pagerank-dmat-ue

February 11, 2025

1 Algebra for Search Engines

We'll see how **Linear Algebra** is at the heart of **Online Search Engines** such as Qwant, Ecosia or Google.

1.1 About me

- Francisco Coelho
- `fc@uevora.pt`
- Mathematics, Logic, Computer Science, Artificial Intelligence

1.2 Small Intro

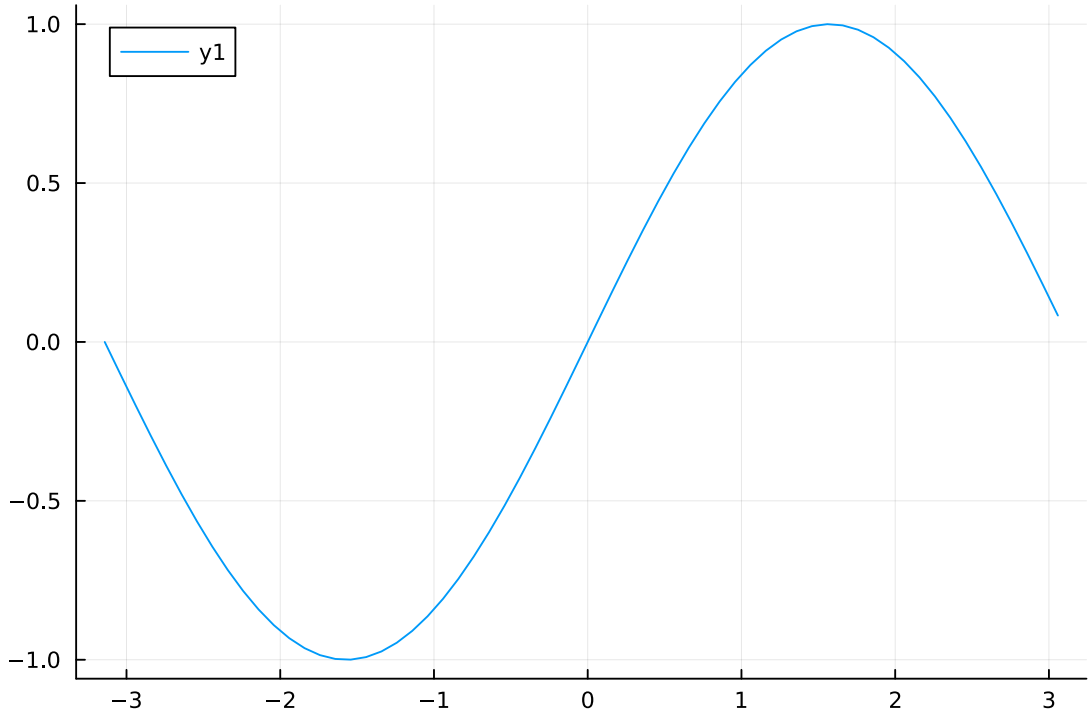
1.2.1 Julia for research

Julia is a great programming language for *research*.

[1]:
```
using Plots
```

[2]:
```
x = -:0.1: ;
y = sin.(x);
```

[3]:
```
plot(x,y)
```



1.2.2 Notebooks for communication

Jupyter notebooks are a great tool to *communicate* science.

Hopefully, this presentation is a good example.

1.3 A Mathematical Model of the Internet

The internet is a **digraph**. Each page is a **vertex** and there's an **edge** from a to b if a "links" to b.

$$G = (V, E)$$

Let's create a digraph in Julia.

[4]:
```
using Graphs, GraphPlot, GraphRecipes, Printf
```

A digraph can be setup "by hand". *Let's **not** do that.*

[5]:
```
# G = SimpleDiGraph(5);

# add_edGe!(G, 1, 2);
# add_edGe!(G, 1, 4);
```

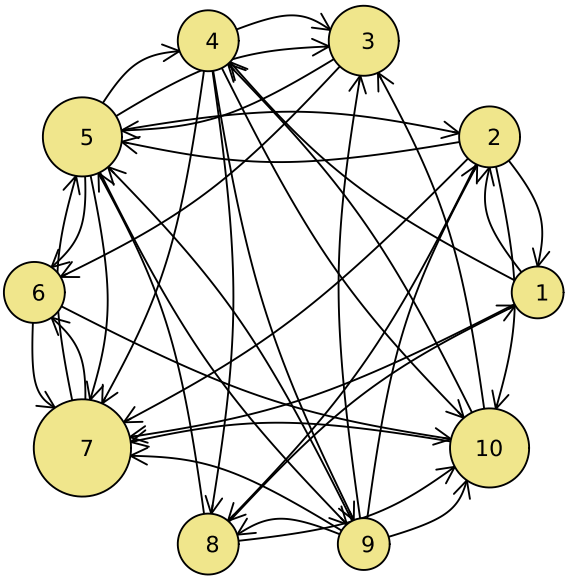
```
# add_edGe!(G, 2, 3);
# add_edGe!(G, 2, 5);
# add_edGe!(G, 3, 4);
# add_edGe!(G, 3, 1);
# add_edGe!(G, 4, 2);
# add_edGe!(G, 4, 1);
# add_edGe!(G, 5, 1);
```

Instead, here's an arbitrary sized random graph, without loops. *For fun, of course.*

[6]:
```
n = 10;

G = SimpleDiGraph(n);          # A digraph with n vertices
V = vertices(G);               # These are the vertices of G
for a in V
    outs = rand(V, rand(1:n)) # Generate a list of descendents of vertex a
    for b in outs
        if b != a              # No loops!
            add_edge!(G, a, b)# Add an edge a -> b
        end
    end
end
```

A plot of the graph.

[7]:
```
sizes = [indegree(G, v) for v in V]
graphplot(G,
    method=:circular,
    names=[@sprintf("%2d", x) for x in 1:n], # Hack to the node size $(
    nodeweights=sizes,
    markercolor=colorant"khaki",
    nodeshape=:circle,
    nodesize=1,
    fontsize=8,
    linecolor=:darkgrey,
    markersize=0.2
)
```



The size of the node reflects the *indegree*: the number of *incoming* edges.

What is the indegree of each vertex?

[8]:
```
importances = Dict(zip(V, sizes)) # match each vertex with its in-degree
sort(importances, # sort
    byvalue=true, # by in-degree
    rev=true      # in reverse order: the largest first
)
```

```
OrderedCollections.OrderedDict{Int64, Int64} with 10 entries:
  7  => 7
  5  => 5
  10 => 5
  3  => 4
  4  => 3
  6  => 3
  2  => 3
  8  => 3
  9  => 2
  1  => 2
```

1.3.1 Research Questions

What is the **most important** vertex in that graph?

Better yet: How to **rank the vertices** by *importance*?

> **Is *indegree* a good *importance* measure? - No.** *It is easy to cheat.* To "promote" a vertex just create many parents pointing to it.
>
> **Condition 1** The *importance* of a vertex must depend on the *importance* of it's parents.

Should all the edges contribute the same? - No. A vertex with many descendents would have disproportional impact.

> **Condition 2** The *contribute* of a vertex must be divided by it's descendents.

- We would like to rank pages respecting conditions 1 and 2.
- And, of course, the algorithm to do that must be **efficient**.

1.3.2 A Random Walk Approach

You start somewhere in the graph and go on, from vertex to vertex, every time choosing a random neighbour to jump.

> *Where would you end after infinite jumps?*

Or better:

> **What is the probability of ending on a given vertex after infinite jumps?**

Enter **algebra**.

1.4 Algebraic Treatment of Graphs

A graph $G = (V, E)$ has (among others) these two common matrix representations:

The **incidence** matrix relates *vertices* and *edges*: - m_{ij} iff vertex v_i belongs to edge e_j.

The **adjacency** matrix "defines" each *edge* through its *vertices*: - a_{ij} iff $(v_j, v_i) \in E$

> Column i shows the *outedges* of vertex v_i.

Here is the adjacency matrix of the graph above:

[42]:
```
adjacency_matrix(G)'
```

```
10×10 Adjoint{Int64, SparseArrays.SparseMatrixCSC{Int64, Int64}} with 37 stored␣
↪entries:
   1               1
 1          1          1
         1  1          1  1
 1          1             1
   1  1          1  1  1
      1     1     1
 1  1     1  1  1        1  1
   1     1             1
         1  1
   1     1     1     1  1
```

This is a *sparse* matrix. It is an *optimized* representation of matrices with lots of zeros.

Our examples are small enough to use *dense* matrices.

[43]:
```
A = Matrix(adjacency_matrix(G)')
```

```
10×10 Matrix{Int64}:
 0  1  0  0  0  0  0  1  0  0
 1  0  0  0  1  0  0  0  1  0
 0  0  0  1  1  0  0  0  1  1
 1  0  0  0  1  0  0  0  0  1
 0  1  1  0  0  0  1  1  1  0
 0  0  1  0  1  0  1  0  0  0
 1  1  0  1  1  1  0  0  1  1
 0  1  0  1  0  0  0  0  1  0
 0  0  0  1  1  0  0  0  0  0
 0  1  0  1  0  1  0  1  1  0
```

So, column i shows the descendants of vertex i.

[11]:
```
# The descendants of vertex 3
A[:,3]
```

```
10-element Vector{Int64}:
 0
 0
 0
 0
 1
 1
 0
 0
 0
 0
```

A 1 in position j of column i means that there's an edge $v_i \to v_j$.

Does it check?

The outedges of v_2 are:

[44]:
```
[i for i in 1:n if A[i, 2] != 0]
```

```
5-element Vector{Int64}:
  1
  5
  7
  8
 10
```

Does it check?

1.4.1 Playing with the Adjacency Matrix

The adjacency matrix *describes paths* in the graph.

Vertices Vertices are represented by *one-hot* column vectors.

Vertex i is represented by the vector $v_i \in \mathbb{R}^{n\times 1}$ such that $(v_i)_j = \delta_{ij}$.

v_i is the i-th column of the identity matrix $I^{n\times n}$ for a graph with n vertices.

[13]:
```
v(i,n) = [ i == j for j in 1:n ]
```

```
v (generic function with 1 method)
```

Let's check.

If we start in vertex 3, we have v_3:

[14]:
```
v3 = v(3,n)
```

```
10-element Vector{Bool}:
 0
 0
 1
 0
 0
 0
 0
 0
 0
 0
```

Descendants And in G we can reach the vertices Av_3:

[15]:
```
descendants_3 = A * v3
```

```
10-element Vector{Int64}:
 0
 0
 0
 0
 1
 1
 0
 0
 0
 0
```

This means that in g, there is a path of length 1 from vertex 3 to vertices…

[16]:
```
[i for i in 1:n if descendants_3[i] == 1]
```

```
2-element Vector{Int64}:
 5
 6
```

Does it check?

Let's state this:

> If A is the adjacency matrix of graph G and v_i the column representation of vertex i then
>
> $$Av_i$$
>
> are the vertices reachable from v_i.

Paths If we repeat this process we get the paths of length 2, 3, *etc.*

Where can we get in two steps, starting at vertex 3?

$$AAv_3$$

[17]:
```
A * A * v3
```

```
10-element Vector{Int64}:
 0
 1
 1
 1
 0
 1
 2
 0
 1
 1
```

The values we get here are **the number of paths** from the start vertex.

Does it check?

This is nice! Let's state it:

> Let A and v_i as before. Then the j-th row of
>
> $$A^n v_i$$
>
> is the number of length n paths $v_i \to v_j$.

1.4.2 Playing with Random Walks

What is a **random walk** in a graph?

1. Start at a given vertex, say v_0.
2. Move through a *random* edge.
3. Repeat.

The **key** question here is:

> What is the **probability of reaching a given vertex** u after n steps starting at vertex v_0?
>
> To address this question we transform the adjacency matrix into a **transition** matrix.

Instead of the (boolean) information about edges $v_i \to v_j$ we encode the **probability** of going from v_i to v_j:

$$P(v_i \to v_j)$$

We can also think as the edges having "*weights*", "*costs*", *etc.*

Recall our research question:

> **What is the probability of ending on a given vertex after infinite jumps?**

In any given vertex there are no *preferred* targets. This means that

> The **transition** matrix results from the **adjacency matrix** by normalizing each column.

Right?

[18]:
```
using LinearAlgebra
```

Recall that the **adjacency matrix** is...

[19]:
```
A
```

```
10×10 Matrix{Int64}:
 0  1  0  0  0  0  0  1  0  0
 1  0  0  0  1  0  0  0  1  0
 0  0  0  1  1  0  0  0  1  1
 1  0  0  0  1  0  0  0  0  1
 0  1  1  0  0  0  1  1  1  0
 0  0  1  0  1  0  1  0  0  0
 1  1  0  1  1  1  0  0  1  1
 0  1  0  1  0  0  0  0  1  0
 0  0  0  1  1  0  0  0  0  0
 0  1  0  1  0  1  0  1  1  0
```

... so the **transition matrix** becomes:

[20]:
```
# Normalize a column (ie: sum(column) = 1)
normalizer(col) = col ./ sum(col)

# Apply "normalizer" to each column and gather the results in a new matrix.
T = hcat( normalizer.(eachcol(A))... )
```

```
10×10 Matrix{Float64}:
 0.0       0.2  0.0  0.0  0.0       0.0  0.0  0.333333  0.0       0.0
 0.333333  0.0  0.0  0.0  0.166667  0.0  0.0  0.0       0.166667  0.0
 0.0       0.0  0.0  0.2  0.166667  0.0  0.0  0.0       0.166667  0.333333
```

```
0.333333  0.0  0.0  0.0  0.166667  0.0  0.0  0.0       0.0       0.333333
0.0       0.2  0.5  0.0  0.0       0.0  0.5  0.333333  0.166667  0.0
0.0       0.0  0.5  0.0  0.166667  0.0  0.5  0.0       0.0       0.0
0.333333  0.2  0.0  0.2  0.166667  0.5  0.0  0.0       0.166667  0.333333
0.0       0.2  0.0  0.2  0.0       0.0  0.0  0.0       0.166667  0.0
0.0       0.0  0.0  0.2  0.166667  0.0  0.0  0.0       0.0       0.0
0.0       0.2  0.0  0.2  0.0       0.5  0.0  0.333333  0.166667  0.0
```

Let's check:

[21]:
```
sum.(eachcol(T))'
```

```
1×10 adjoint(::Vector{Float64}) with eltype Float64:
 1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

Playing with the Transition Matrix What is the probability distribution of reaching each vertex, starting at v_3?

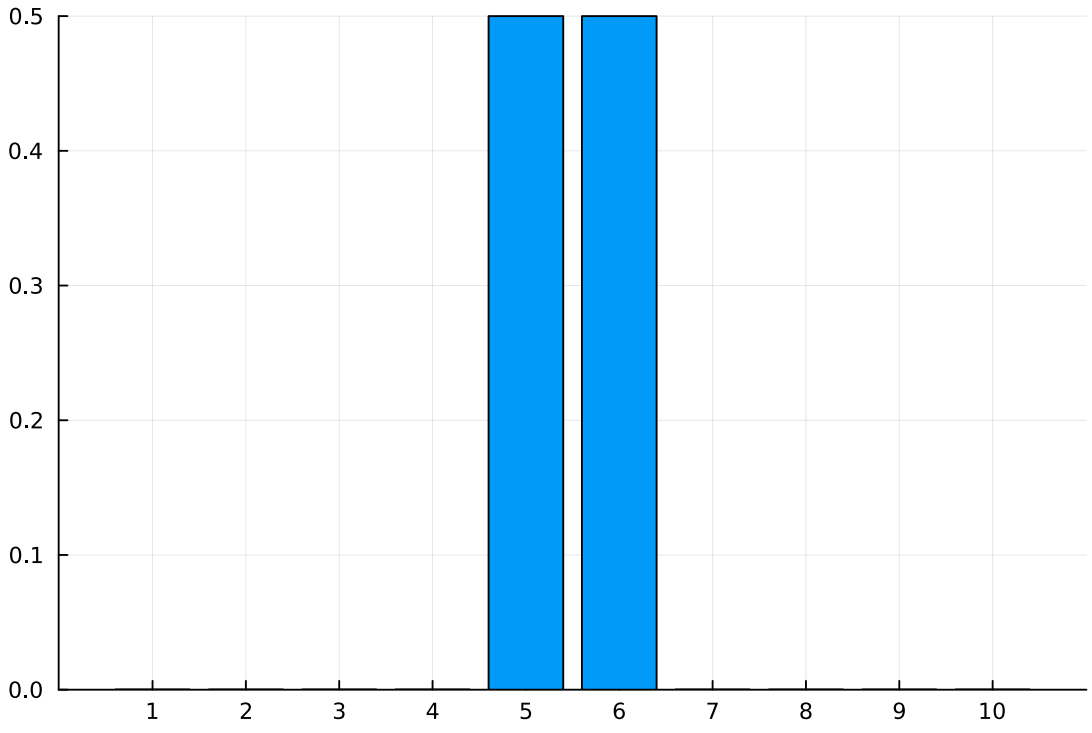
$$Tv_3$$

[22]:
```
T * v3
```

```
10-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.5
 0.5
 0.0
 0.0
 0.0
 0.0
```

We can plot this, for better visualization:

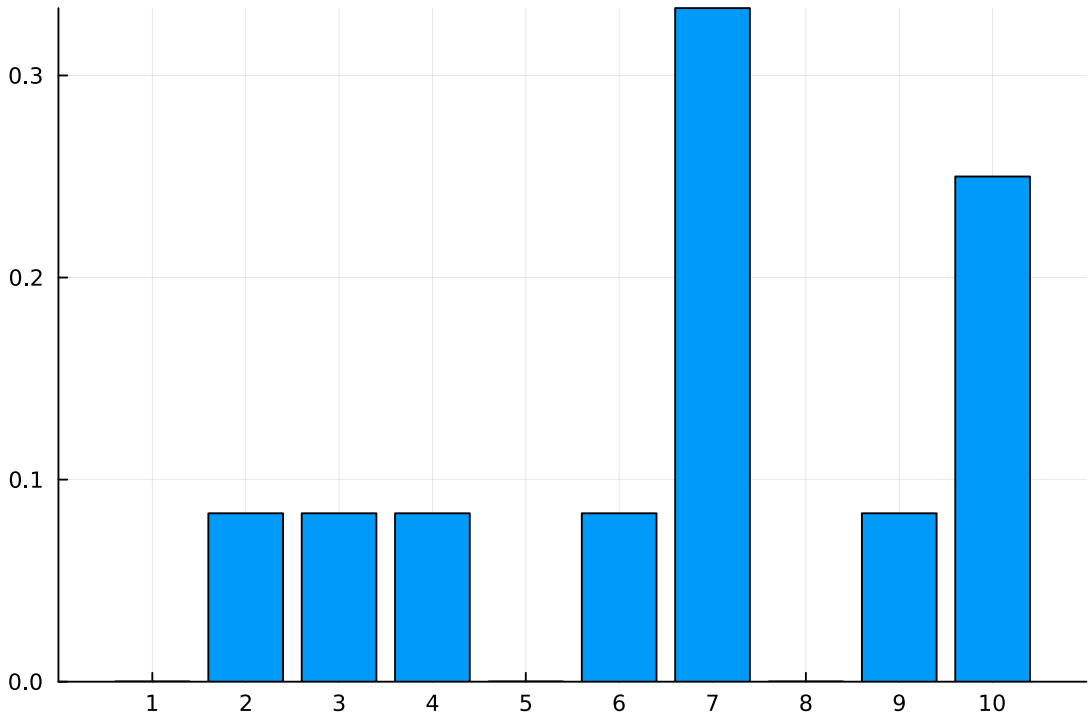
[23]:
```
bar(T * v3, xticks=1:n, legend=nothing)
```



And after two steps?

$$TTv_3$$

[24]:
```
bar(T * T * v3, xticks=1:n, legend=nothing)
```

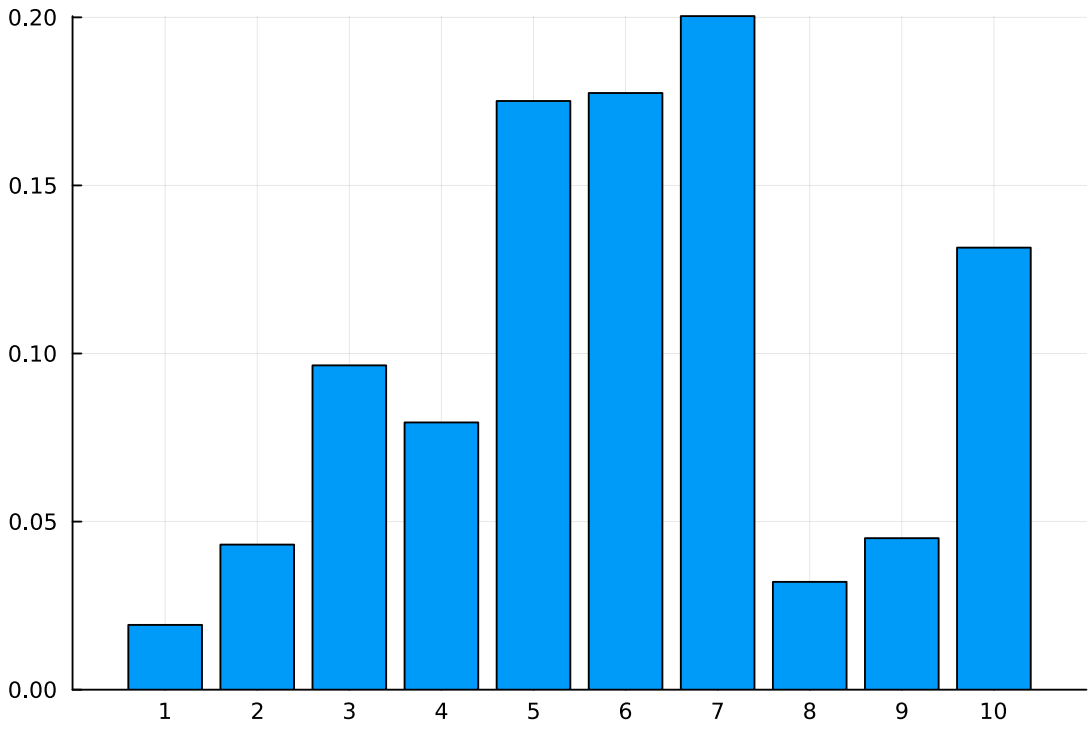


And after 10 steps?

$$

T^{10} v_3 $$

[25]:
```
bar(T^10 * v3, xticks=1:n, legend=nothing)
```



By the way… *is this a probability distribution*?

[26]:
```
sum(T^10 * v3)   1
```

```
true
```

And if we **don't know where to start?**

Suppose that we can start at vertex 2 with probability 0.7 and at vertex 4 with probability 0.3.

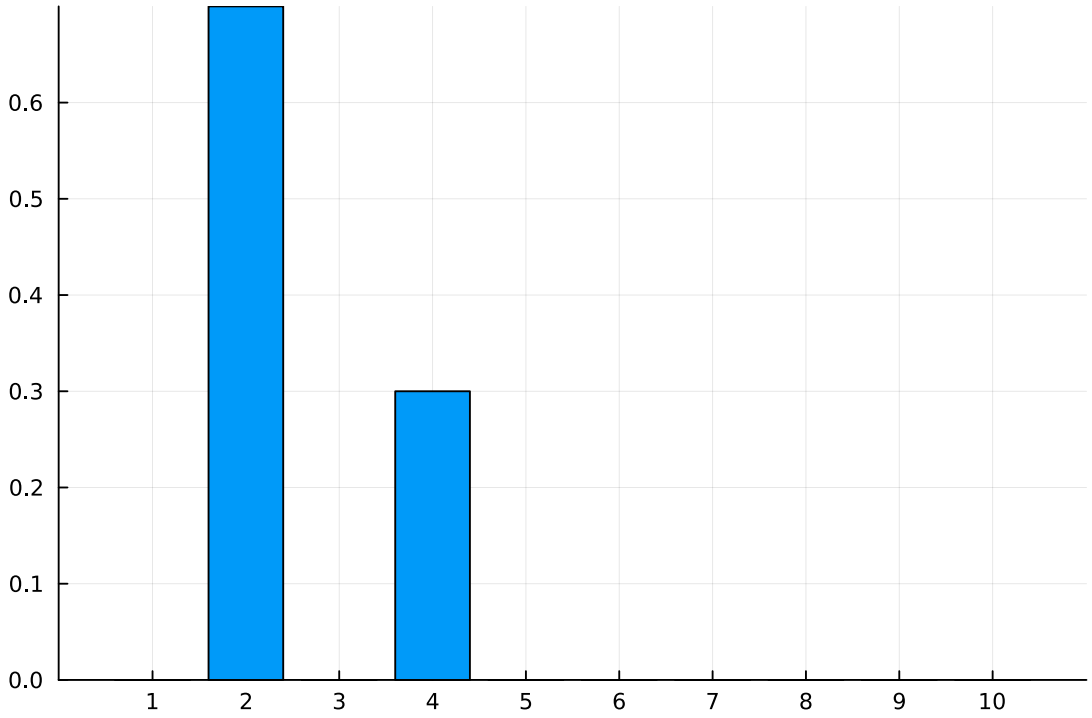
Then

$$v_0 = \begin{bmatrix} 0.0 \\ 0.7 \\ 0.0 \\ 0.3 \\ 0.0 \\ \vdots \end{bmatrix}$$

[27]:
```
v0 = zeros(n)
v0[2] = 0.7
v0[4] = 0.3;
```

Initial distribution

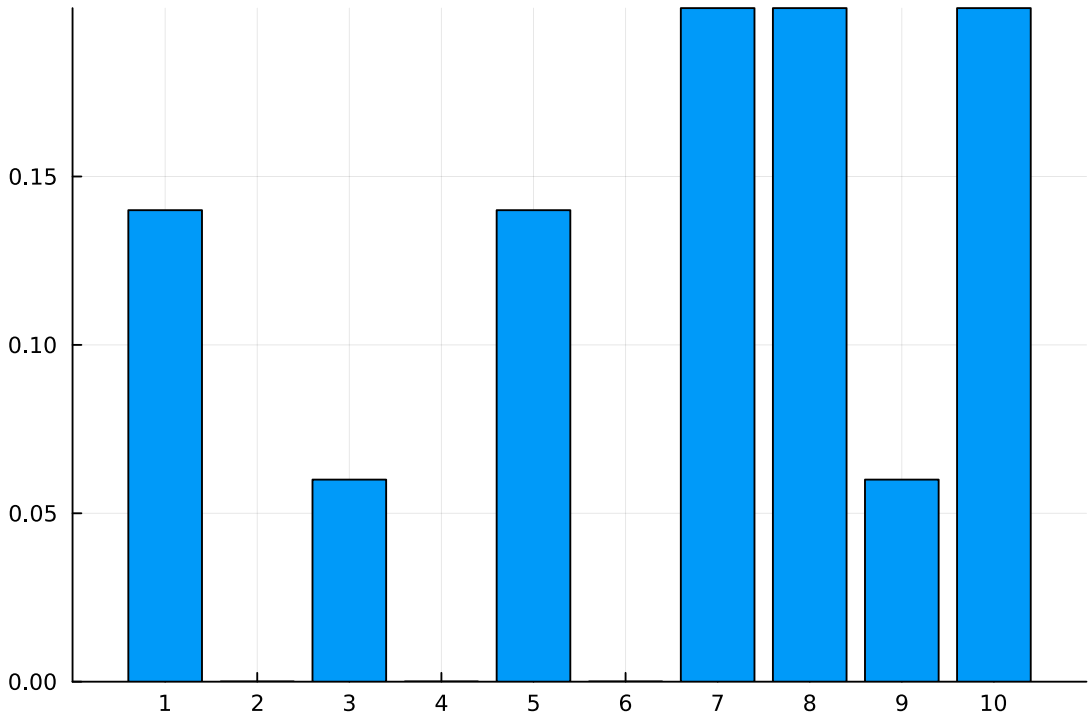
[28]:
```
bar(v0, xticks=1:n, legend=nothing)
```



After one step

$$Tv_0$$

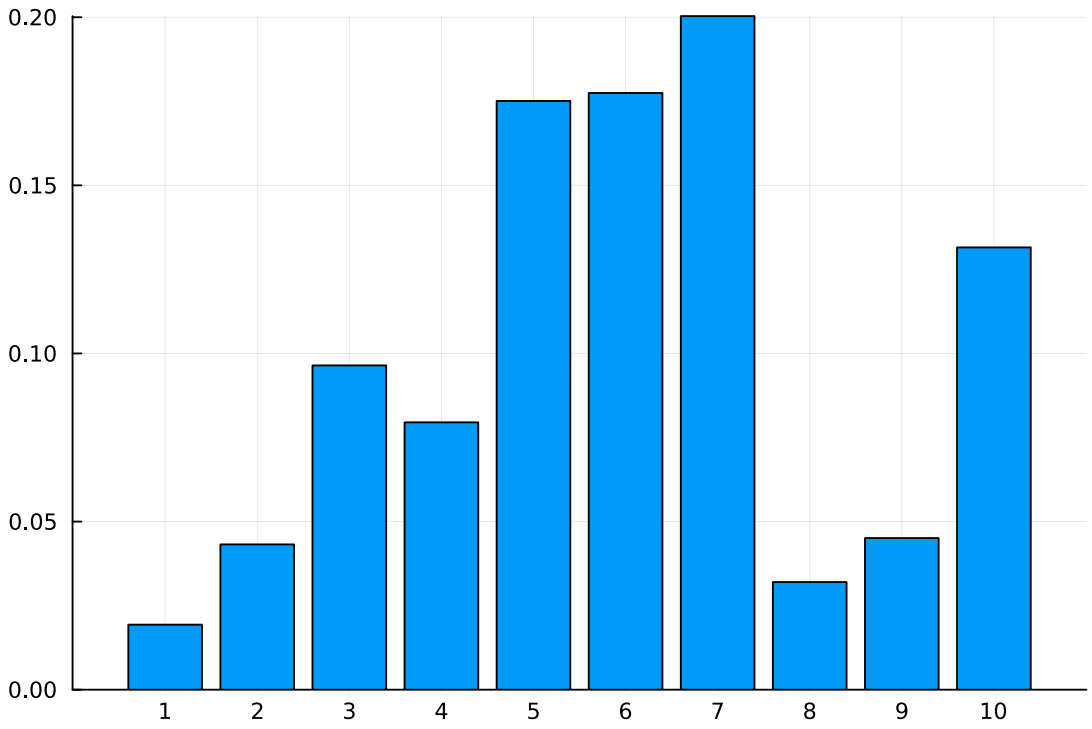
[29]:
```
bar(T * v0, xticks=1:n, legend=nothing)
```



After 10 steps

$$T^{10} v_0$$

[30]:
```
bar(T^10 * v0, xticks=1:n, legend=nothing)
```



Suppose that each vertex has the same probability to start:

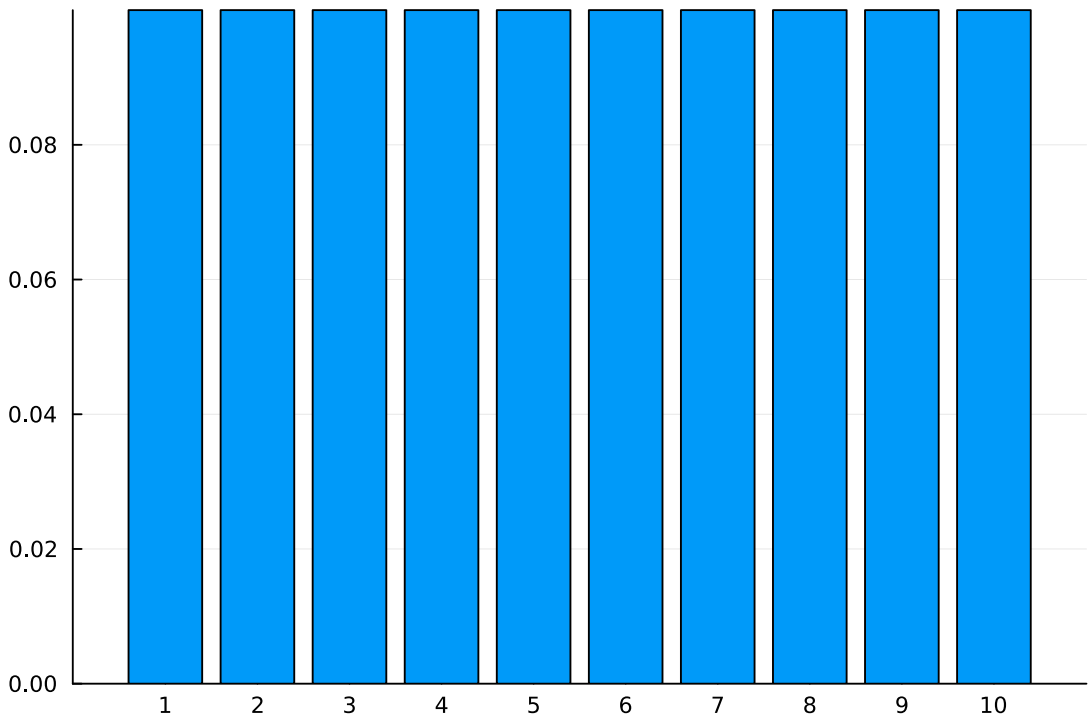
Then

$$v_0 = \begin{bmatrix} p \\ \vdots \\ p \end{bmatrix}$$

where $p = \frac{1}{n}$.

Initial distribution

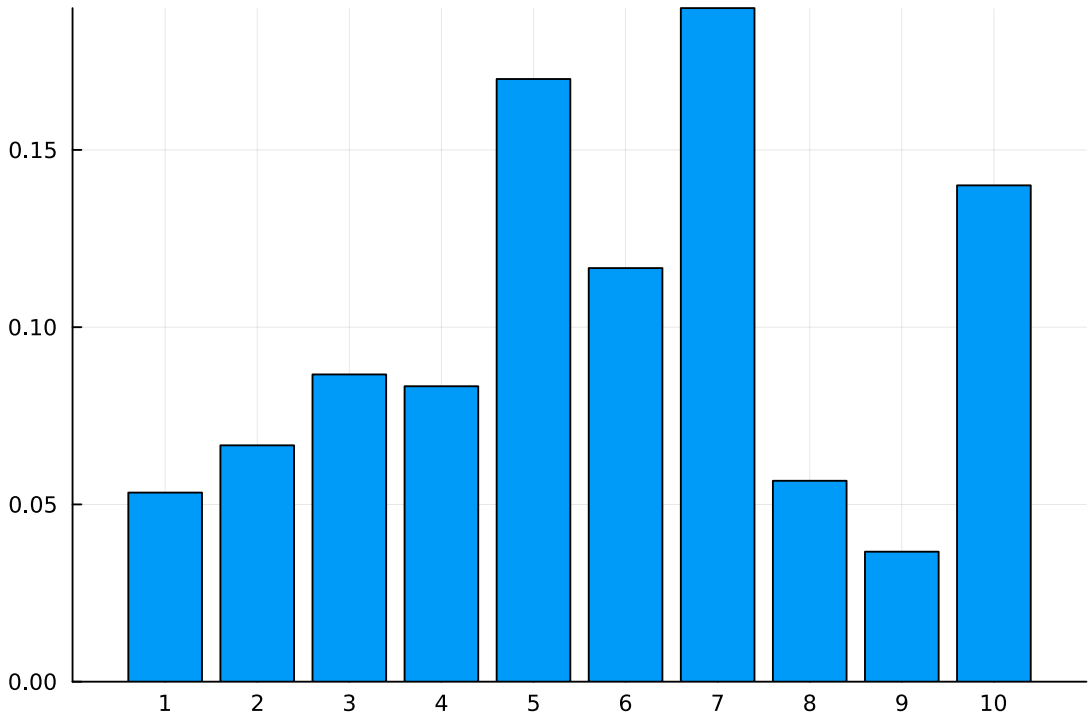
[31]:
```
p = 1.0 / n;
vp = p .* ones(n);

bar(vp, xticks=1:n, legend=nothing)
```



After one step

$$Tv_p$$

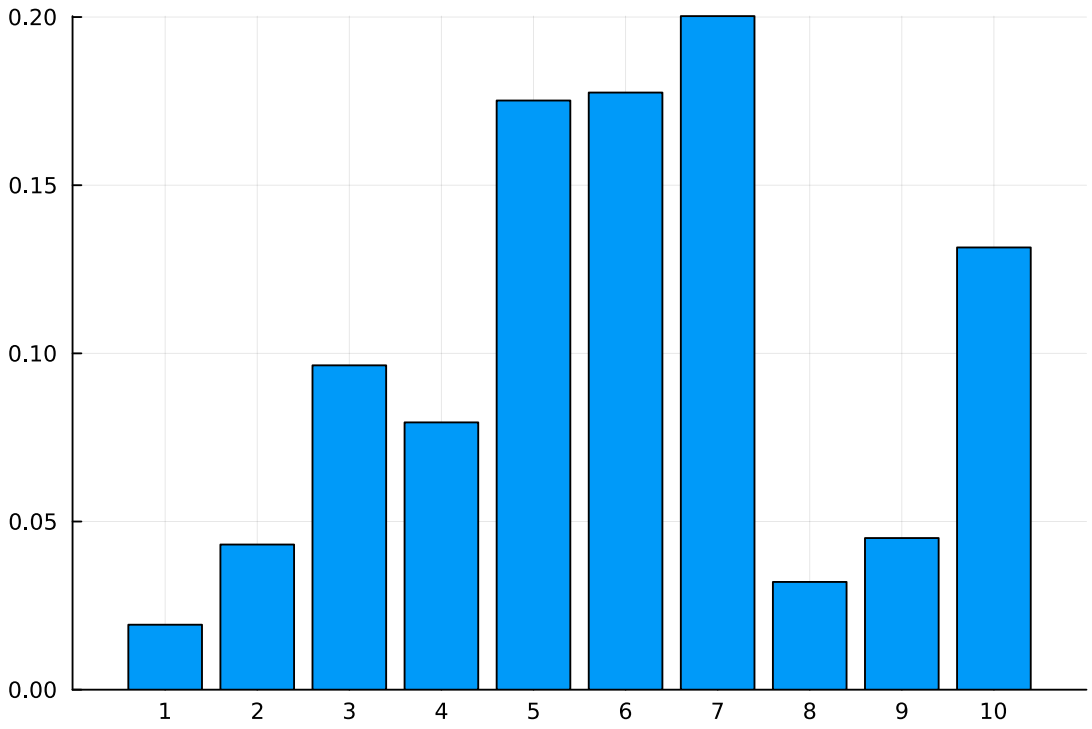
[32]:
```
bar(T * vp, xticks=1:n, legend=nothing)
```



After 10 steps

$$T^{10} v_p$$

[33]:
```
bar(T^10 * vp, xticks=1:n, legend=nothing)
```



Let's summarize.

1. The **transition matrix**, T, represents the probability of crossing the edge $v_i \to v_j$.
2. Given an **initial distribution** of the vertices, v_0, the **final distribution** after n steps is given by
$$v_n = T^n v_0$$
where v_0 is the *uniform* distribution of vertices.

1.5 To the infinite

Now we have an algebraic formulation of our research question:

What is the probability of ending on a given vertex after infinite jumps?

Given the *transition matrix* of the internet, T, and a *uniform initial distribution* of the pages, v_0, what is
$$\lim_{n\to\infty} T^n v_0$$

There are two ways of addressing this problem.

1.5.1 The *Naive* resolution

- Suppose that T is diagnosable (*why not*?). Then
$$T = PDP^{-1}$$

where D is a diagonal matrix.

Furtermore: D contains the *eigenvalues* of T and the *columns* of P are the (associated) *eigenvectors*.

So

$$T^n = PD^nP^{-1}$$

and D^n is very easy to compute: $(D^n)_{ii} = (d_{ii})^n$.

It is always a good idea to look at the eigenvalues…

In our graph, the eigen values/vectors of T are

[34]:
```
evv = eigen(T)
```

```
Eigen{ComplexF64, ComplexF64, Matrix{ComplexF64}, Vector{ComplexF64}}
values:
10-element Vector{ComplexF64}:
    -0.3497671963601029 - 0.2608191080457634im
    -0.3497671963601029 + 0.2608191080457634im
   -0.25619614214600794 - 0.31010273881464334im
   -0.25619614214600794 + 0.31010273881464334im
   -0.20458588344395368 + 0.0im
 1.7344840435774148e-16 + 0.0im
   0.015394113017583741 - 0.09071069229697648im
   0.015394113017583741 + 0.09071069229697648im
     0.3857243344210082 + 0.0im
     1.0000000000000022 + 0.0im
vectors:
10×10 Matrix{ComplexF64}:
  0.0473934+0.07705im    …     0.45495+0.0im  -0.0515084+0.0im
  -0.248773+0.0440162im        0.375535+0.0im   -0.115103+0.0im
 0.00927168-0.153492im        -0.121192+0.0im   -0.257254+0.0im
   -0.13541-0.119039im         0.165546+0.0im   -0.211981+0.0im
   0.596437-0.0im            -0.0884154+0.0im   -0.467316+0.0im
   0.368311+0.125783im   …     -0.60215+0.0im   -0.473647+0.0im
  -0.400116-0.126623im        -0.313864+0.0im   -0.534268+0.0im
   0.159822-0.144342im         0.301135+0.0im  -0.0854636+0.0im
  -0.100266+0.142836im        0.0476332+0.0im   -0.120282+0.0im
  -0.296669+0.15381im         -0.219177+0.0im   -0.350775+0.0im
```

Pfff… there are lots of complex numbers and this is not really helping …

Wait!!!

1 is an eigen value!

[35]:
```
evv.values .  1
```

```
10-element BitVector:
 0
```

```
 0
 0
 0
 0
 0
 0
 0
 0
 1
```

An associated eigen vector is

[36]:
```
ev = evv.vectors[:, evv.values .  1]
```

```
10×1 Matrix{ComplexF64}:
 -0.051508388059579274 + 0.0im
  -0.11510254314989735 + 0.0im
  -0.25725418394002103 + 0.0im
   -0.2119805169677277 + 0.0im
  -0.46731632518858357 + 0.0im
  -0.47364696506182774 + 0.0im
   -0.5342676377874402 + 0.0im
   -0.0854636382887989 + 0.0im
   -0.1202821575916429 + 0.0im
   -0.3507750002493124 + 0.0im
```

which is a **real** vector. Let's check that:

[37]:
```
all(imag.(ev) .== 0)
```

```
true
```

Let's discard the imaginary parts:

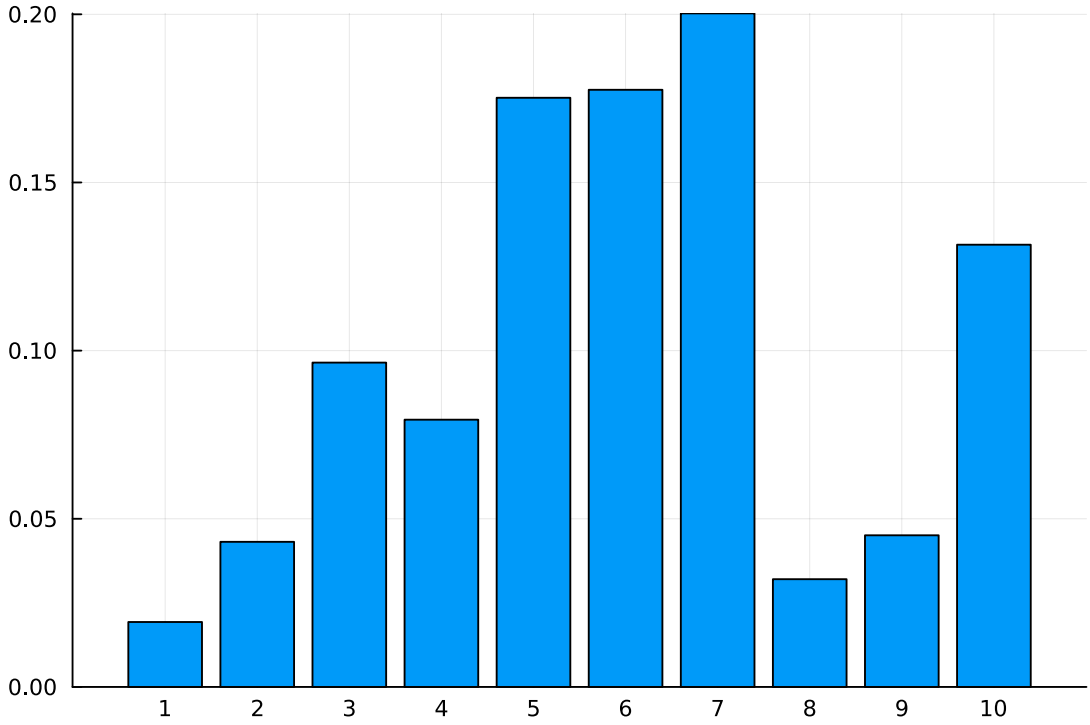
[38]:
```
evr = real.(ev)
```

```
10×1 Matrix{Float64}:
 -0.051508388059579274
 -0.11510254314989735
 -0.25725418394002103
 -0.2119805169677277
 -0.46731632518858357
 -0.47364696506182774
 -0.5342676377874402
 -0.0854636382887989
 -0.1202821575916429
 -0.3507750002493124
```

And visualize the **limit**

$$\lim_{n\to\infty} T^n v_0$$

[39]:
```
bar(evr ./ sum(evr), xticks=1:n, legend=nothing)
```



But

This method is not practical.

The "real" T matrix today would have around 1100 millions of rows. This is too much to process "directly" with `eigen`.

Also...

Why the focus on the *eigenvalue* 1?

well... the **eigenvalue** 1 and the **associated eigenvector** leads us to...

1.5.2 The *Elegant* Resolution

Let

$$v = \lim_{n\to\infty} T^n v_0.$$

So it must be

$$Tv = v.$$

Because if $Tv \neq v$ then v is not the $\lim_{n\to\infty} T^n v$. *Right?*

$$v = \underbrace{T \cdots T}_{n\to\infty} v_0 = T \underbrace{T \cdots T}_{n\to\infty} v_0$$

Therefore v **is an eigenvector associated to the eigenvalue** 1.

To compute an eigenvector associated to the eigenvalue 1 we solve

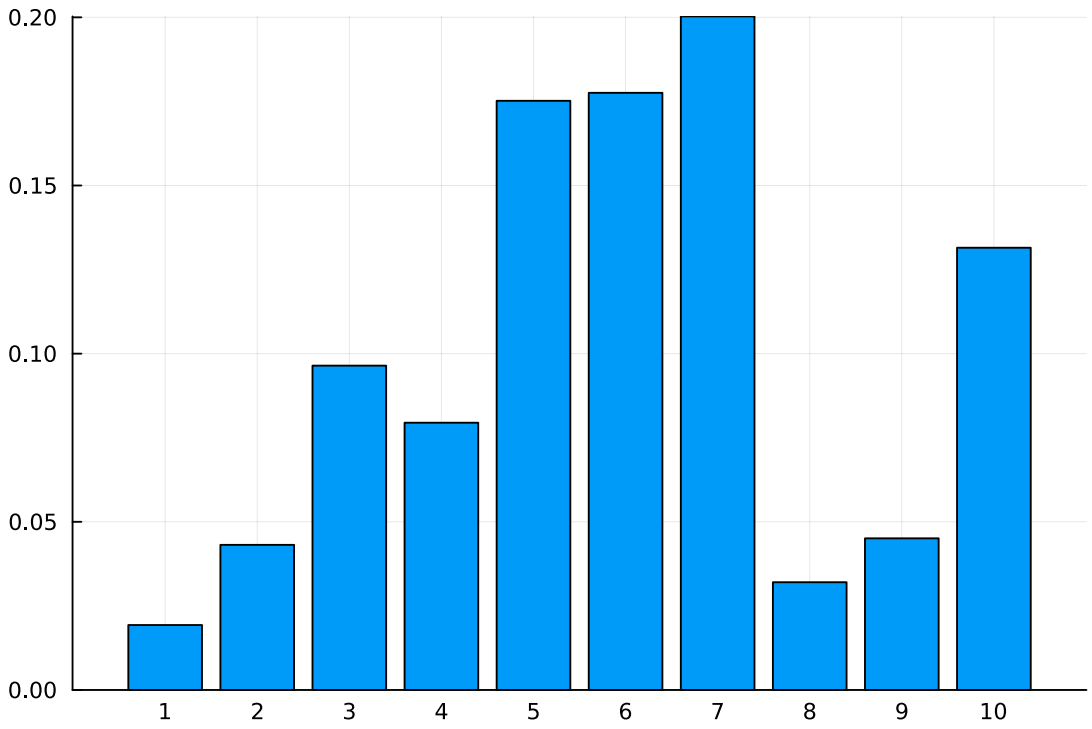
$$Tv - v = 0 \Leftrightarrow (T - I)v = 0$$

[40]:
```
ev = nullspace(T - I(n))
```

```
10×1 Matrix{Float64}:
 0.051508388059579135
 0.11510254314989754
 0.25725418394002086
 0.21198051696772766
 0.4673163251885837
 0.473646965061828
 0.5342676377874407
 0.0854636382887988
 0.12028215759164282
 0.35077500024931235
```

The final page rank is:

[41]:
```
bar(ev ./ sum(ev), xticks=1:n, legend=nothing)
```



1.6 And behind

We have shown that

> **Linear Algebra**, in particular **Spectral Analysis**, is at the core of the **PageRank** algorithm, used every day in **Search Engines**.

However *there are limitations to the* ***basic*** *method presented here*. What if the network has separate components?

I also took the liberty of deviating substantially from the traditional presentation to illustrate two modern tools that I have come to appreciate substantially:

- The Julia language to *prototype* reseach ideas — and do the *heavy computations* if required.
- Jupyter notebooks to *communicate* those ideas.

These are *free*, *open source* tools, as it **must be** to do true **Science**.

Thank you!

Questions?
