week07-02–FSM

February 24, 2025

1 George McNinch Math 87 - Spring 2025

2 Week 7

3 Finite-state machines

3.1 Finite-state machines

In this discussion, we are going to use a form of graph model to represent *states* and transitions between them.

Let's start with a pretty simple example:

3.2 Example: a door

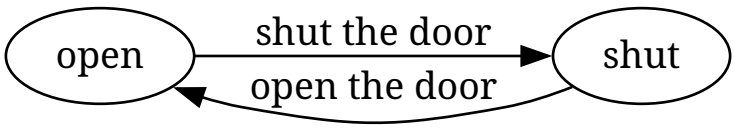
We are going to model exactly two states: *open*, and *shut*.

We represent these states as *nodes* in a directed graph.

The edges in the directed graph will represent *transitions*: we can *open the door* and we can *shut the door*.

[1]:
```
from graphviz import Digraph
door = Digraph()
door.attr(rankdir='LR')
door.node("open")
door.node("shut")
door.edge("shut","open","open the door")
door.edge("open","shut","shut the door")

door
```

[1]:



3.3 Example; elevator:

Consider an elevator in a building with two floors.

We'll model two states: *first floor* and *second floor*. These states represent the *current position of the elevator.*

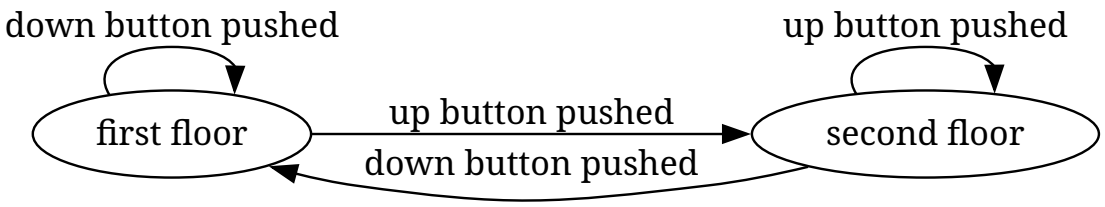
Inside the elevator, there are two buttons: *up* and *down*.

When the elevator is on the first floor, pushing the up button initiates a trip up, but the up button does nothing when the elevator is on the second floor.

Similar remarks hold for the down button.

We can model these states and transitions using the following diagram:

[2]:
```
from graphviz import Digraph
elev = Digraph()
elev.attr(rankdir='LR')
elev.node("first floor")
elev.node("second floor")

elev.edge("first floor","second floor","up button pushed")
elev.edge("first floor","first floor","down button pushed")
elev.edge("second floor","first floor","down button pushed")
elev.edge("second floor","second floor","up button pushed")

elev
```

[2]:


3.4 Definition of finite-state machine

A **finite state machine** (FSM, also called a finite-state automatata or FSA) is an ordered collection $(\Sigma, S, s_o, \delta, F)$ where:
- Σ is the "alphabet", a finite and non-empty set of symbols
- S is a finite, non-empty set of states
- $s_o \in S$ is an initial state
- $\delta : S \times \Sigma \to S$ is the state-transition function
- $F \subset S$ is the set of final states (F is possibly empty).

Sometimes we may drop the adjective *finite* and consider just a **state machine**; in this case, the set S of states may be infinite.

Given a finite state machine $M = (\Sigma, S, s_o, \delta, F)$, the symbols in Σ represent ways in which the state of M may transform:

- If M is in state s, then after applying the symbol $x \in \Sigma$, M is in state $\delta(s, x)$.
- If M is in state s, then we can apply a sequence $x_1, x_2, \ldots, x_n$ of symbols in Σ. After applying the sequence, M is in state

$$\delta(\cdots(\delta(\delta(\delta(s, x_1), x_2), x_3), \ldots), x_n)$$

Remark: sometimes δ is permitted to be a *partial* function. Thus, δ may not be defined for all pairs $(s, x) \in S \times \Sigma$.

3.5 Example:

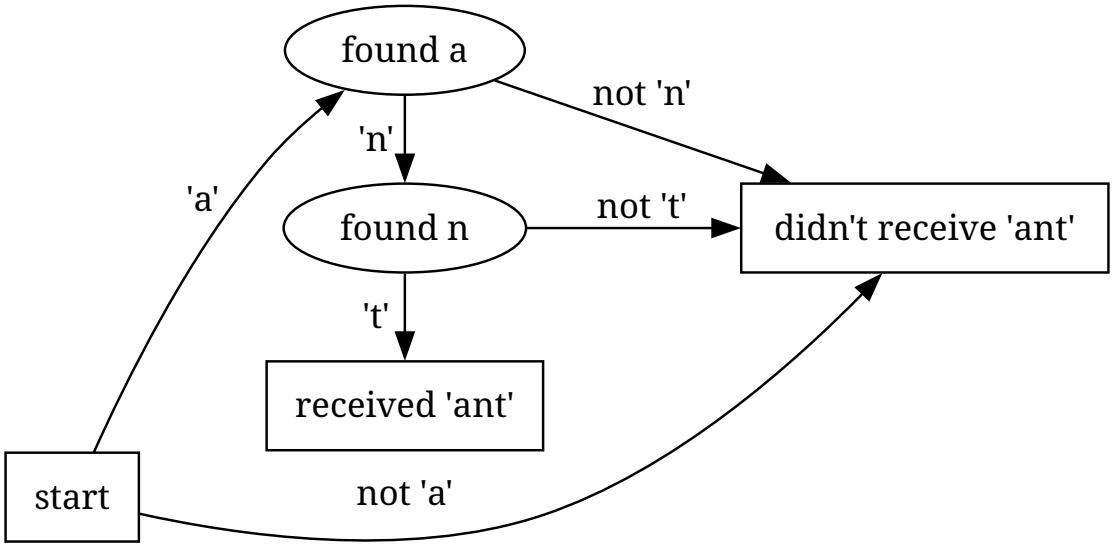
Consider a finite state machine which receives a sequence of characters, and checks whether the string is equal to "ant".

The initial state is "start".

The final states are "received 'ant' " and "didn't receive 'ant' "

[3]:
```
ant = Digraph("ant")
ant.attr(rankdir='LR')

ant.node("start",shape='box')

with ant.subgraph() as c:
    ant.attr(rank='same')
    c.node("found a")
    c.node("found n")
    c.node("received 'ant'",shape='box')

ant.node("didn't receive 'ant'",shape='box')

ant.edge("start","found a","'a'")
ant.edge("start","didn't receive 'ant'","not 'a'")
ant.edge("found a","found n","'n'")
ant.edge("found a","didn't receive 'ant'","not 'n'")
ant.edge("found n","received 'ant'","'t'")
ant.edge("found n","didn't receive 'ant'","not 't'")
ant
```

[3]:



Let's identify the components of the finite-state machine in this example:

- the alphabet Σ is the usual "alphabet" `[a,b,c,...,z]`
- S is the set of nodes of the diagram above, namely `[ start, found a, found n, received "ant", didn't receive "ant"]`
- `s0 = start`
- the final states are `[ received "ant", didn't receive "ant" ]`
- the transition function $\delta : S \times \Sigma \to S$ describes how to evolve the state as a letter is processed.

Observe that evolving the state by the characters of any word of length `>= 3` is guaranteed to arrive at one of the final states.

4 Probability and transition

Finite state machines provide a useful way of describing systems where the transitions between states have a probabilistic description.

For example:

4.1 "Drunkard's walk"

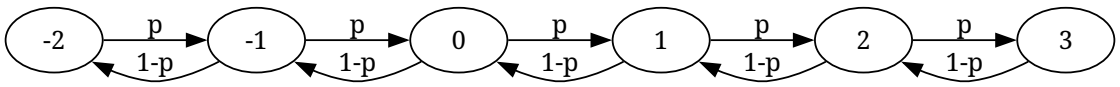
This is a so-called random walk. You are given nodes labelled by integers $n \in \mathbb{Z}$. (so in particular, there are infinitely many nodes!)

For each integer n, there is an edge $n \to n+1$ labelled with p, and there is an edge $n \to n-1$ labelled with $1-p$, for some $0 \leq p \leq 1$

Let's make a partial diagram:

[4]:
```
dw = Digraph("dw")
partial = list(range(-2,4))
dw.attr(rankdir='LR')

for n in partial:
    dw.node(f"{n}")

for n in partial:
    if n+1 in partial:
        dw.edge(f"{n}",f"{n+1}","p")

for n in partial:
    if n-1 in partial:
        dw.edge(f"{n}",f"{n-1}","1-p")

dw
```

[4]:



This system can be understood as follows:

- A state represents the position of some object.
- the object undergoes state changes – "move left" or "move right" – with respective probabilities `1-p` and `p`
- a standard choice is $p = 1/2$; then the state changes `move left` and `move right` happen with equal probabilty

As to our formal description of the FSM, in this case we have

- the states are given by the (infinite) set $S = \{0, \pm 1, \pm 2, \pm 3, \cdots\}$.
- $\Sigma =$ `[ left, right ]`
- `S0 = starting position`, often we'll take `S0 = 0`
- in this case, there are no distinguished final states; in fact `F = S`

The randomness is involved with the *input* to the finite state machine.

You can imagine coin tosses determining a sequence of inputs:

`[ left, left, right, left, right, right, right, ... ]`

4.2 multi-dimensional version

Of course, the "drunkard's walk" need not just occur in a line!

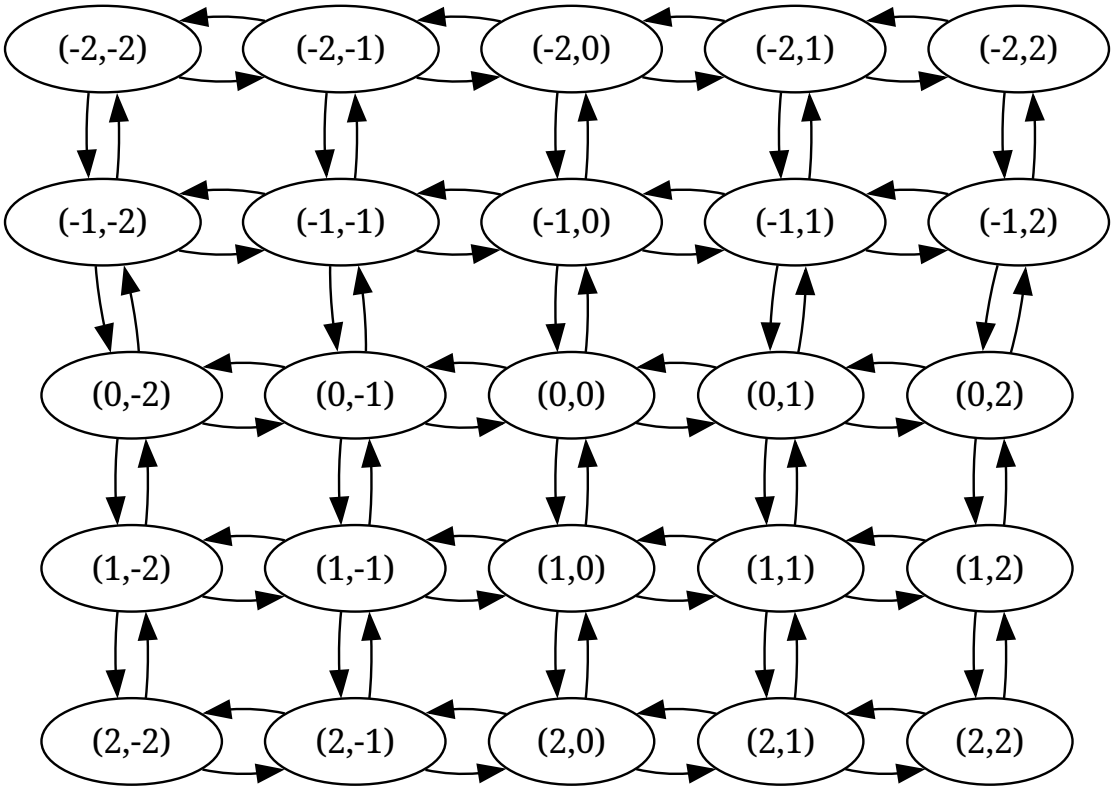
One might consider states labelled by pairs $(m, n) \in \mathbb{Z} \times \mathbb{Z}$.

Consider edges
- $(m, n) \to (m + 1, n)$

- $(m,n) \to (m,n+1)$
- $(m,n) \to (m-1,n)$
- $(m,n) \to (m,n-1)$

In this case, the object undergoes state changes ("move up", "move right", "move left", "move down") each with probability 1/4.

[5]:
```
import itertools as it

ddw = Digraph("dw")

p = list(range(-2,3))

pp = list(it.product(p,p))

for m in p:
    with ddw.subgraph() as c:
        c.attr(rank='same')
        for n in p:
            c.node(f"({m},{n})")

for (m,n) in pp:
    if (m+1,n) in pp:
        ddw.edge(f"({m},{n})",f"({m+1},{n})")
    if (m,n+1) in pp:
        ddw.edge(f"({m},{n})",f"({m},{n+1})")
    if (m-1,n) in pp:
        ddw.edge(f"({m},{n})",f"({m-1},{n})")
    if (m,n-1) in pp:
        ddw.edge(f"({m},{n})",f"({m},{n-1})")

ddw
```
[5]:



4.3 Transition probabilities

We want to track the state of a probabilistic FSM. Let's describe this using our example.

For the (1 dimensional) drunkard's walk, consider the following description:

- at time 0, we begin at state 0.
- at time 1, `M` is at state 1 with .50 probabilty, and we are at state -1 with .50 probability
- at time 2, `M` is at state 2 with .25 probability, at state 0 with .50 probability, and state -2 with .25 probability
- at time 3, `M` is| at state 3 with probability .125, at state 1 with probability .375, at state -1 with probability .375, and state -3 with probability .125

Here `time` refers to the `number of state transitions`

etc…

Let's describe the probabilities for the 2-dimensional version:

The possible state transitions are `[ u, d, l, r ]` each occuring with probabilty `1/4`

time	probability	possible states
0	1	[(0,0)]
1	.25	[(0,1), (0,-1), (-1,0), (1,0)]

Once we reach time 2, there is some redundancy in how a state is reached.

- For example, the transitions `[l, u]` and `[u, l]` both reach the state `(-1,1)`.
- On the other hand, the only sequence of transitions resulting in `(-2,0)` is `[ l, l ]`, which
- Finally, there are 4 ways of reaching the origin `(0,0)`, namely: `[l,r]`, `[r,l]`, `[u,d]`, `[d,u]`.

Note that there are $4^2 = 16$ possible sequences of transitions. So we see that - `(-2,0)` occurs with probability`1/16 = .0625` - `(-1,1)` occurs with probability `2/16 = .125` - `(0,0)` occurs with probability `4/16 = .25`

More generally, we have | time | probability | possible states | | —-:| ————:| ————————:| | 2 | 0.125 | [(-1,1), (-1,-1), (1,-1), (1,1)] | | 2 | 0.0625 | [(-2,0), (2,0), (0,2), (0,-2)] | | 2 | 0.25 | [(0,0)] |

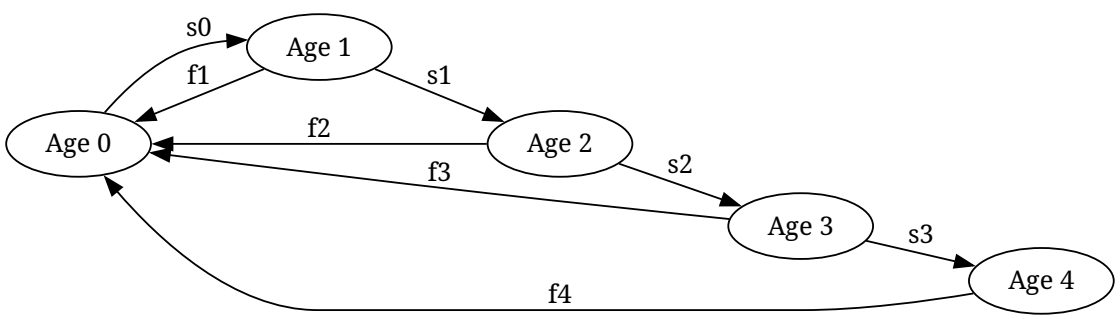
4.4 Example: aging and population growth

The nodes of the diagram will represent the age of an individual in a population.

The transitions corresponding to labels edges $s_i = (i \to i+1)$ represent probability of survival from age i to age $i+1$.

And the transitions $f_i : (i \to 0)$ represent probability of having an offspring at age i.

[6]:
```
pop = Digraph("pop")
pop.attr(rankdir='LR')

p = list(range(5))
with pop.subgraph() as c:
#     c.attr(rank='same')
    for i in p:
        c.node(f"Age {i}")

for i in p:
    if i+1 in p:
        pop.edge(f"Age {i}",f"Age {i+1}",f"s{i}")
    if i != 0:
        pop.edge(f"Age {i}","Age 0",f"f{i}")

pop
```

[6]:


4.5 Interpreting this model

- at time 0, suppose that the size of the population which is of age 0 is equal to p_0, the size of the population of age 1 is equal to p_1, etc.

  More succinctly, the population is described by the sequence $(p_0, p_1, \dots)$.

  Note that the total population is equal to the sum $\sum_{i=0}^{\infty} p_i$, which looks a bit odd! But, the infinite sum isn't really infinite – p_i must be equal to 0 for all sufficiently large values of i).

- at time 1, the size of the population of age 0 is given by

  $f_1 p_1 + f_2 p_2 + \cdots = \sum_{i=1}^{\infty} f_i p_i$.

  The size of the population of age 1 is given by the product $s_0 \cdot p_0$, and more generally for $i \geq 1$ the size of the population of age i is given by $s_{i-1} p_{i-1}$.

  Thus the population at time 1 is described by the sequence

  $(\sum_{i=1}^{\infty} f_i p_i,\ s_0 p_0,\ s_1 p_1,\ \dots)$

  And in particular the total population at time 1 is given by

  $\sum_{i=1}^{\infty} f_i p_i + \sum_{j=0}^{\infty} s_j p_j$.

- at time 2,

  it is easy to see that for $i > 1$, the size of the population of age i is equal to $s_{i-1} s_{i-2} p_{i-2}$

  The sizes of the populations having age 0 and 1 have a more complicated description!!

Given a "better" description of the population(s) at all times $t \geq 0$, we might hope to answer questions such as: "is the population decaying or growing?"

4.6 Matrix description

We are now going to give a more compact description of the preceding example, under an additional assumption.

Let's suppose that the lifespan of the populace is no more than 7 time units – i.e. we suppose that $s_7 = 0$.

Under this assumption, we can represent the population at time t by a vector

$$\mathbf{p}^{(t)} = \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_7 \end{bmatrix} \in \mathbb{R}^8$$

If the population at time t is described by $\mathbf{p}^{(t)} = \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_7 \end{bmatrix}$ then the population at time $t+1$ is given by

$$\mathbf{p}^{(t+1)} = \begin{bmatrix} \sum_{i=0}^{7} f_i p_i \\ s_0 p_0 \\ \vdots \\ s_6 p_6 \end{bmatrix} = A\mathbf{p}^{(t)}$$

where

$$A = \begin{bmatrix} f_0 & f_1 & f_2 & \cdots & f_6 & f_7 \\ s_0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & s_1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & s_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & s_6 & 0 \end{bmatrix}.$$

Thus if we begin with population $\mathbf{p}^{(0)} = \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_9 \end{bmatrix}$, then

$$\mathbf{p}^{(1)} = A\mathbf{p}^{(0)}$$

and

$$\mathbf{p}^{(2)} = A\mathbf{p}^{(1)} = A \cdot A \cdot \mathbf{p}^{(0)} = A^2\mathbf{p}^{(0)}$$

.

where A^2 denotes the $A \cdot A$, the *square* or *second power* of the matrix A.

In general, for $j \geq 0$ we have

$$\mathbf{p}^{(j)} = A^j\mathbf{p}^{(0)}$$

Thus computing the long-range behaviour of the system amounts to understanding the powers A^j of the 8×8 matrix A.

In particular, we find the total population at time t by computing

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \cdot A \cdot \mathbf{p}^{(t)}.$$

4.7 Case `fA,sA`

Let's compute several $\begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \cdot A^j$ for a particular A, namely when we make the following assumptions on the f_i and s_i:

```
fA = [.30,.50,.35,.25,.25,.15,.15,.5]
sA = [.30,.60,.55,.50,.30,.15,.05,0]
```

Remeber that for a given population vector $\mathbf{p}$, the resulting population at time j is given by $\begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \cdot A^j \cdot \mathbf{p}$.

[25]:
```
import numpy as np
from pprint import pprint

float_formatter = "{:.4f}".format
np.set_printoptions(formatter={'float_kind':float_formatter})


def sbv(index,size):
    return np.array([1.0 if i == index-1 else 0.0 for i in range(size)])

ones = np.ones(8)

fA = np.array([.30,.50,.35,.25,.25,.15,.15,.5])
sA = np.array([.30,.60,.55,.50,.30,.15,.05,0])

# we use numpy.linalg.matrix_power to compute powers of a matrix

def onePowers(f,s,iter=20,skip=1):
    A = np.concatenate([ [f], [ s[i]*sbv(i,len(f)) for i in range(7)] ], axis =␣
 ↪0)
    s ={ j : ones @ np.linalg.matrix_power(A,j)
              for j in range(0,iter,skip)}
    return s

pprint(onePowers(f=fA,s=sA,iter=35,skip=2))
```

```
{0: array([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000]),
 2: array([0.7800, 0.7525, 0.5150, 0.2850, 0.2475, 0.1600, 0.1350, 0.4500]),
 4: array([0.4009, 0.4052, 0.2977, 0.1776, 0.1533, 0.1009, 0.0814, 0.2715]),
 6: array([0.2199, 0.2212, 0.1608, 0.0968, 0.0837, 0.0549, 0.0448, 0.1494]),
 8: array([0.1200, 0.1205, 0.0877, 0.0528, 0.0456, 0.0299, 0.0244, 0.0812]),
 10: array([0.0655, 0.0657, 0.0479, 0.0288, 0.0249, 0.0163, 0.0133, 0.0443]),
 12: array([0.0357, 0.0359, 0.0261, 0.0157, 0.0136, 0.0089, 0.0073, 0.0242]),
 14: array([0.0195, 0.0196, 0.0142, 0.0086, 0.0074, 0.0048, 0.0040, 0.0132]),
 16: array([0.0106, 0.0107, 0.0078, 0.0047, 0.0040, 0.0026, 0.0022, 0.0072]),
 18: array([0.0058, 0.0058, 0.0042, 0.0025, 0.0022, 0.0014, 0.0012, 0.0039]),
 20: array([0.0032, 0.0032, 0.0023, 0.0014, 0.0012, 0.0008, 0.0006, 0.0021]),
 22: array([0.0017, 0.0017, 0.0013, 0.0008, 0.0007, 0.0004, 0.0004, 0.0012]),
 24: array([0.0009, 0.0009, 0.0007, 0.0004, 0.0004, 0.0002, 0.0002, 0.0006]),
 26: array([0.0005, 0.0005, 0.0004, 0.0002, 0.0002, 0.0001, 0.0001, 0.0003]),
 28: array([0.0003, 0.0003, 0.0002, 0.0001, 0.0001, 0.0001, 0.0001, 0.0002]),
 30: array([0.0002, 0.0002, 0.0001, 0.0001, 0.0001, 0.0000, 0.0000, 0.0001]),
 32: array([0.0001, 0.0001, 0.0001, 0.0000, 0.0000, 0.0000, 0.0000, 0.0001]),
 34: array([0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000])}
```

The calculations above suggetss that

$$\begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix} \cdot A^j = \mathbf{0} \quad \text{for} \quad j \geq 34.$$

More precisely, it suggests that the entries of `[1,1,...,1] @ A^j` are 0 to 4 decimal places for `j >= 20`.

Thus with the given matrix A, the model suggests that the total population will decay to 0.

[28]:
```
def computePops(apowers,pop):
    return {j: apowers[j] @ pop for j in apowers.keys()}

p = 10*sbv(1,8)
computePops(onePowers(f=fA,s=sA,iter=35,skip=2),p)
```

[28]:
```
{0: 10.0,
 2: 7.800000000000001,
 4: 4.0095,
 6: 2.1985649999999994,
 8: 1.2003113249999997,
 10: 0.65464922925,
 12: 0.3571069035037499,
 14: 0.19480016616704995,
 16: 0.10626220632051167,
 18: 0.057965348421534224,
 20: 0.031619725246313,
 22: 0.017248356865952965,
 24: 0.00940886779550352,
 26: 0.0051324769018262755,
 28: 0.002799733158111125,
 30: 0.0015272364409127217,
 32: 0.000833097661359451,
 34: 0.00045444941907475374}
```

5 Case `fB,sB`

Now let's consider different probabilities, as follows:

```
fB = [.50,.70,.55,.35,.35,.15,.15,.5]
sB = [.40,.70,.55,.50,.35,.15,.05,0]
```

[10]:
```
fB = [.50,.70,.55,.35,.35,.15,.15,.5]
sB = [.40,.70,.55,.50,.35,.15,.05,0]

pprint(onePowers(f=fB,s=sB,iter=20))
```

```
{0: array([1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00, 1.00]),
 1: array([1.20, 1.25, 1.05, 0.70, 0.50, 0.20, 0.15, 0.50]),
 2: array([1.33, 1.23, 0.91, 0.49, 0.44, 0.21, 0.18, 0.60]),
 3: array([1.30, 1.20, 0.96, 0.54, 0.49, 0.23, 0.20, 0.67]),
 4: array([1.32, 1.21, 0.96, 0.54, 0.49, 0.23, 0.20, 0.65]),
 5: array([1.34, 1.22, 0.97, 0.54, 0.49, 0.23, 0.20, 0.66]),
 6: array([1.35, 1.23, 0.98, 0.55, 0.50, 0.23, 0.20, 0.67]),
```

```
 7: array([1.36, 1.24, 0.99, 0.55, 0.50, 0.24, 0.20, 0.67]),
 8: array([1.37, 1.26, 1.00, 0.56, 0.51, 0.24, 0.20, 0.68]),
 9: array([1.39, 1.27, 1.01, 0.56, 0.51, 0.24, 0.21, 0.69]),
 10: array([1.40, 1.28, 1.02, 0.57, 0.52, 0.24, 0.21, 0.69]),
 11: array([1.41, 1.29, 1.03, 0.57, 0.52, 0.24, 0.21, 0.70]),
 12: array([1.42, 1.30, 1.04, 0.58, 0.53, 0.25, 0.21, 0.71]),
 13: array([1.44, 1.32, 1.05, 0.58, 0.53, 0.25, 0.21, 0.71]),
 14: array([1.45, 1.33, 1.06, 0.59, 0.54, 0.25, 0.22, 0.72]),
 15: array([1.46, 1.34, 1.07, 0.60, 0.54, 0.25, 0.22, 0.73]),
 16: array([1.48, 1.35, 1.08, 0.60, 0.55, 0.26, 0.22, 0.73]),
 17: array([1.49, 1.37, 1.09, 0.61, 0.55, 0.26, 0.22, 0.74]),
 18: array([1.51, 1.38, 1.10, 0.61, 0.56, 0.26, 0.22, 0.75]),
 19: array([1.52, 1.39, 1.11, 0.62, 0.56, 0.26, 0.23, 0.75])}
```

In this case, note that the first entry of the vector

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} \cdot A^j$$

appears to be an increasing function of j.

Thus, for example we expect that given an initial population $\mathbf{p}^{(0)}$ with $p_0 > 0$, the total population is increasing as a function of j, rather than decaying.

[29]:
```
p = 10*sbv(1,8)
computePops(onePowers(f=fB,s=sB,iter=35,skip=2),p)
```

[29]:
```
{0: 10.0,
 2: 13.35,
 4: 13.210999999999999,
 6: 13.46822875,
 8: 13.72457581875,
 10: 13.982033212249998,
 12: 14.244391579877968,
 14: 14.511705408818932,
 16: 14.784034904839313,
 18: 15.061474708657759,
 20: 15.344121005426935,
 22: 15.632071494972397,
 24: 15.925425714260237,
 26: 16.224285070706642,
 28: 16.528752874710595,
 30: 16.838934375390473,
 32: 17.154936796988775,
 34: 17.47686937593729}
```

[ ]: