight function w counts how many times an edge $e \in E$ is duplicated.

Example We'd like to introduce and exercise some more generally helpful terminology on the examples of the following two graphs.





Formally the left graph consists of $G_L = (V_L, E_L)$, where $V_L = \{A, B, C, D\}$ and $E_L = \{(A,B), (A,C), (A,D), (C,B)\}$, which is the set theoretic notation for the fact that $E_L = \{A \to B, A \to C, A \to D, C \to B\}$.

The right graph can be defined as the weighted graph $G_R = (V_R, E_R, w_R)$ where $V_R = \{\alpha, \beta, \gamma, \delta, \epsilon, \zeta\}$ with $E_R = \{\alpha \to \beta, \alpha \to \gamma, \beta \to \gamma, \epsilon \to \zeta, \zeta \to \epsilon\}$ and $w = \{(\alpha,\beta) \mapsto 1, (\beta,\gamma) \mapsto 0, (\alpha,\gamma) \mapsto \pi, (\epsilon,\zeta) \mapsto 2, (\zeta,\epsilon) \mapsto 3\}$. Sometimes we wish to switch between the notations for edges, e.g. $w((\alpha,\beta)) = w(\alpha \to \beta) = 1$.

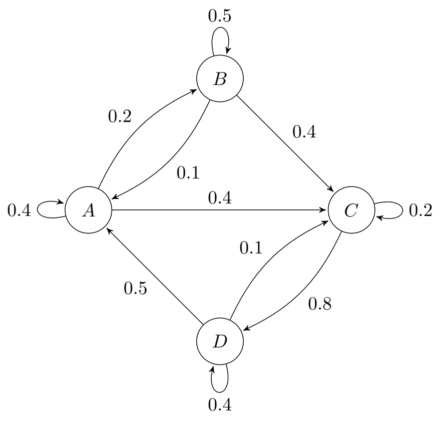
Terminology A *path* is a sequence of vertices (A, C, B), where every subsequent pair of vertices shares an edge between them e.g. $A \to C \to B$. We call B *reachable* from A, since there exists (at least) one path from A to B. If we ignore the direction of edges, and every vertex can be reached from every other vertex we call the graph *connected.* A graph that is not connected, like G_R, is called *disconnected.* Once we leave A, there is no edge back into A, which makes A a *source* vertex. Once we arrive at D, there is no edge away from D, which makes D a *sink* vertex. The node δ is both a source and sink, and as such called an *isolated* vertex. We call the path $\epsilon \to \delta \to \epsilon$ a cycle, as we can walk the same path over and over again starting from ϵ.

C.2 Markov Chains

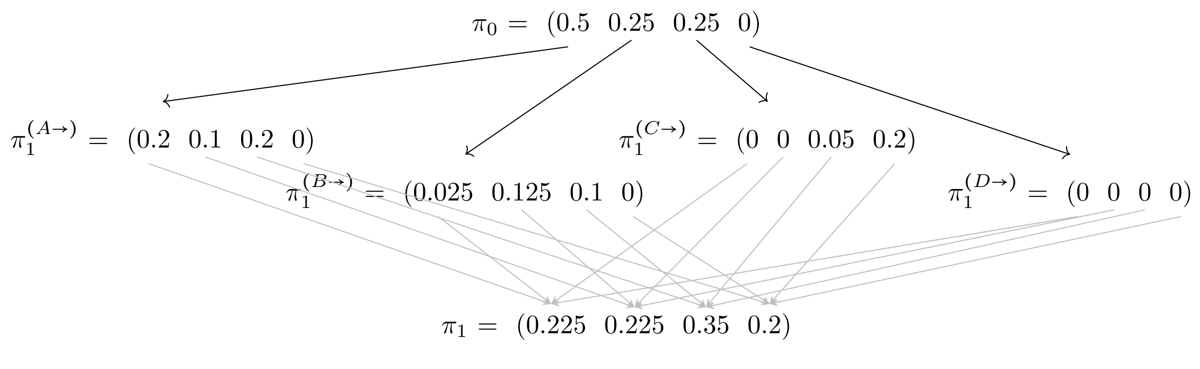
Definition With our background in graph theory, we can briefly define a (finite, discrete-time) Markov chain C as a graph $C = (V, E, \mathbb{P}, \pi_0)$, where V is a finite set containing the vertices of the graph, E comprises edges, $\mathbb{P} : E \to \mathbb{R}^{\geq 0}$ is the edge probability function and $\pi_0 : \underbrace{\mathbb{R}^{\geq 0} \times \cdots \times \mathbb{R}^{\geq 0}}_{|V| \text{ times}}$ is the initial probability distribution over the vertices.

$(V, E, \mathbb{P})$ form a directed, weighted graph. In addition, we require for every vertex v, that the weights of all outgoing edges sums up to 1. Similarly, we require that the entries of π_0 are all non-negative and sum up to one.

Interpretation and Example Let us consider above example Markov chain $C = (V, E, \mathbb{P}, \pi_0)$ where $V = \{A, B, C, D\}$, E and $\mathbb{P}$ as depicted and $\pi_0 = \begin{pmatrix} 0.5 & 0.25 & 0.25 & 0 \end{pmatrix}$. The vector π_0 indicates that we start our journey inside the Markov chain with 50% chance in A, and with 25% chance in B or C.



Starting with π_0 in time step zero, we can explicitly compute the probability of arriving in a certain state $v \in V$ at a future time step i. To find out, where the initial "probability from A goes" after the first time step, we can simply simulate where someone in A will probably go, according to the probabilities $\mathbb{P}(A \to \cdots)$. With $\mathbb{P}(A \to A) = 40\%$ chance, one stays in A, with a $\mathbb{P}(A \to B) = 20\%$ chance, one goes to B, and with a $\mathbb{P}(A \to C) = 40\%$ chance, one goes to C. However, since we now fixed the starting state to be A, we should weigh the outcoming probabilities according to the fact that we only start in A with a chance of 50%. Thus, we get the resulting probabilities of arriving in states A through D in time step 1 as $\pi_1^{(A\to)} = \begin{pmatrix} 0.4 \cdot 0.5 = 0.2 & 0.2 \cdot 0.5 = 0.1 & 0.4 \cdot 0.5 = 0.2 & 0 \cdot 0.5 = 0 \end{pmatrix}$.



$\pi_0 = (0.5\ \ 0.25\ \ 0.25\ \ 0)$
$\pi_1^{(A\to)} = (0.2\ \ 0.1\ \ 0.2\ \ 0)$
$\pi_1^{(C\to)} = (0\ \ 0\ \ 0.05\ \ 0.2)$
$\pi_1^{(B\to)} = (0.025\ \ 0.125\ \ 0.1\ \ 0)$
$\pi_1^{(D\to)} = (0\ \ 0\ \ 0\ \ 0)$
$\pi_1 = (0.225\ \ 0.225\ \ 0.35\ \ 0.2)$

After computing the intermediate results $\pi_1^{B\to}$, $\pi_1^{C\to}$ and $\pi_1^{D\to}$, we can compute the probabilities of being in a certain state at the next time step π_1, irrespective of where we started. To find the probability of being in state A at time step 1, we simply need to add all probabilities that can lead us to A in time step 1, irrespective from where we are coming. Thus, $\pi_1(A) = \pi_1^{(A\to)} + \pi_1^{(B\to)} + \pi_1^{(C\to)} + \pi_1^{(D\to)}$. Notice how the values of π_1 sum up to one, just like in π_0, showing that no probability "got lost". Continuing this process for all possible arrival states, we arrive at $\pi_1 = \begin{pmatrix} 0.225 & 0.225 & 0.35 & 0.2 \end{pmatrix}$.

Alternatively, you may derive the identical formulae by asking the question "How can I arrive at state A in the next time step" instead of "Where can I get from state A in the next time step". If you write the probabilities $\mathbb{P}(v_i \to v_j)$ into a table, or matrix, you will find that the computations done manually here follow a particularly regular pattern, known as matrix-multiplication.