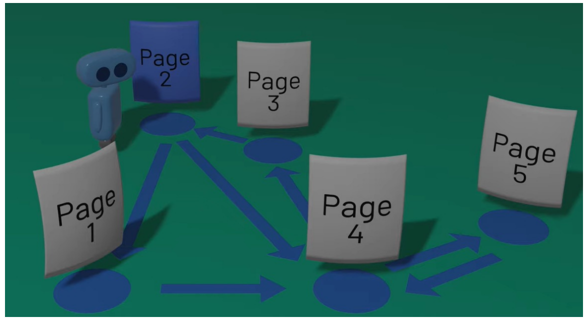
Intro-To-CS

Rec-4: Page Rank

TEL AVIV UNIVERSITY

PageRank

- How did people ***surf*** the web?
    - **Go** to a page by a **known** link.
    - **Follow links** within the page.
    - **Repeat**.
- A need to **rank** the pages' importance.
    - E.g., as part of a **search engine**.
- Page Rank (1996)
    - By **Larry Page** and **Sergey Brin**, Stanford, 1996.
    - Followed by the foundation of **Google**.
    - **Ranks** each **webpage** via an **importance score**.



Internet → Network Graph

- Can model the internet in a graph.
- Each node, a: a webpage.
- Each edge $a \rightarrow b$: a link from web a to web b.

PageRank – Algorithm

- **Input:** Network G, time t, damping factor $0 < p < 1$
- **Output:** **Weights** (ranks) of pages in G

- **Algorithm:**
  - Initialize the current node ($curr$) to 0
  - Initialize a $counter$ for *each* page in G
  - For t times:
    - With probability p: $curr$ = random **link** *from* $curr$
    - Otherwise: $curr$ = random **page** in G
    - Increase the counter of curr by 1
  - Return the $\frac{Page's\ counter}{t}$ for each page in G

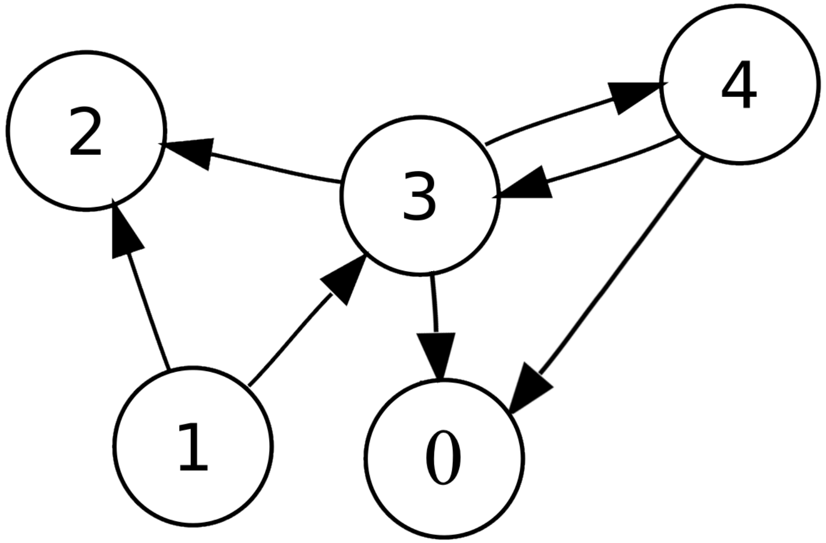
Python implementation

- Need to represent the graph as a **Python object**.
- Formally, the network is made up of two objects:
  - A set of $\boldsymbol{n}$ pages (which we call $0, \dots, n-1$).
  - A set of links of from one page to another.
    - (i, j) with $0 \leq i, j < n$ means that page i has a **link pointing to** page j.
- Our choice: **Nested list** of size n which we call G.
  - The **element** $G[i]$ represents the **links** leaving page i
    (i.e., a list containing other numbers in the range $0, \dots, n-1$).

Graph example

- The set of pages is $0, \ldots, 4$
- The set of links is: $(1,2), (1,3), (3,0),$ $(3,2), (3,4), (4,0), (4,3)$
- The "Pythonic" representation is:

$$\boldsymbol{G} = [\,[\,], [\mathbf{2}, \mathbf{3}], [\,], [\mathbf{0}, \mathbf{2}, \mathbf{4}], [\mathbf{0}, \mathbf{3}]\,]$$



Implementing the algorithm

- Algorithm $(\boldsymbol{G}, \boldsymbol{t}, \boldsymbol{p})$:
  - Initialize the current node ($curr$) to **0**
  - Initialize a $counter$ for *each* page in G
  - For t times:
    - With probability p: $curr$ = random **link** *from* $curr$
    - Otherwise: $curr$ = random **page** in G
    - Increase the counter of curr by **1**
  - Return the $\frac{Page's\ counter}{t}$ for each page in G

```
curr = 0
```

```
counter = [0 for i in range(len(G))]
```

```
curr = random.choice(G[curr])
```

```
curr = random.randrange(len(G))
```

```
counter[curr] += 1
```

```
if random.random() < p:
```

```
[cnt/t for cnt in counter]
```

- There are various methods of dealing with **sink pages** (those with no outgoing links).
- Our choice today: if $curr$ is at a sink, we **jump arbitrarily**.

Accuracy and confidence

- We **run** the algorithm for t steps and return the **weights**.
  - How do we know if these weights are **accurate**?
  - **What** does accurate even **mean**?
- **Claim (some math needed, we won't prove it):** if $p < 1$, then there is a **unique** set of weights w^*.

  (that is, the limit of this process as $t \to \infty$ exists and is unique).
- Not true if $p = 1$, easy example?
- We call our weights w and the "real" weights w^*.
- When do we say that w is *"close"* to w^*?

Accuracy and confidence (cont.)

- We define the **distance** between the **weights** to be the sum of the absolute values of the differences: $d(w, w^*) = \sum_{i=0}^{n-1} |w[i] - w^*[i]|$
- If $d(w, w^*) = 0$, we are clearly **done**.
- **Problem?**
  - WE DON'T KNOW w^*!!!
- **Solution:**
  - Denote w_t as the set of weights at time t
  - Denote w_{t+1} as the set of weights at time $t + 1$.
  - Then if $d(w_t, w_{t+1}) = 0$ we are also **done**; the system is pretty ***stable***.
  - A bit ambitious…
    - So, we choose some **small** $\epsilon > 0$ such that if $d(w_t, w_{t+1}) < \epsilon$ we **stop** the process.