Register Allocation, ii

Graph coloring

Graph Coloring

Idea: reduce register allocation to graph coloring

- Create an interference graph for each program where:
  - nodes are variables
  - edges connect variables that cannot be in the same register
- Color the nodes in the graph such that connected nodes have different colors
- Each color represents a register; after coloring, read variable → register assignment from graph

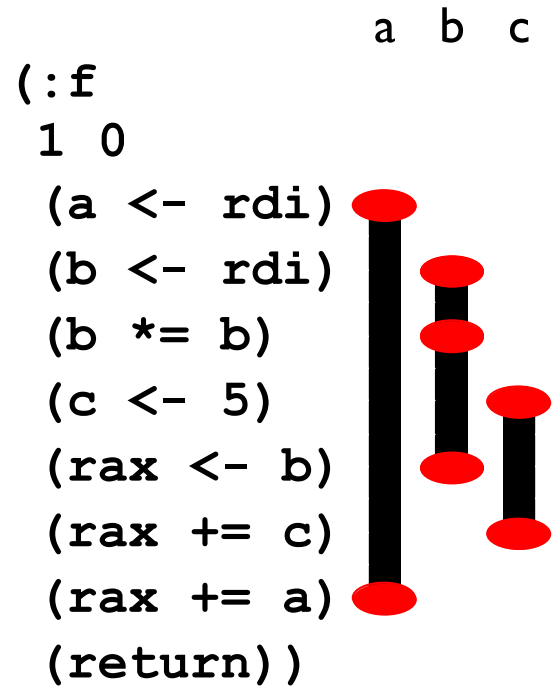
Function with three variables, **a**, **b**, and **c**:

f(x) = x^2 + x + 5

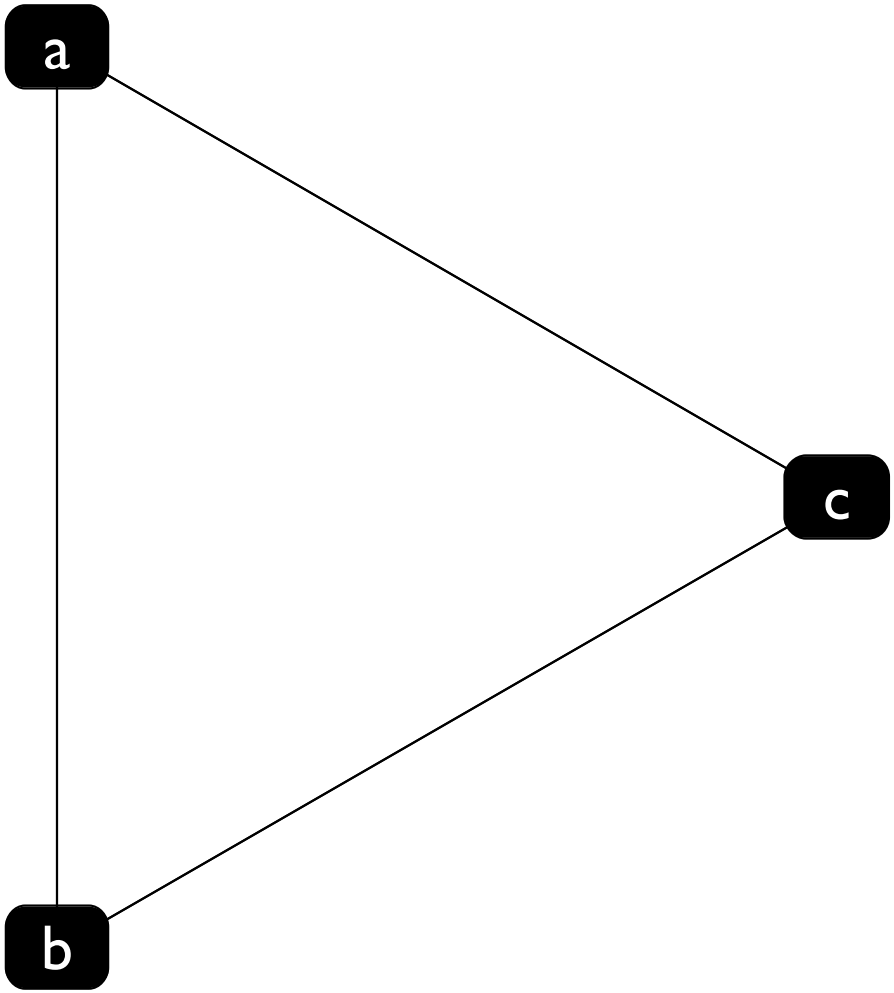
```
(:f
 1 0
 (a <- rdi)
 (b <- rdi)
 (b *= b)
 (c <- 5)
 (rax <- b)
 (rax += c)
 (rax += a)
 (return))
```

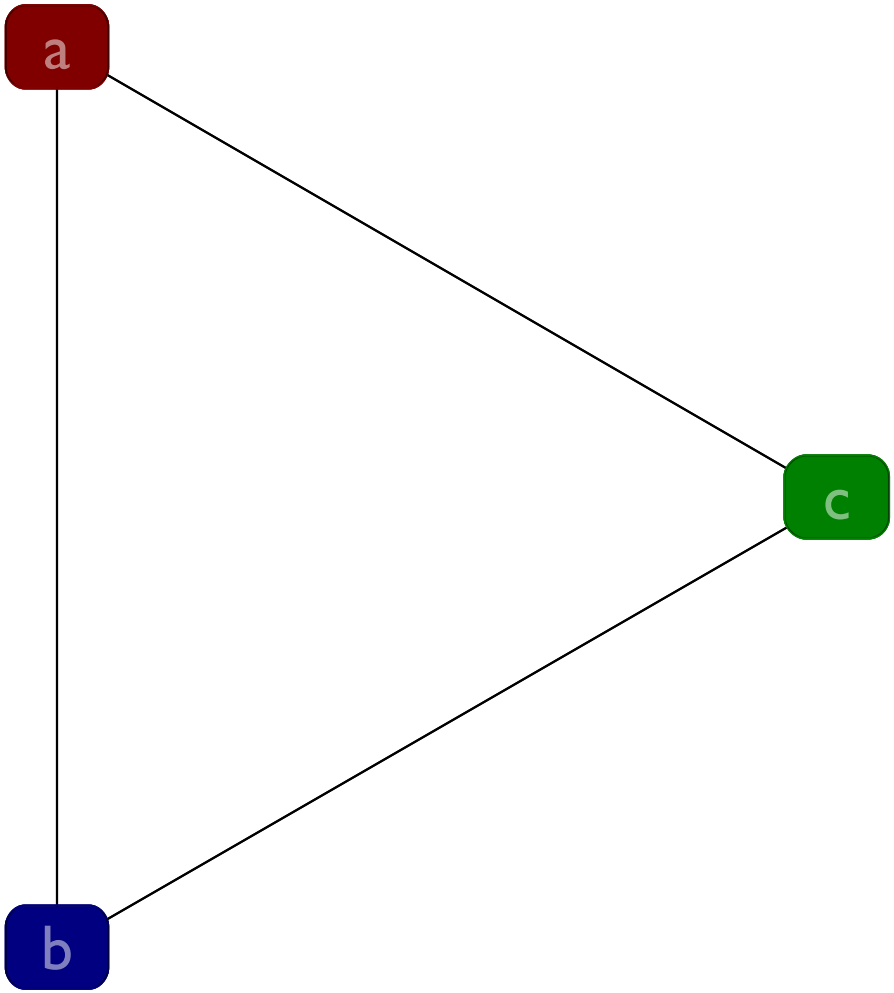
Function with three variables, **a**, **b**, and **c**:

```
f(x) = x^2 + x + 5
```



Black bars show the live ranges of the variables


a
c
b


a
c
b

Three variables, and we needed three registers because we need three colors to color the graph

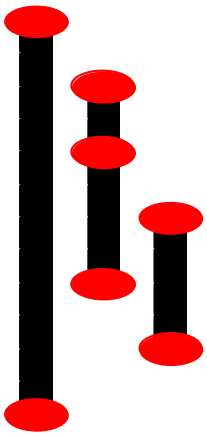
Different implementation of same function

```
(:f
 1 0
 (a <- rdi)
 (b <- rdi)
 (b *= b)
 (c <- 5)
 (rax <- b)
 (rax += c)
 (rax += a)
 (return))
```

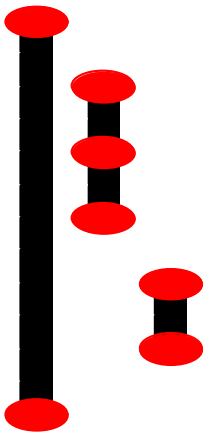
```
(:f
 1 0
 (a <- rdi)
 (b <- rdi)
 (b *= b)
 (rax <- b)
 (c <- 5)
 (rax += c)
 (rax += a)
 (return))
```

Different implementation of same function

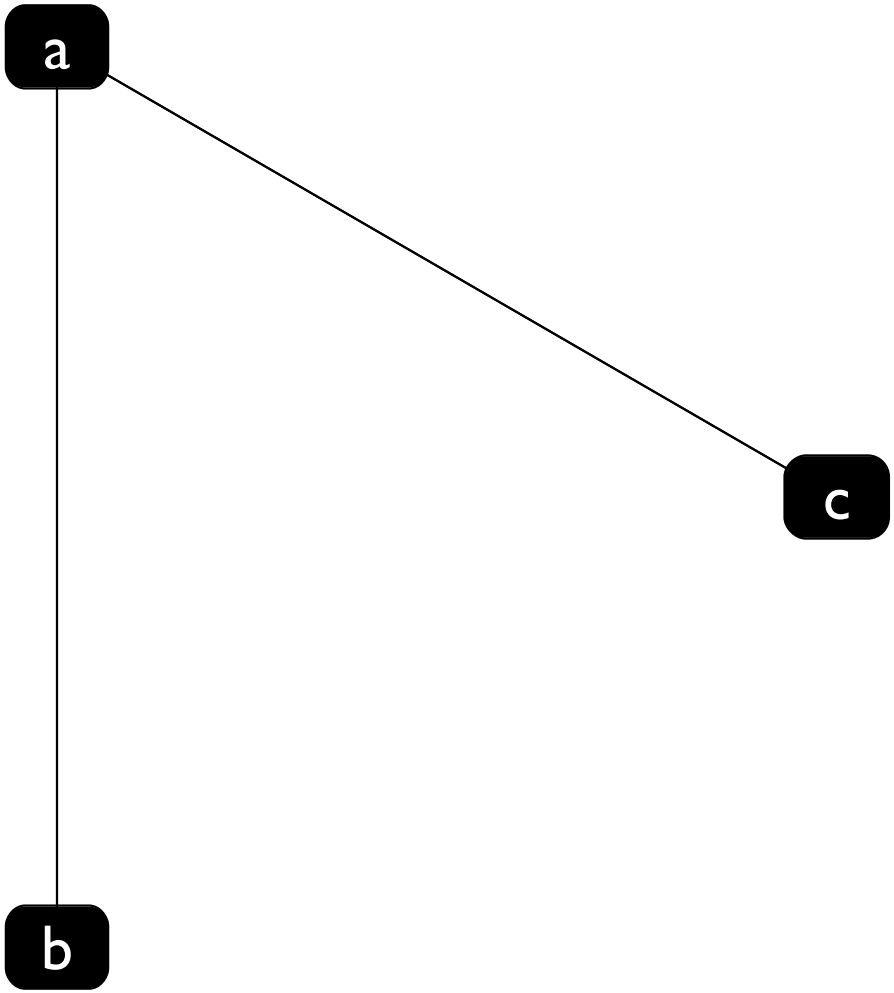
```
                a  b  c
(:f
 1  0
 (a <- rdi)
 (b <- rdi)
 (b *= b)
 (c <- 5)
 (rax <- b)
 (rax += c)
 (rax += a)
 (return))
```

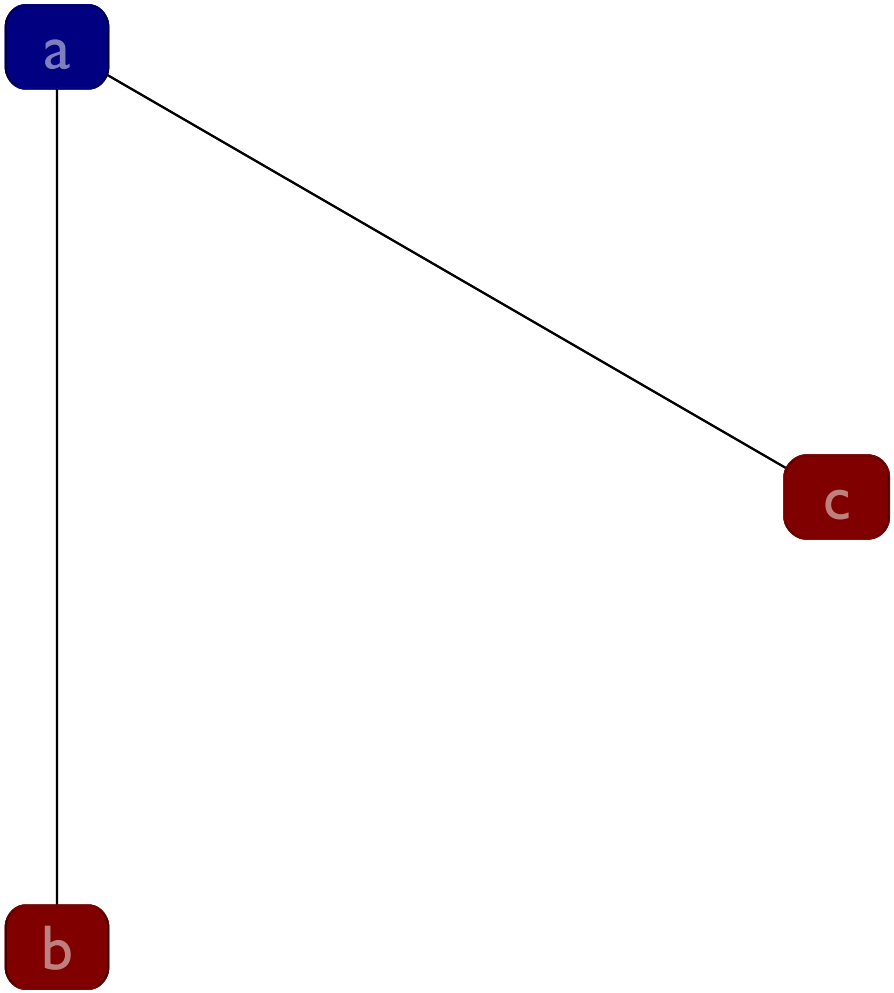


```
                a  b  c
(:f
 1  0
 (a <- rdi)
 (b <- rdi)
 (b *= b)
 (rax <- b)
 (c <- 5)
 (rax += c)
 (rax += a)
 (return))
```



Less overlap in the live ranges of the variables; now we need fewer colors


a
c
b


a
c
b

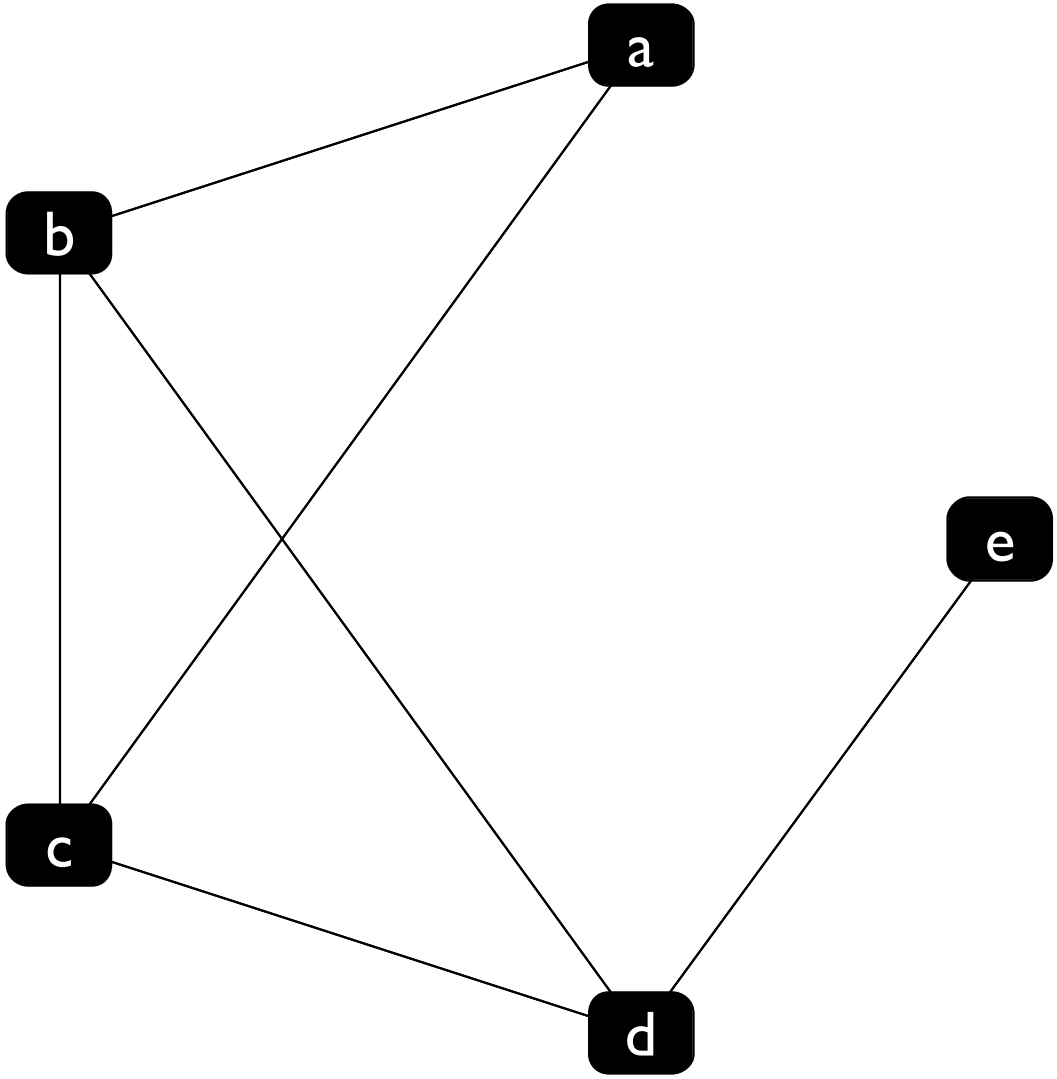
Three variables, but we needed only two registers because we need only two colors to color the graph

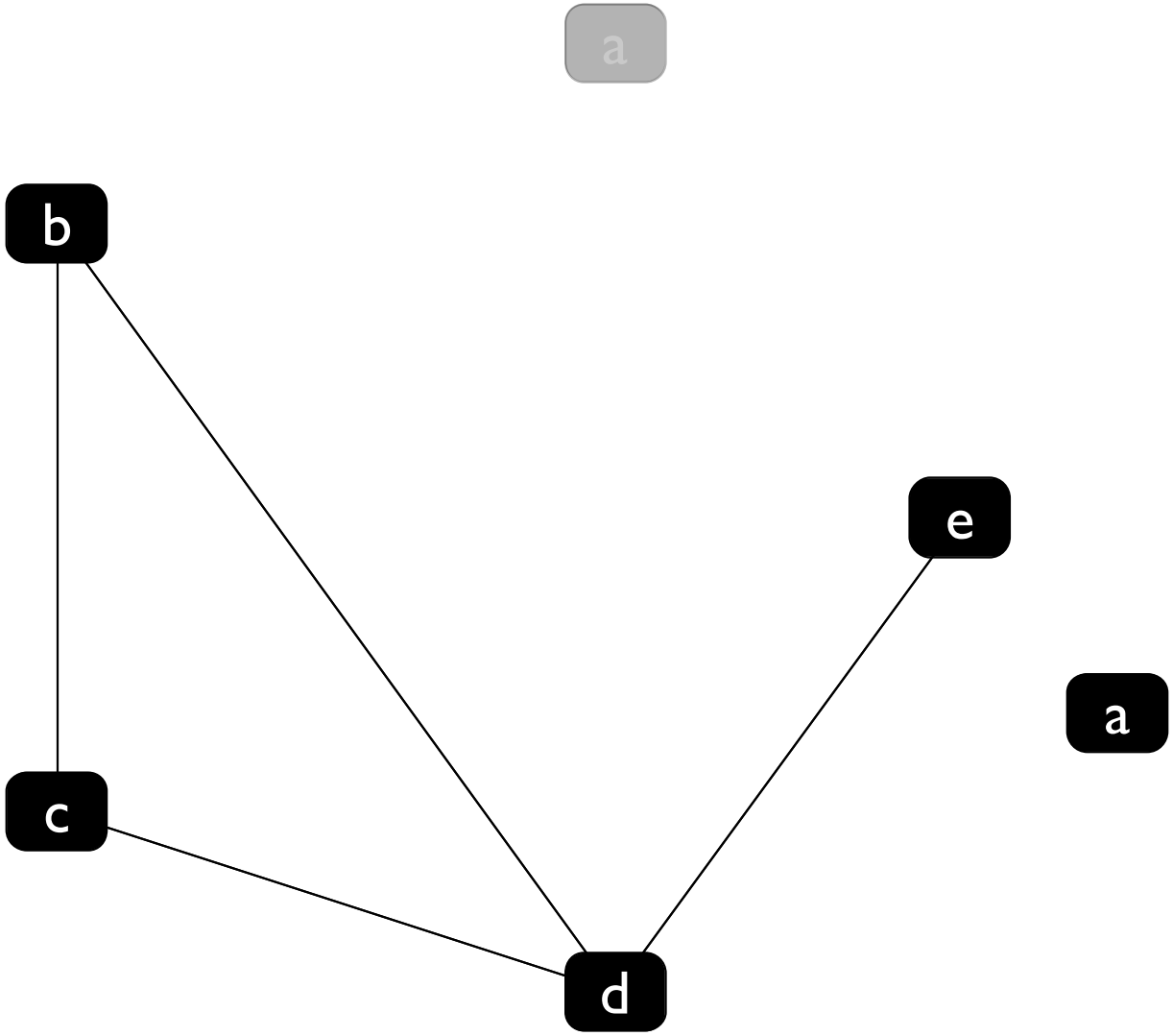
This makes sense because the live ranges of **`b`** and **`c`** didn't overlap, so they can share a register

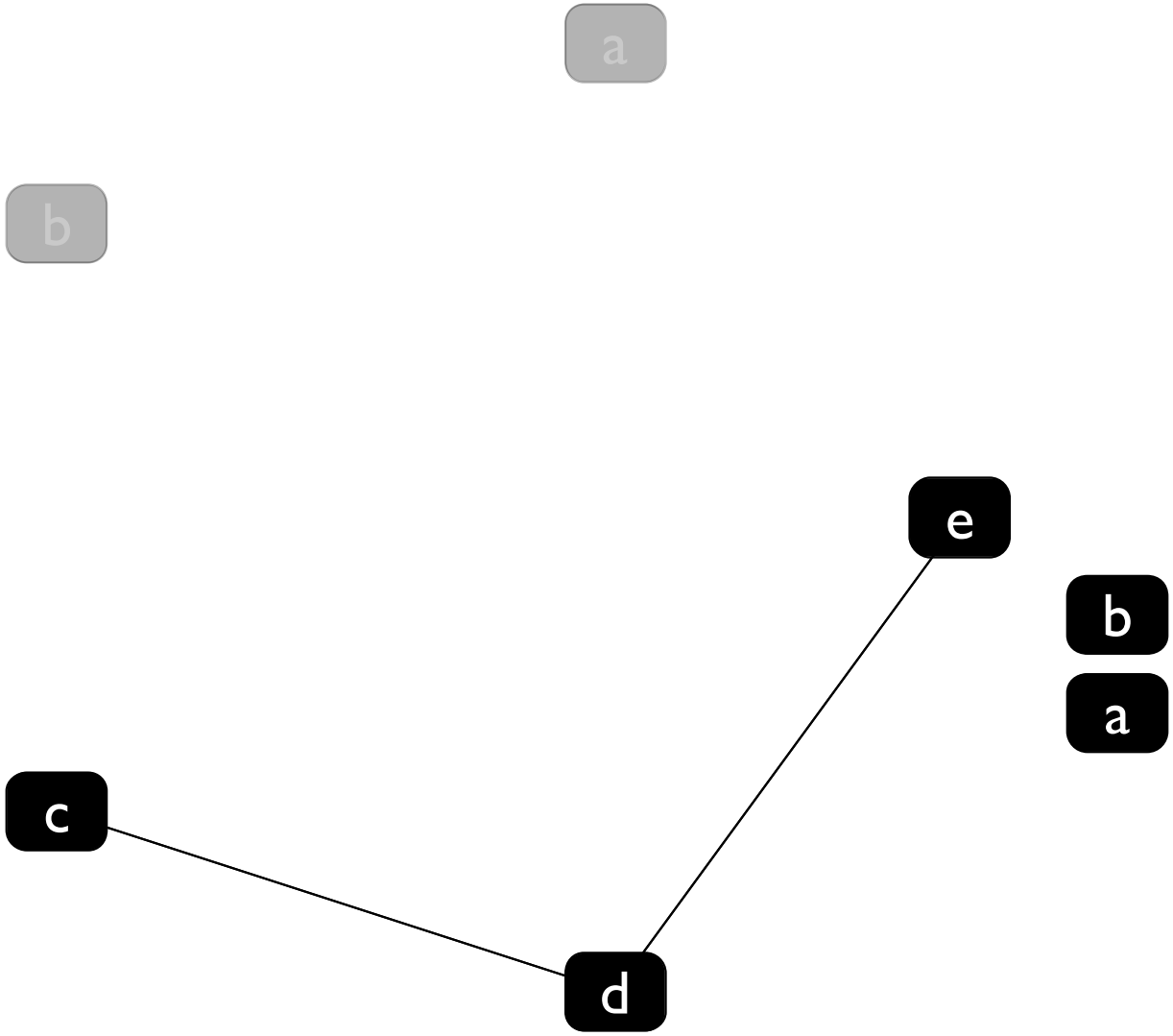
NB: Those graphs are not accurate: they leave out the constraints imposed by the calling convention. We return to this later.

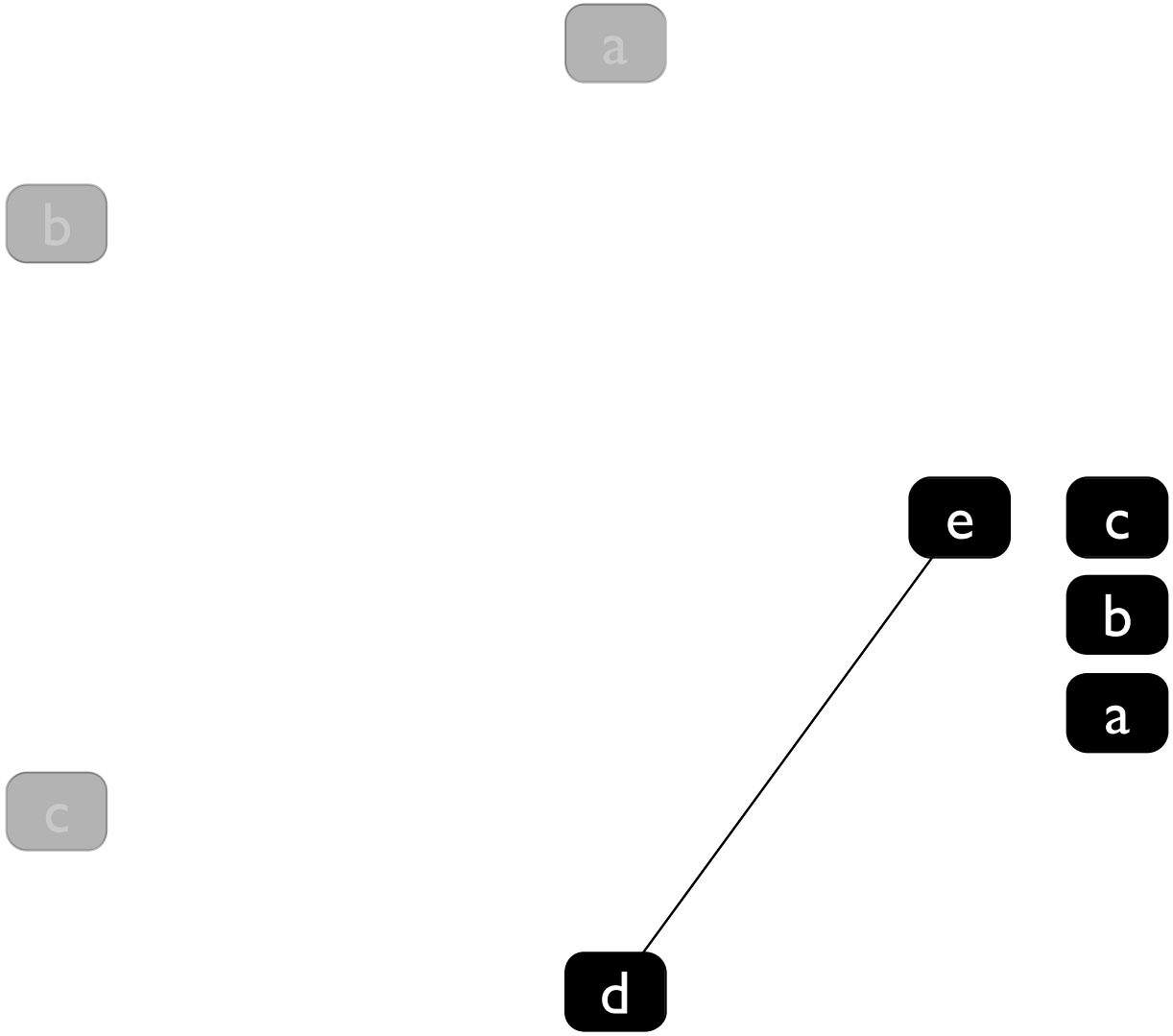
Graph coloring algorithm:

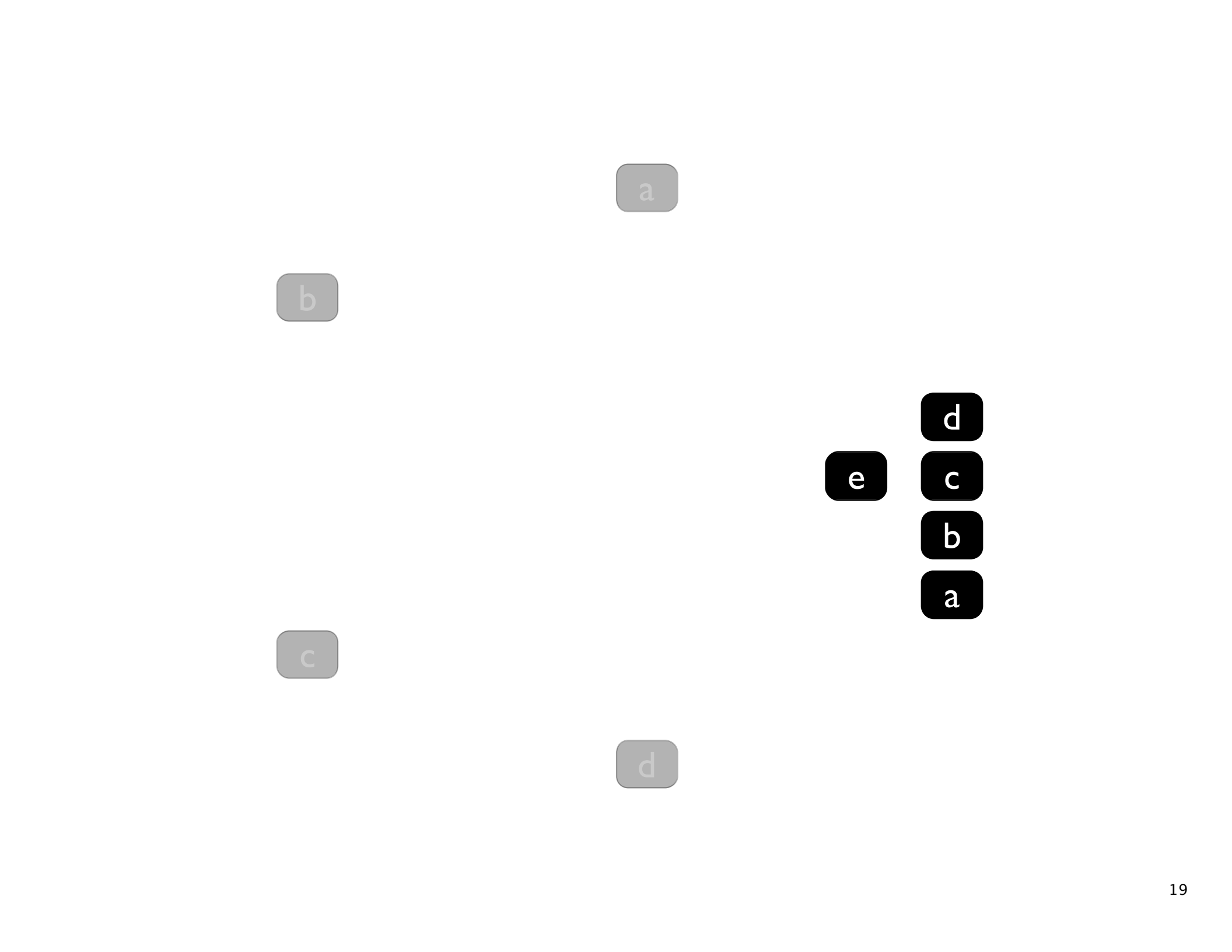
- Repeatedly select a node and remove it from the graph, putting it onto a stack
- When the graph is empty, rebuild it, putting a new color on each node as it comes back into the graph, making sure no adjacent nodes have the same color
- Order the colors; pick the smallest color that isn't a neighbor, use that one
- If there are not enough colors, the algorithm fails (spilling comes in here)

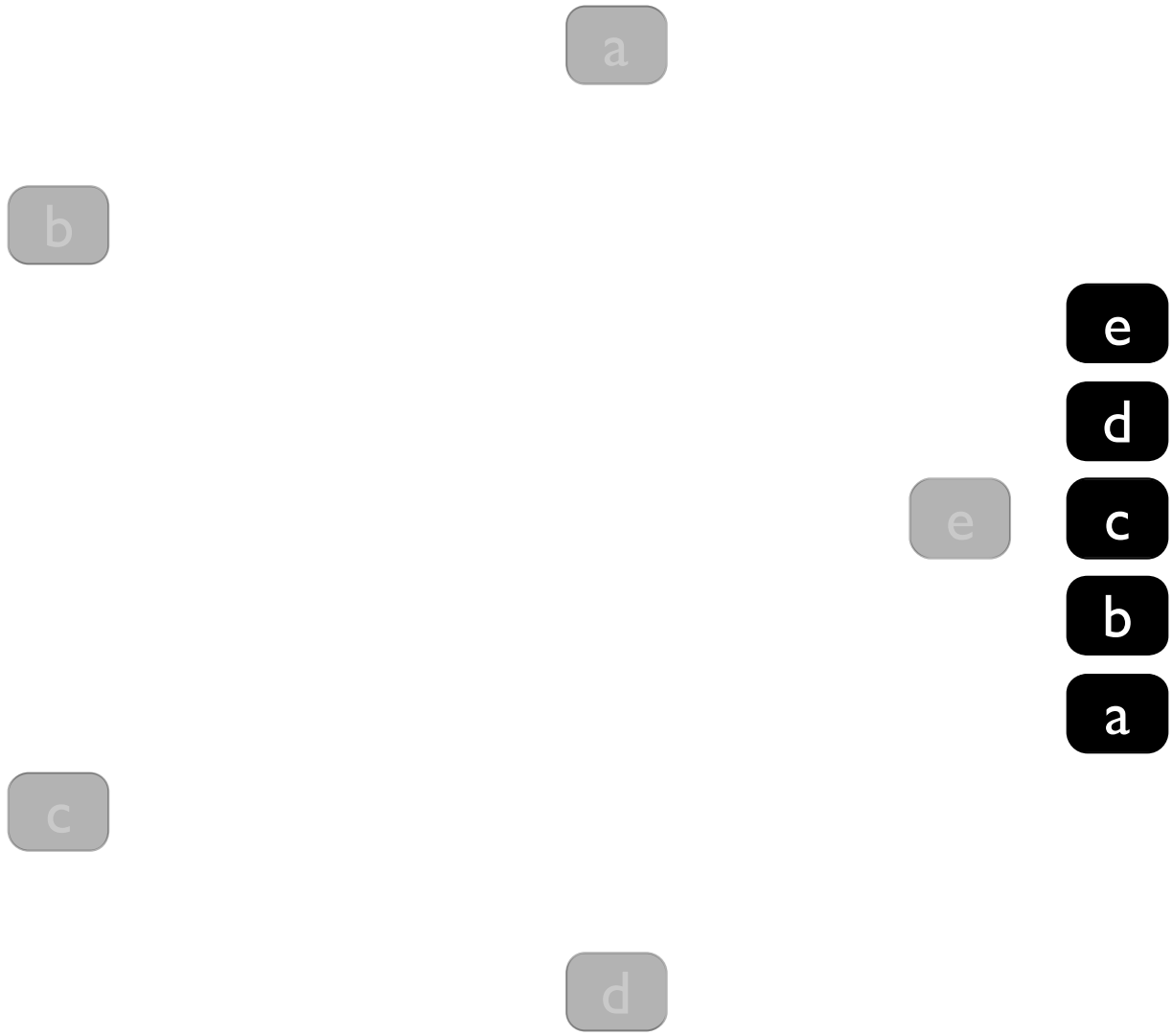

a
b
e
c
d

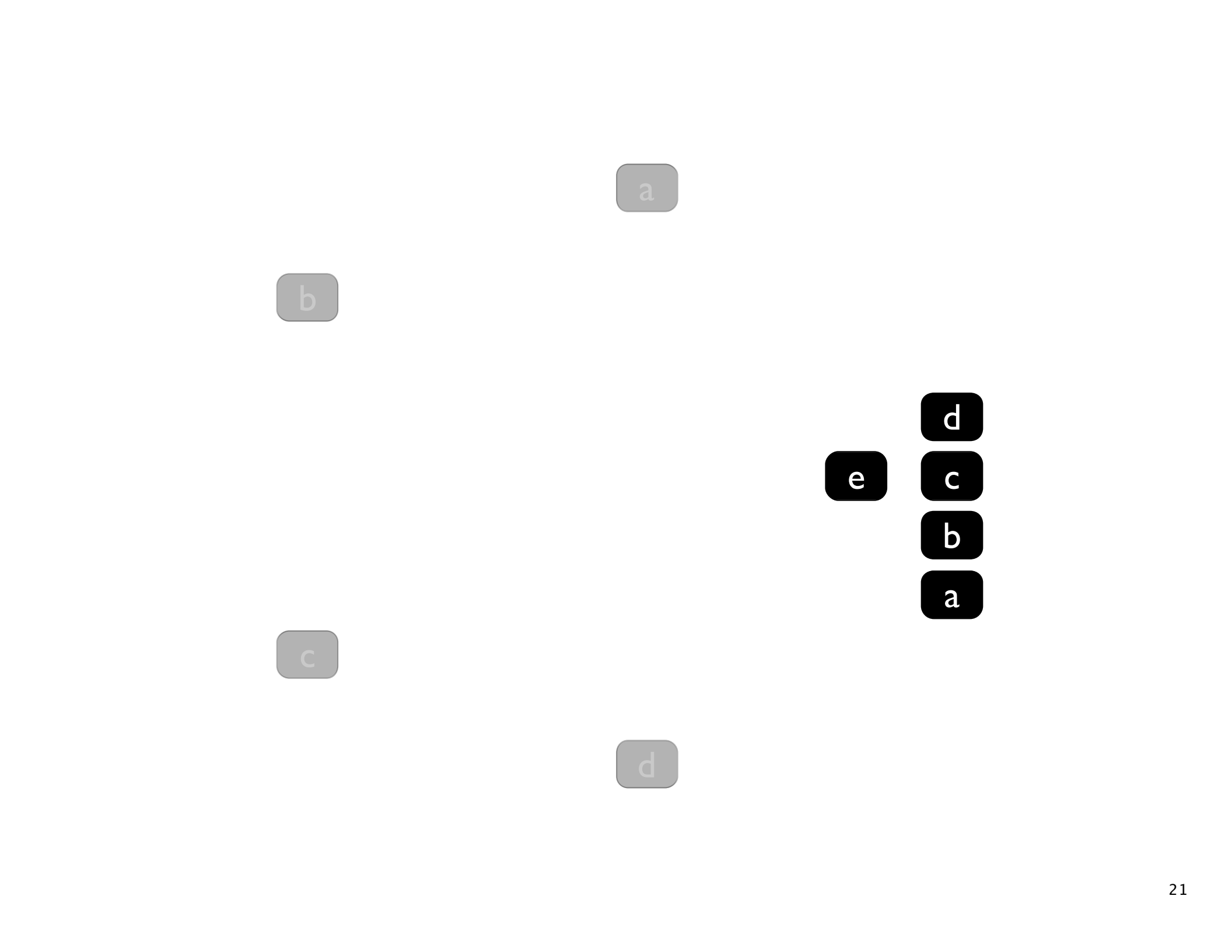

a
b
e
a
c
d

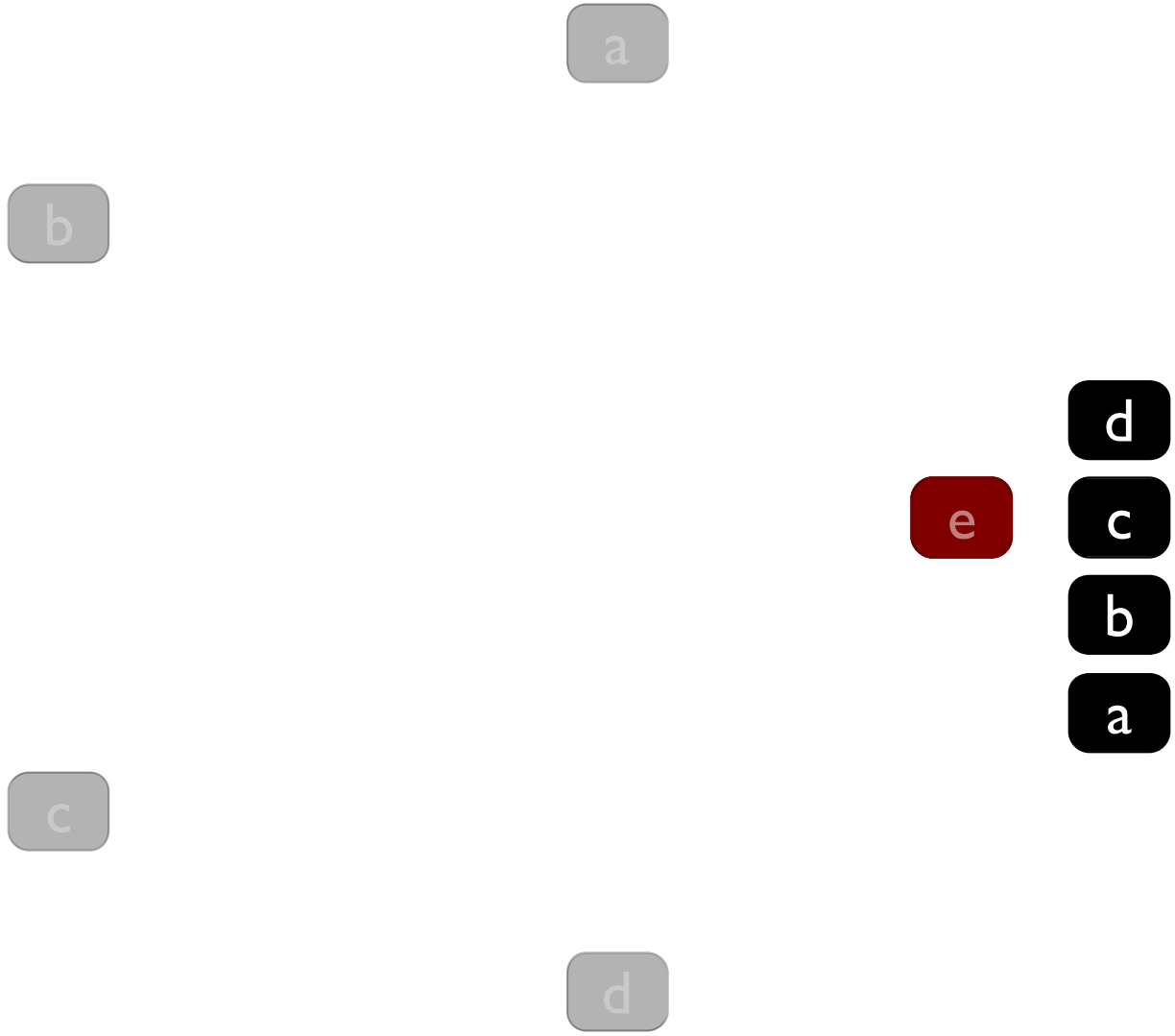

a
b
e
b
a
c
d

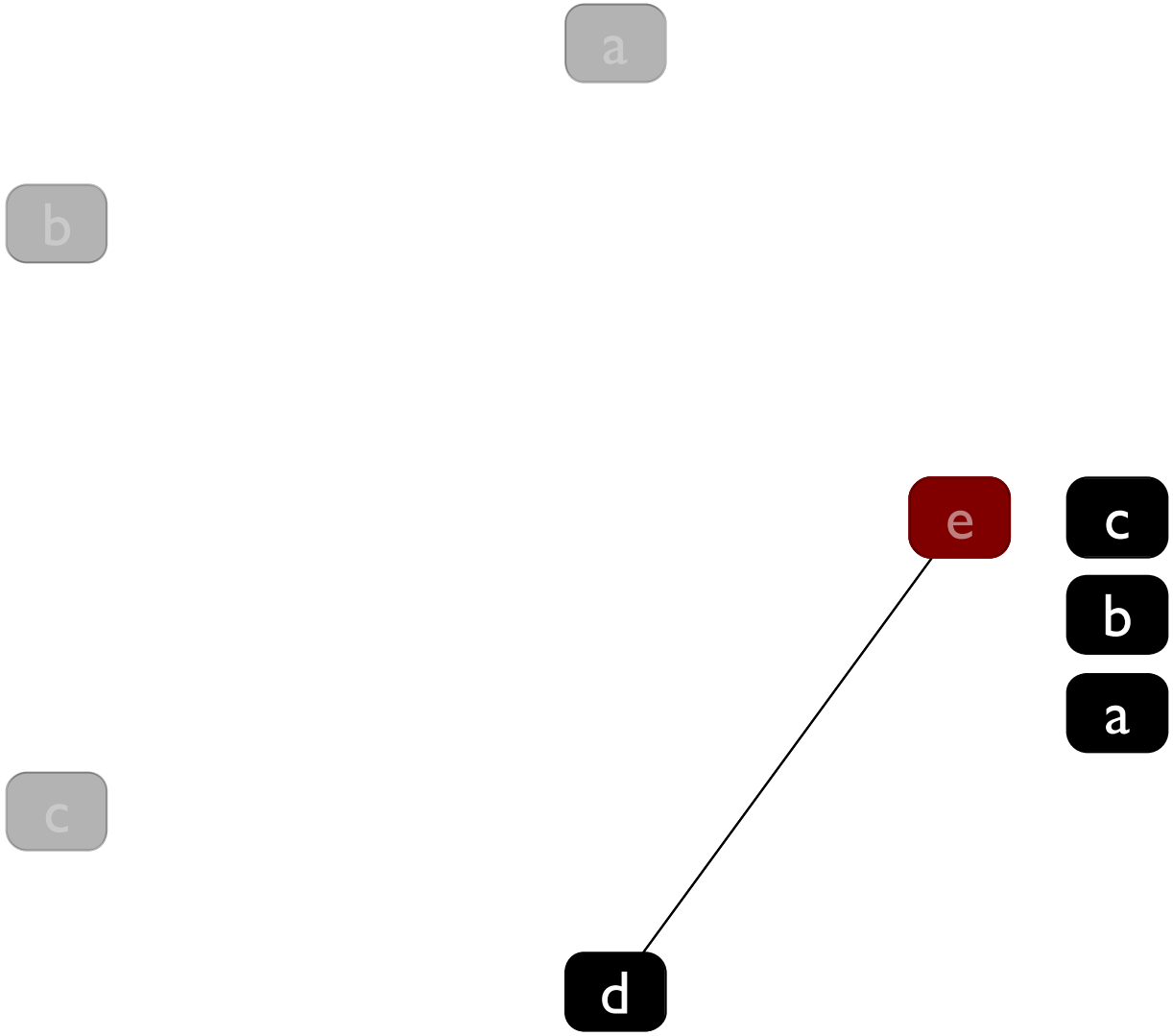

a
b
c
e
c
b
a
d

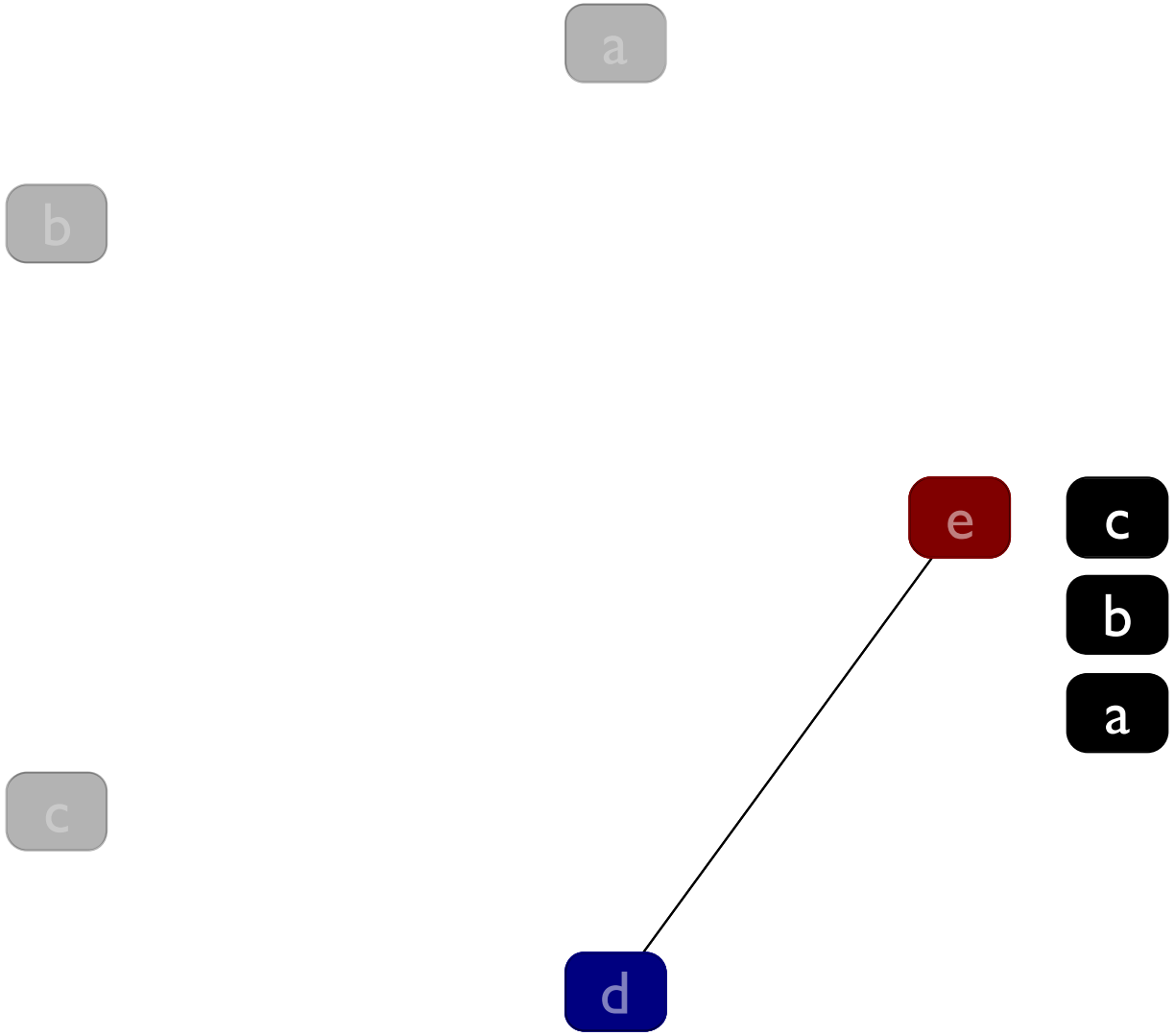

a
b
c
d
e
d
c
b
a

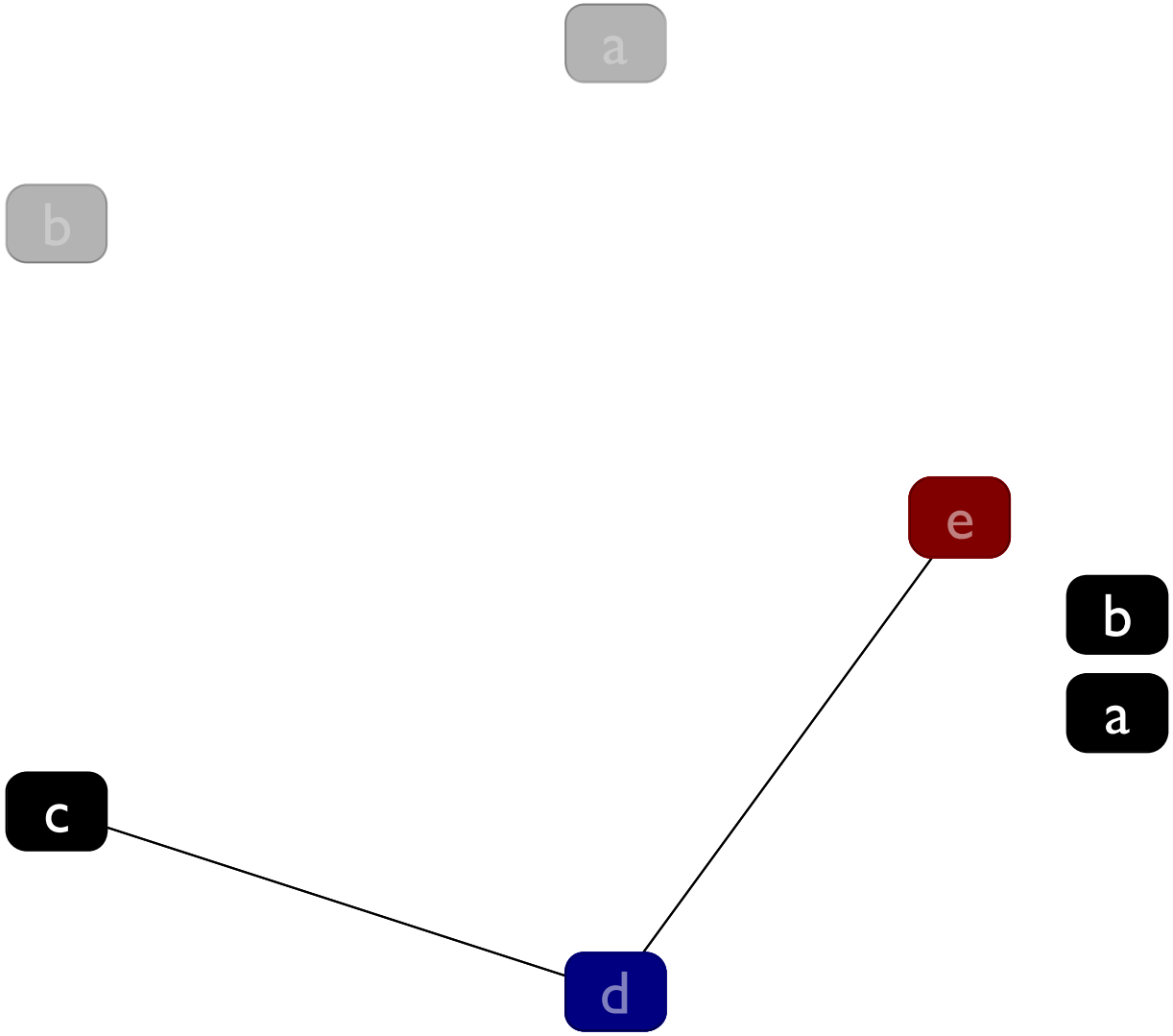

a
b
e
d
c
b
a
e
c
d

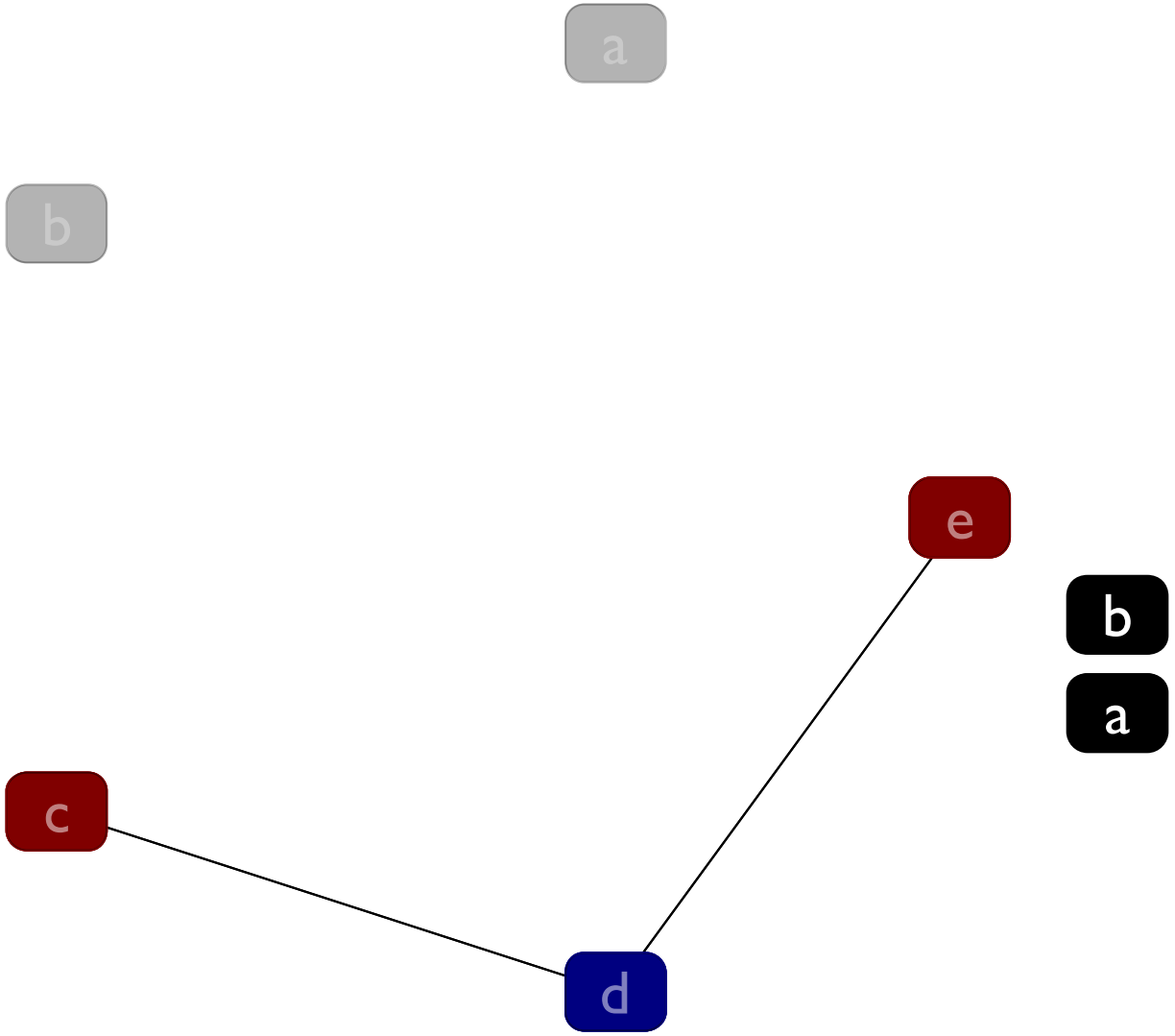

a
b
c
d
e
d
c
b
a

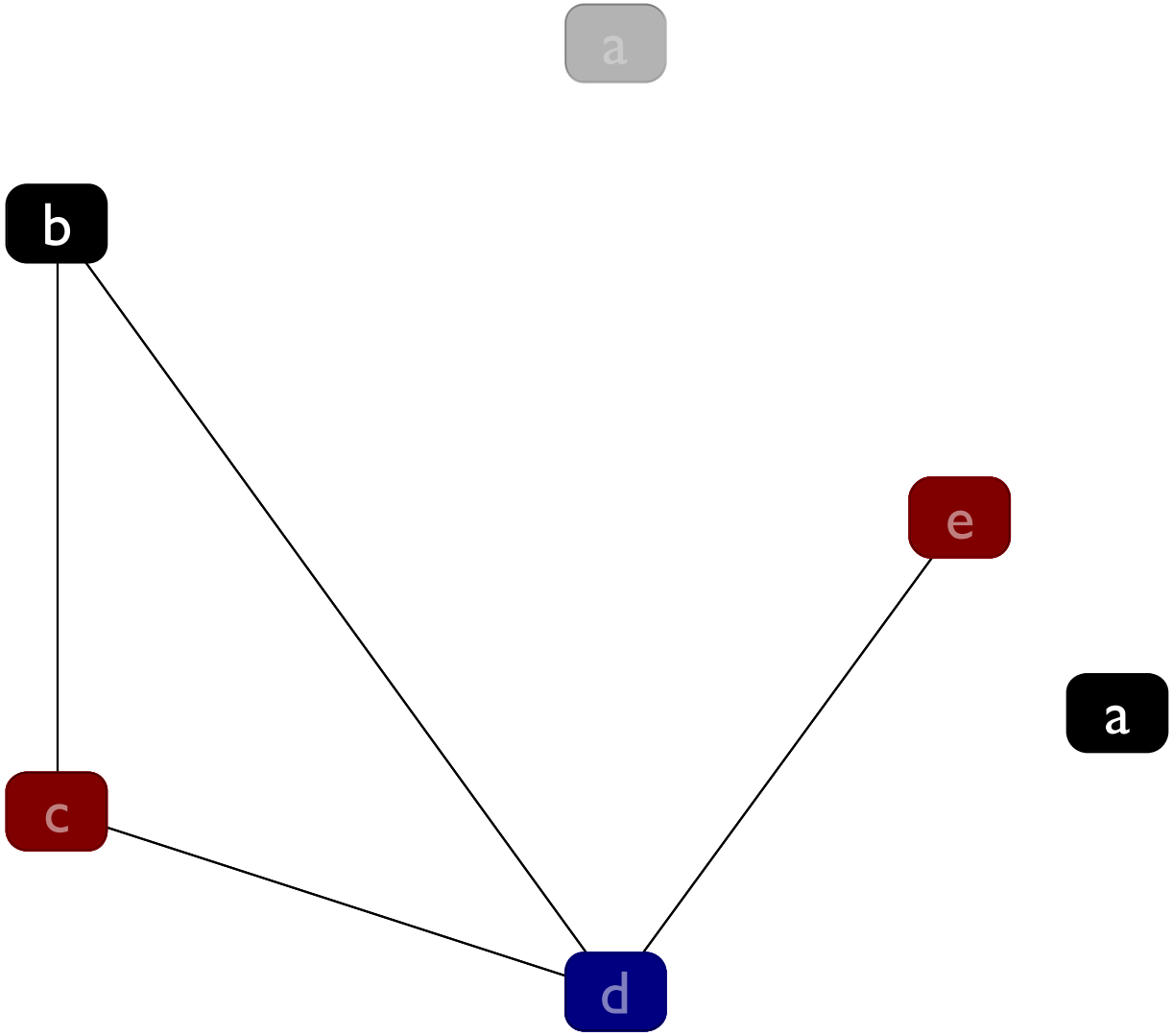

a
b
c
d
e
d
c
b
a

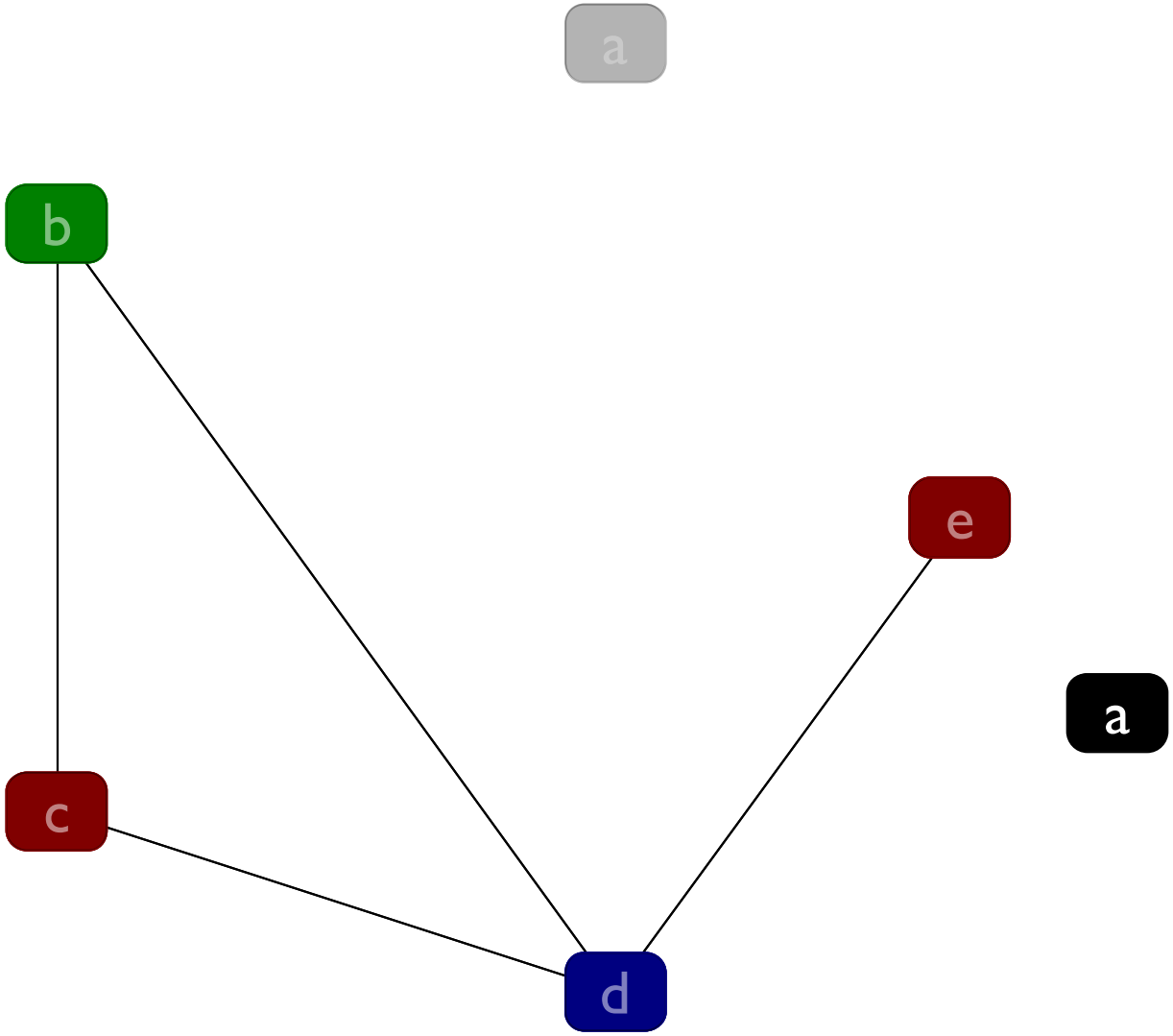

a
b
c
e
d
c
b
a

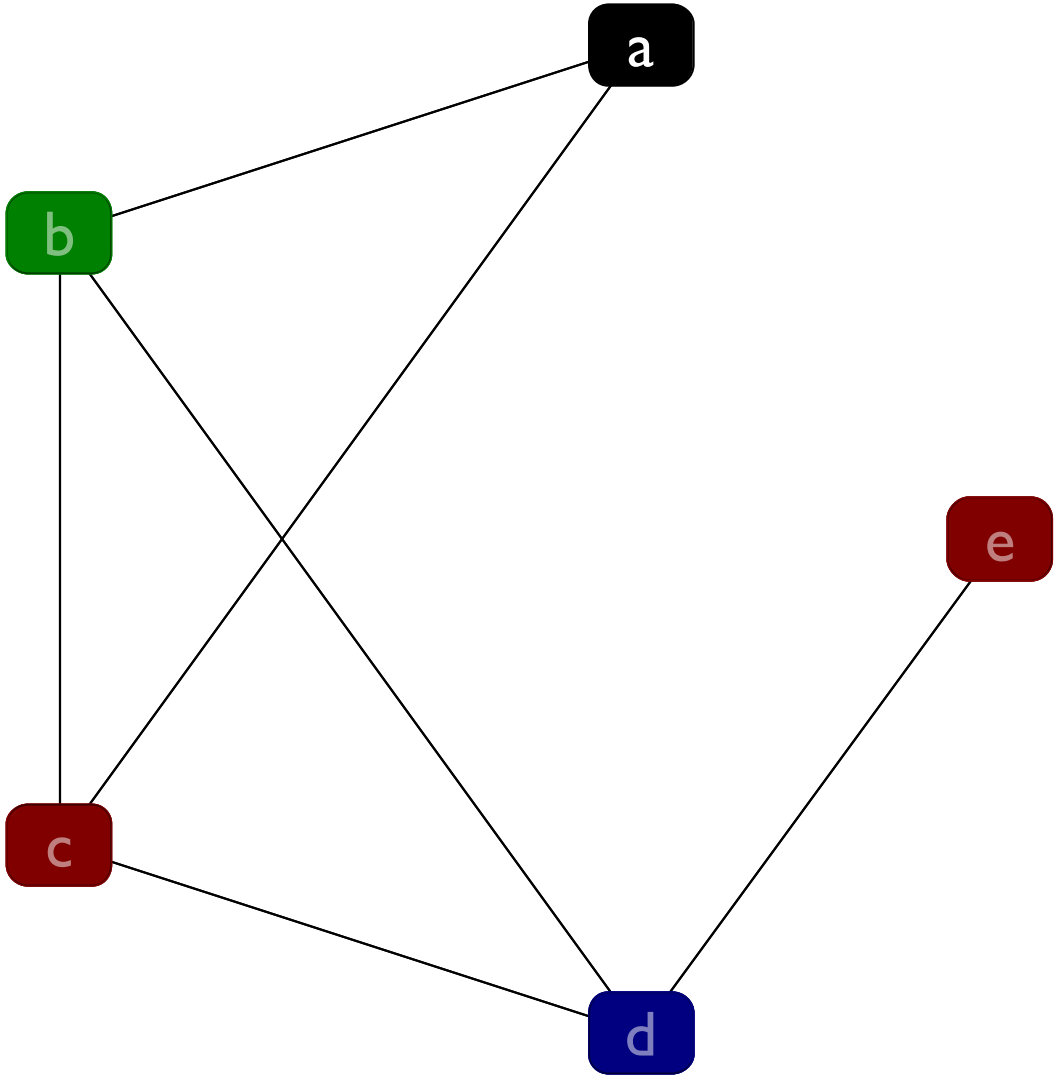

a
b
c
e
d
c
b
a

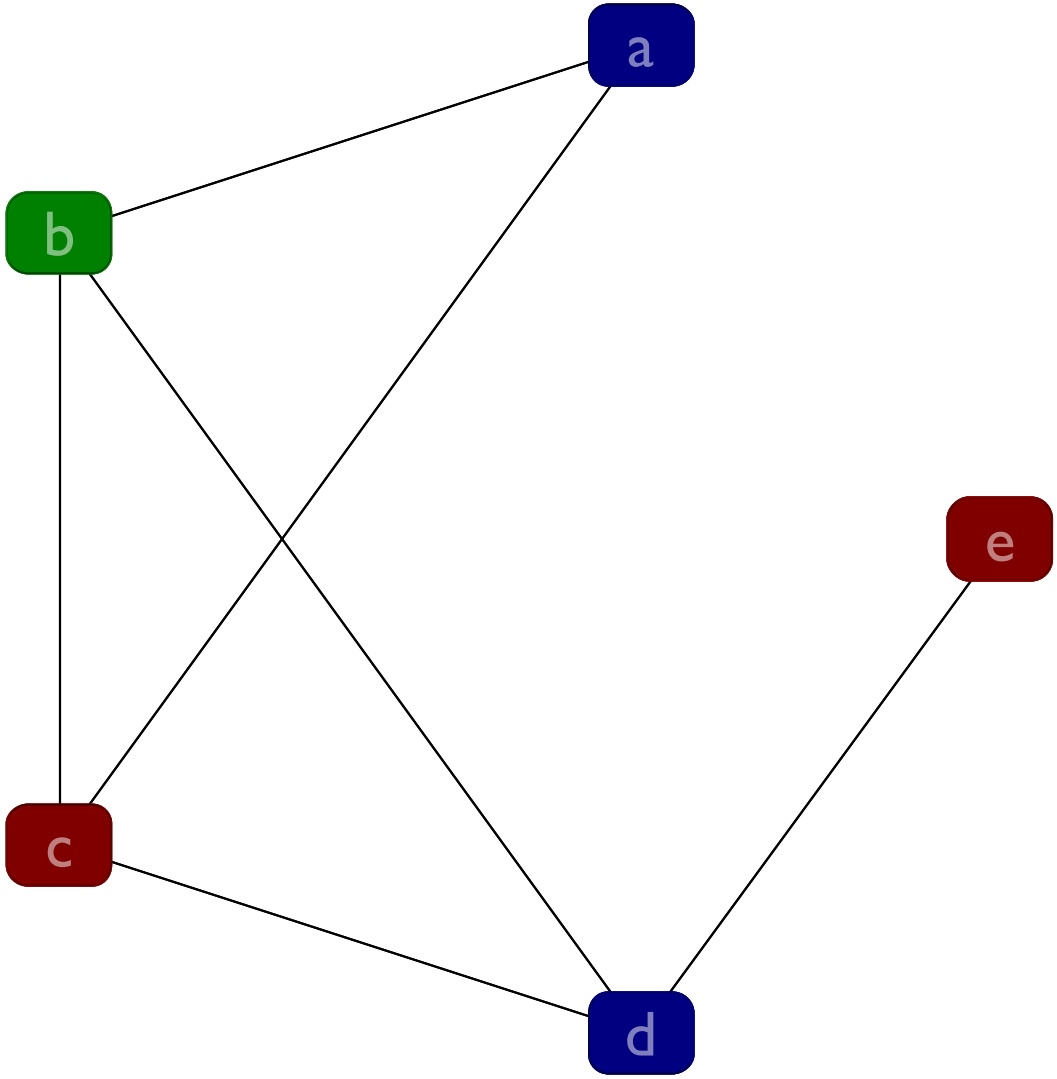

a
b
e
b
a
c
d


a
b
e
b
a
c
d


a
b
e
a
c
d

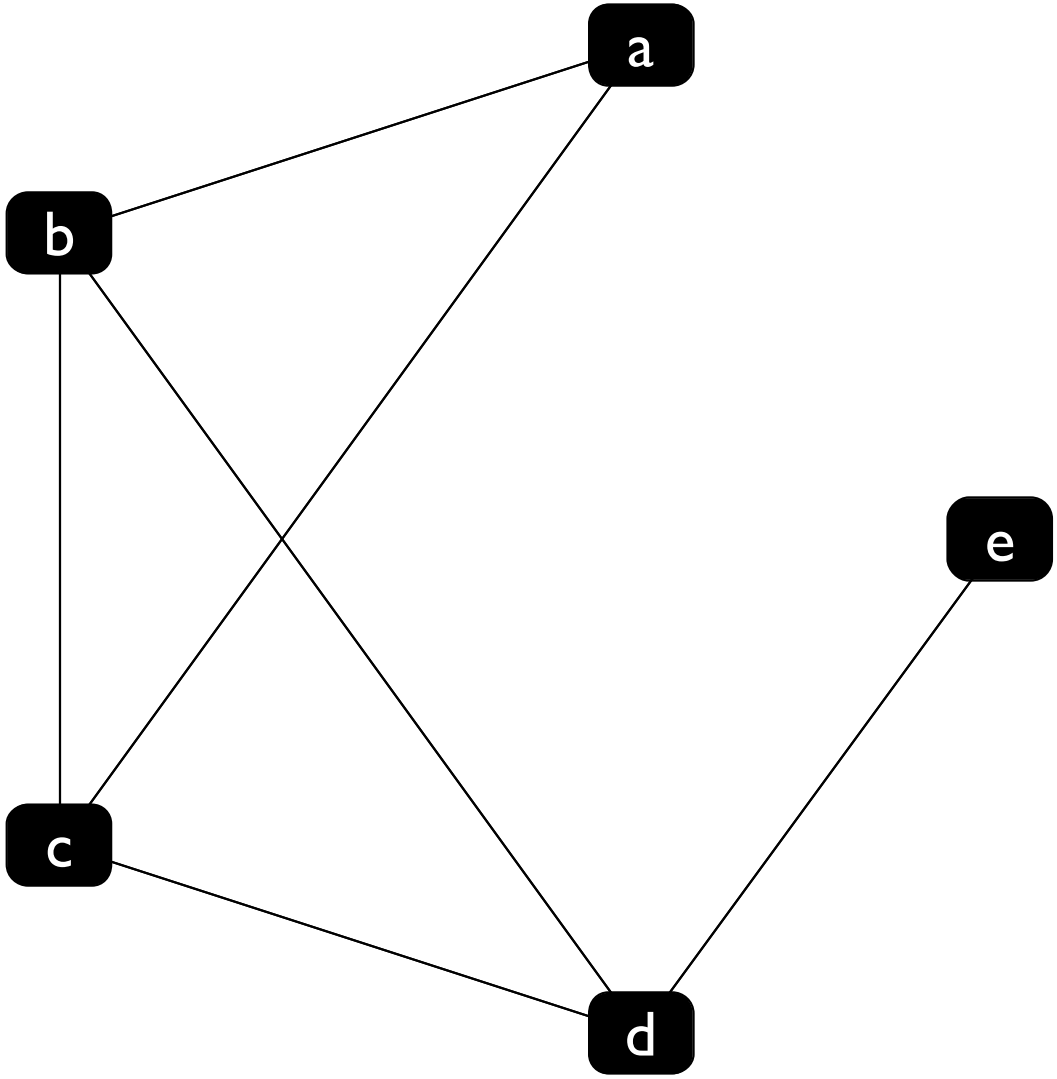

a
b
e
a
c
d

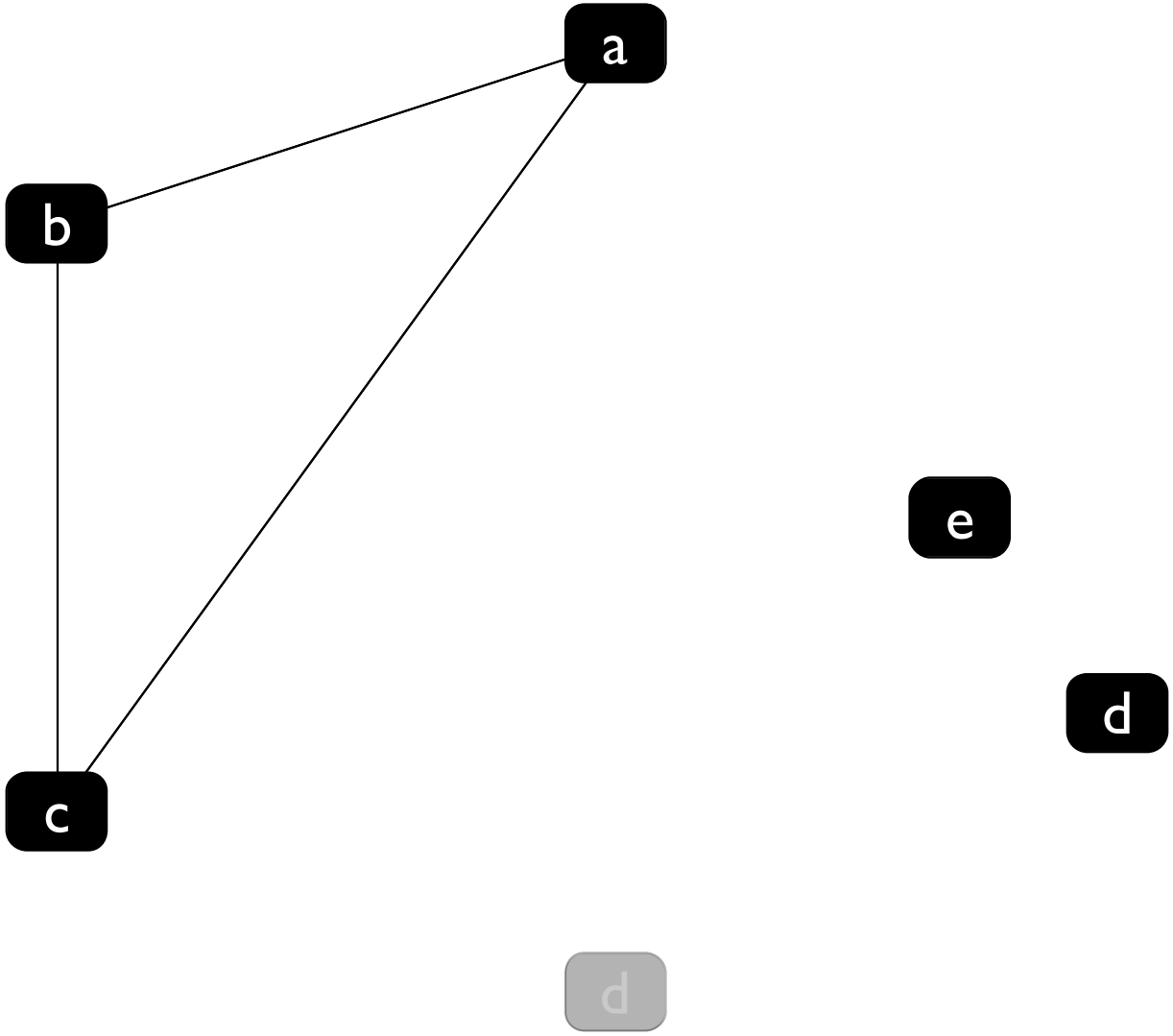

a
b
e
c
d

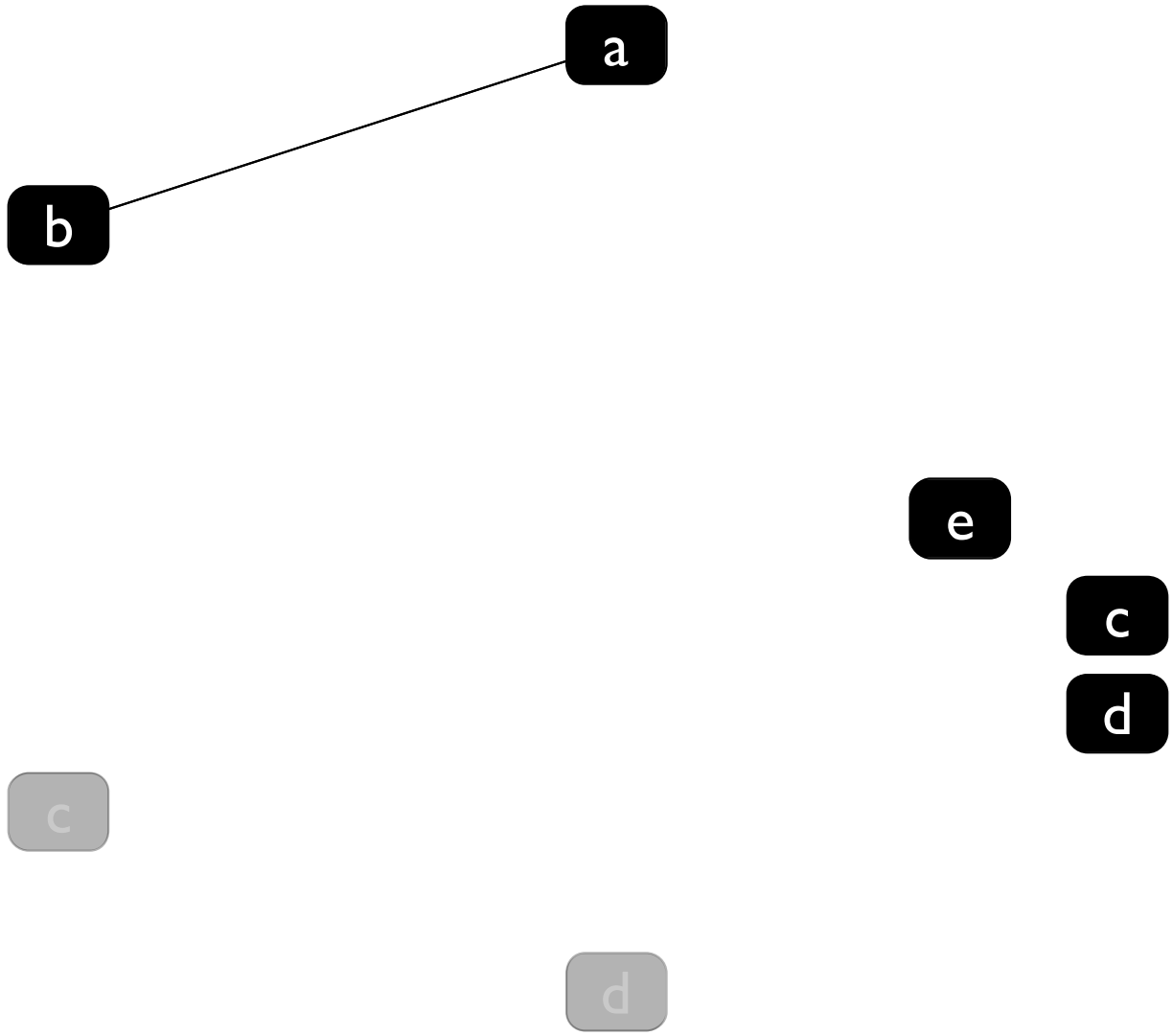

a
b
e
c
d

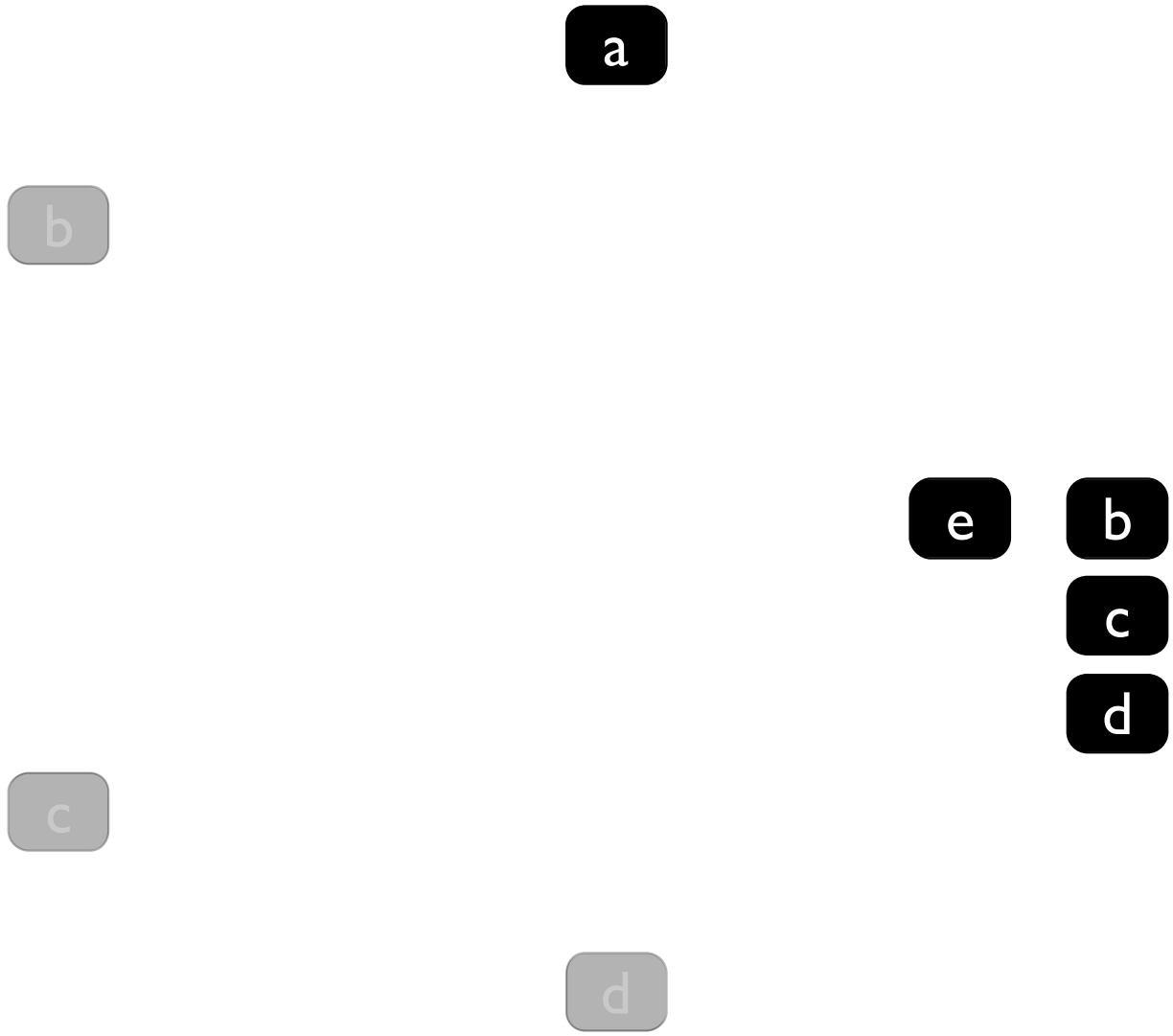
With that ordering, we needed three colors, which is the best possible (because there was a connected subgraph of size 3)

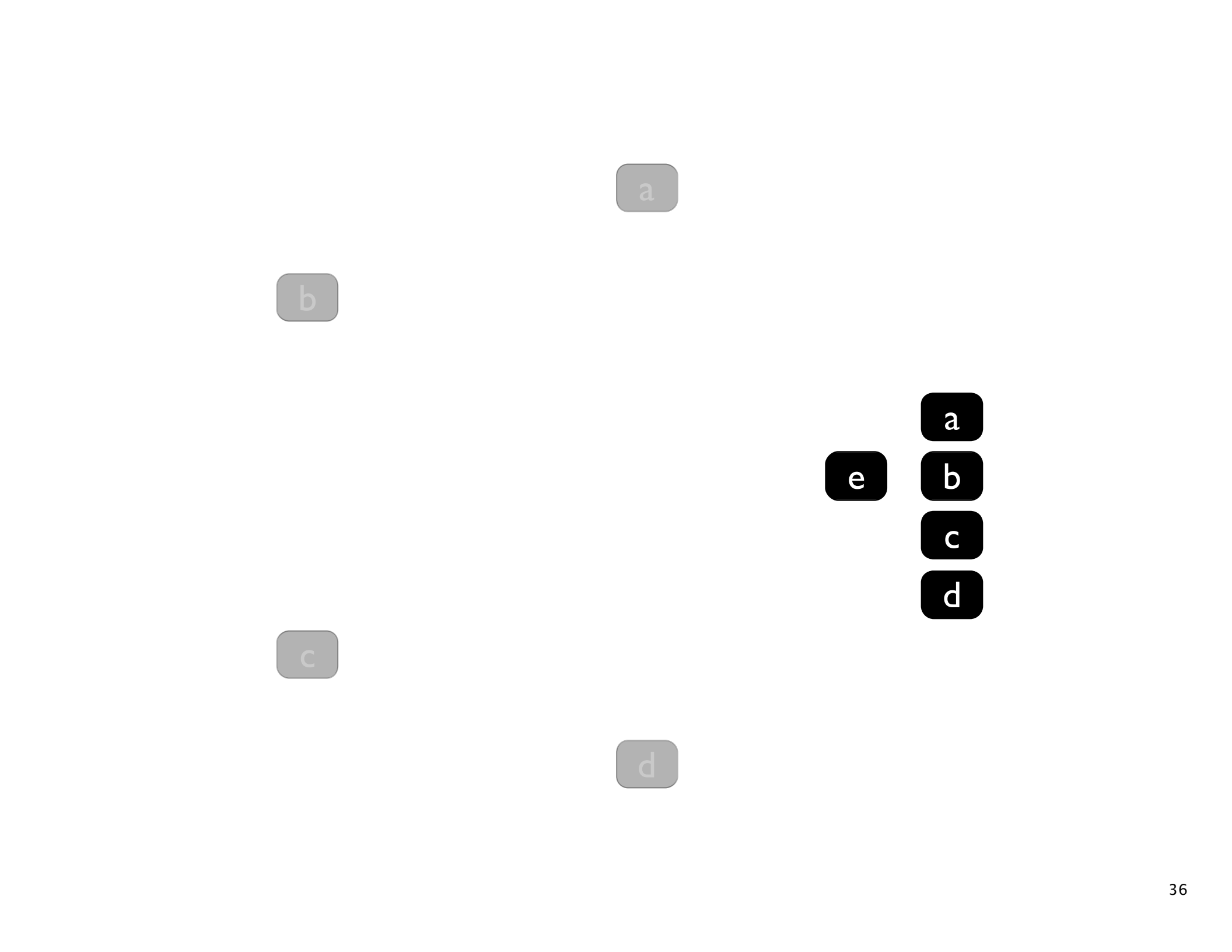
But not all orderings work out so well

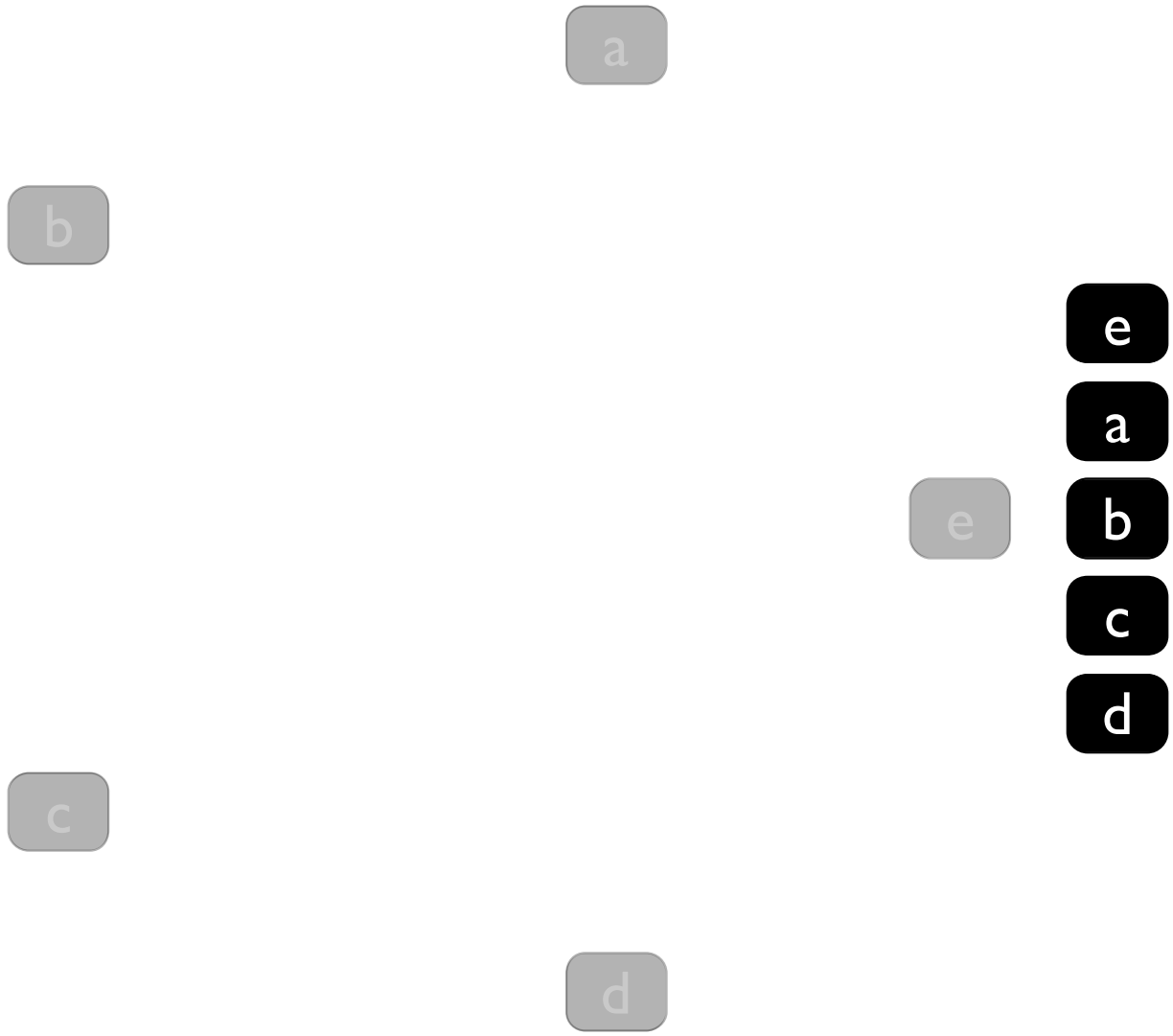

a
b
e
c
d

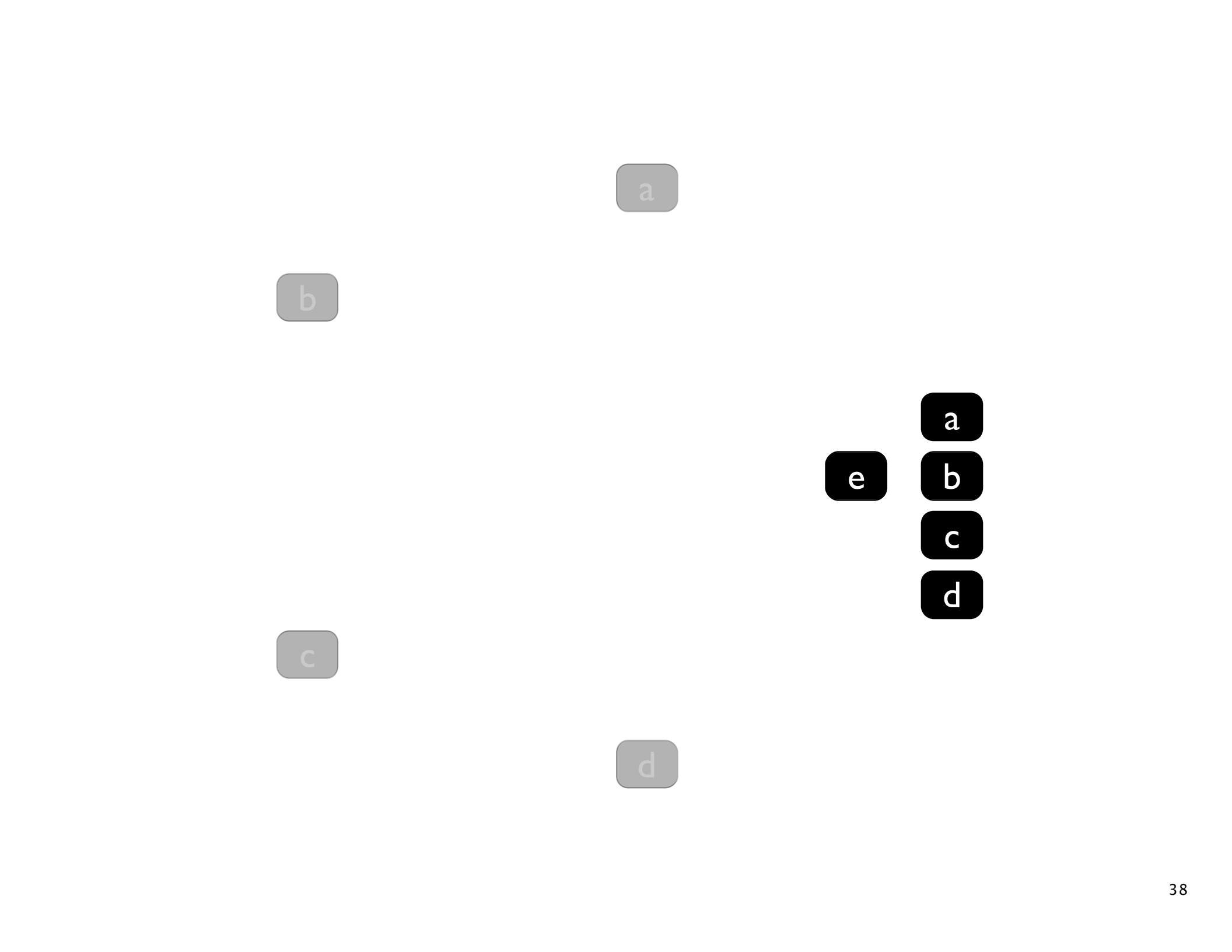

a
b
c
e
d
d

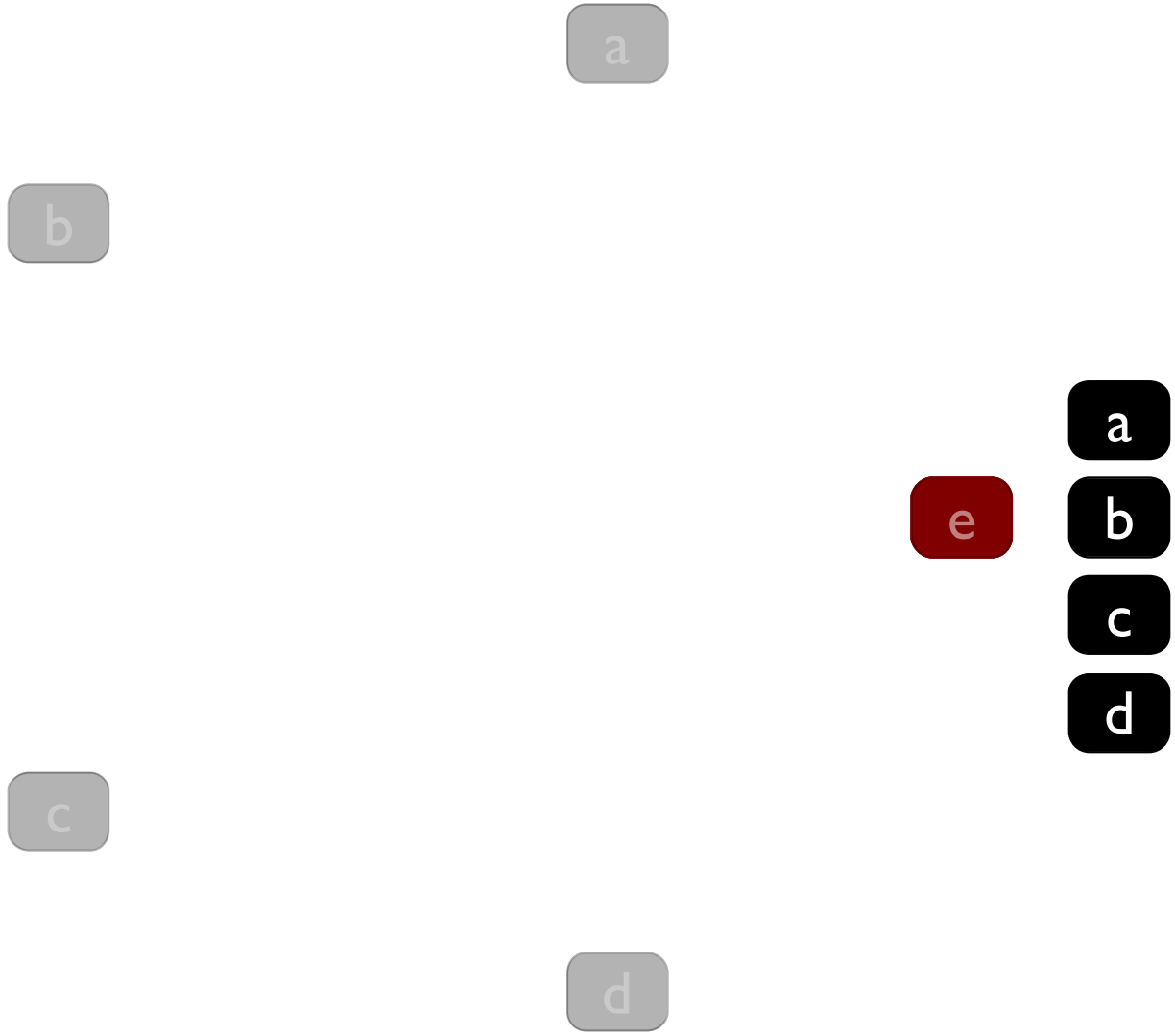

a
b
e
c
d
c
d

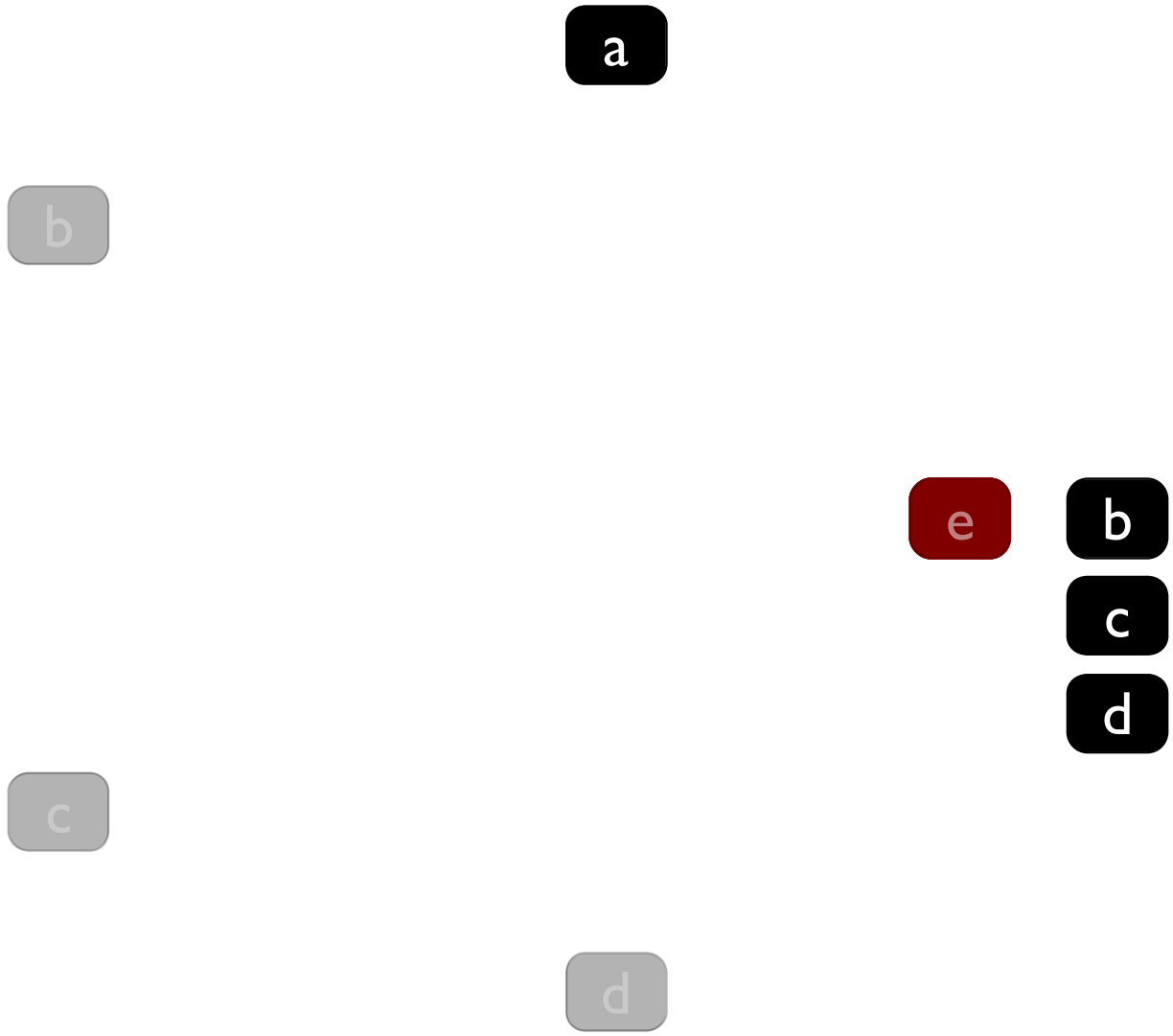

a
b
e
b
c
d
c
d

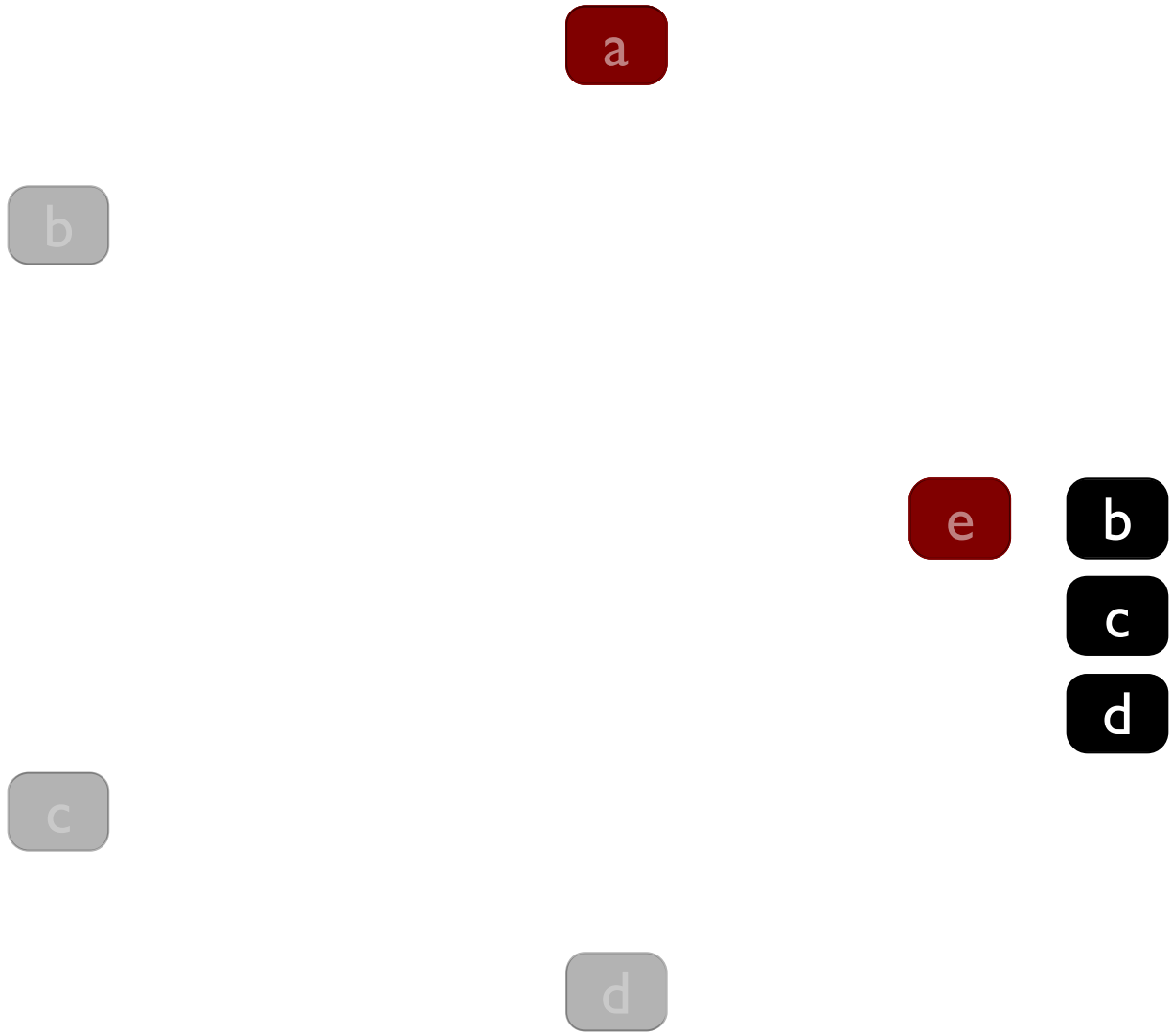

a
b
c
d
e
a
b
c
d

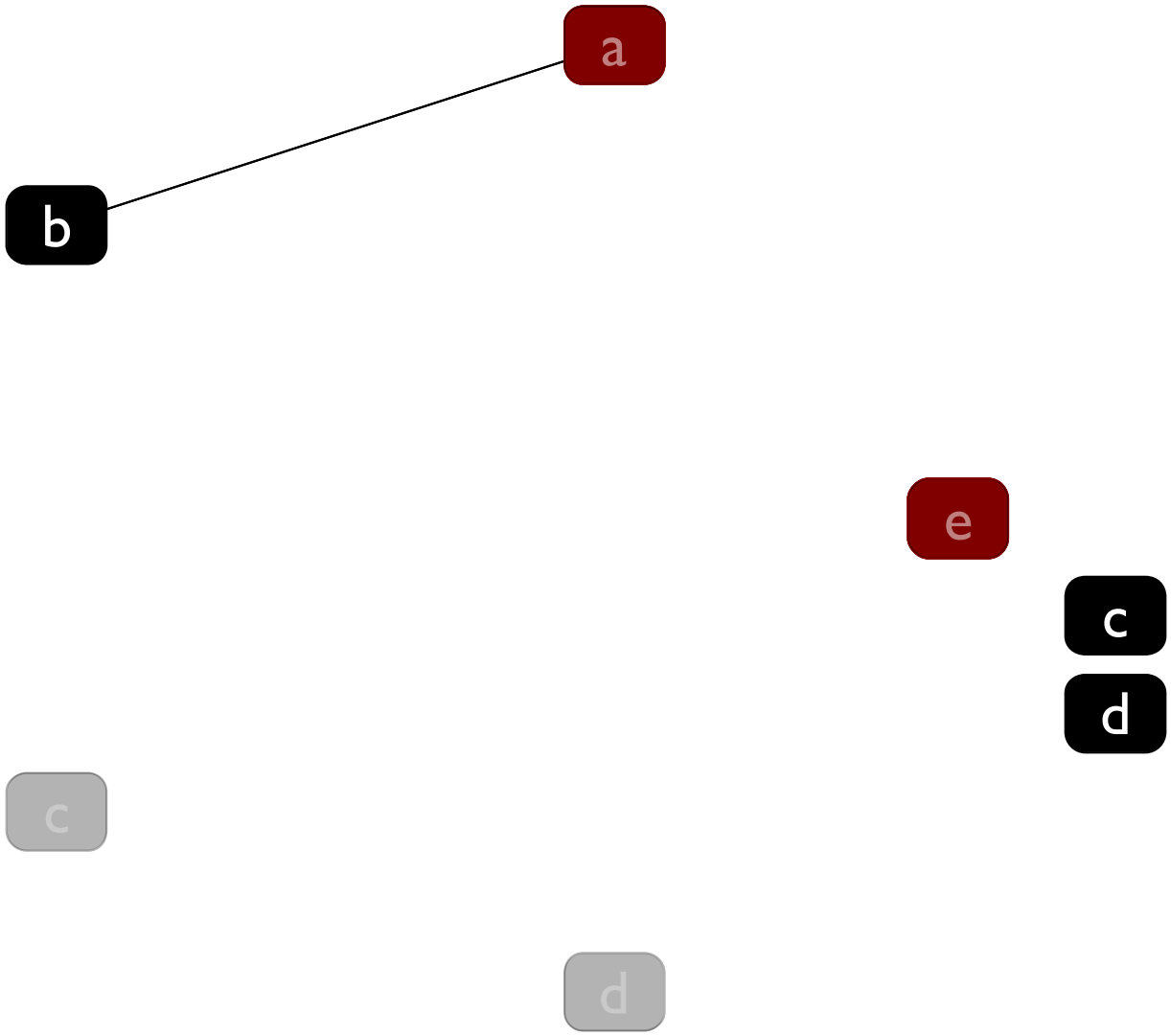

a
b
e
a
e
b
c
d
c
d

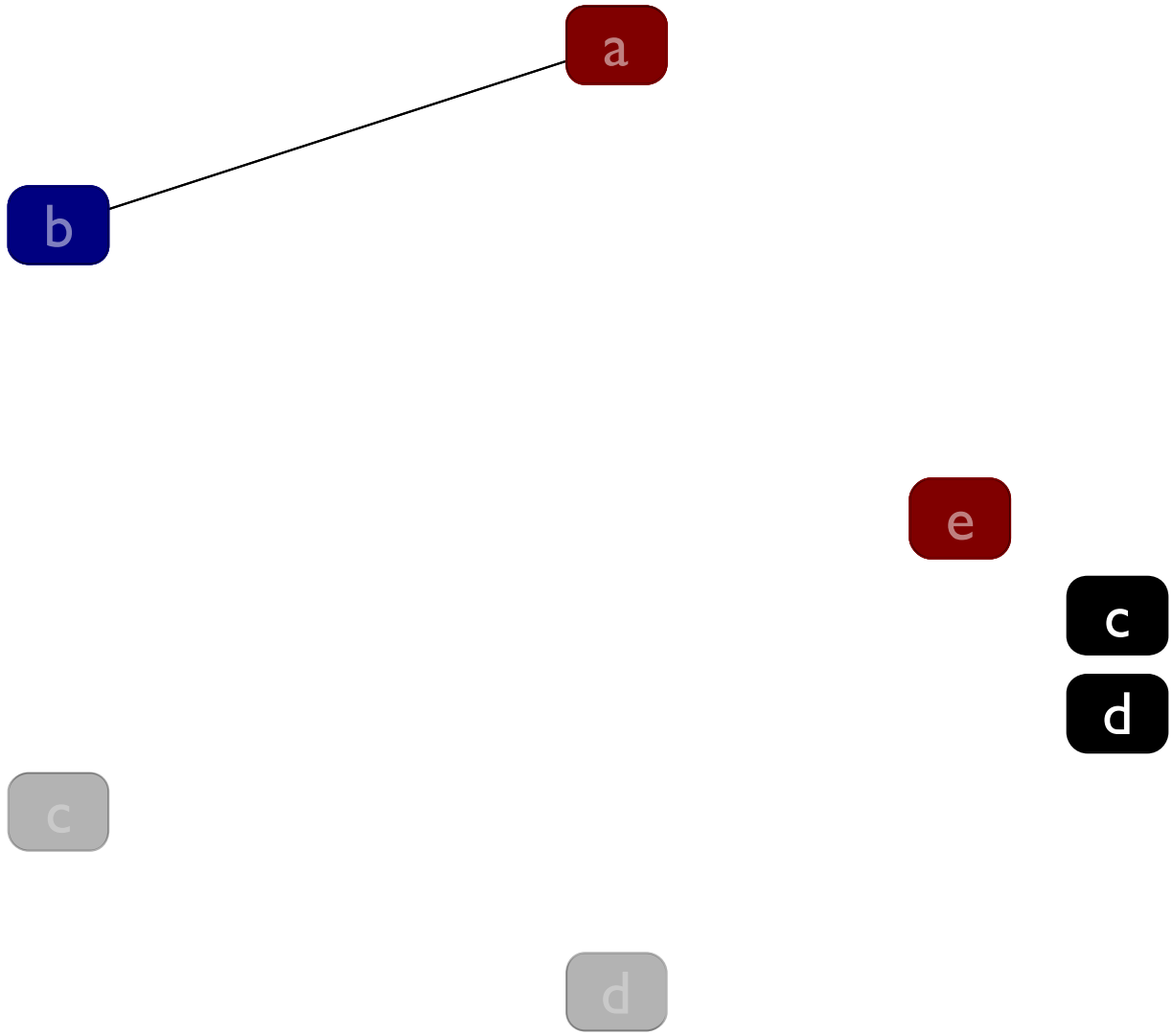

a
b
c
d
e
a
b
c
d

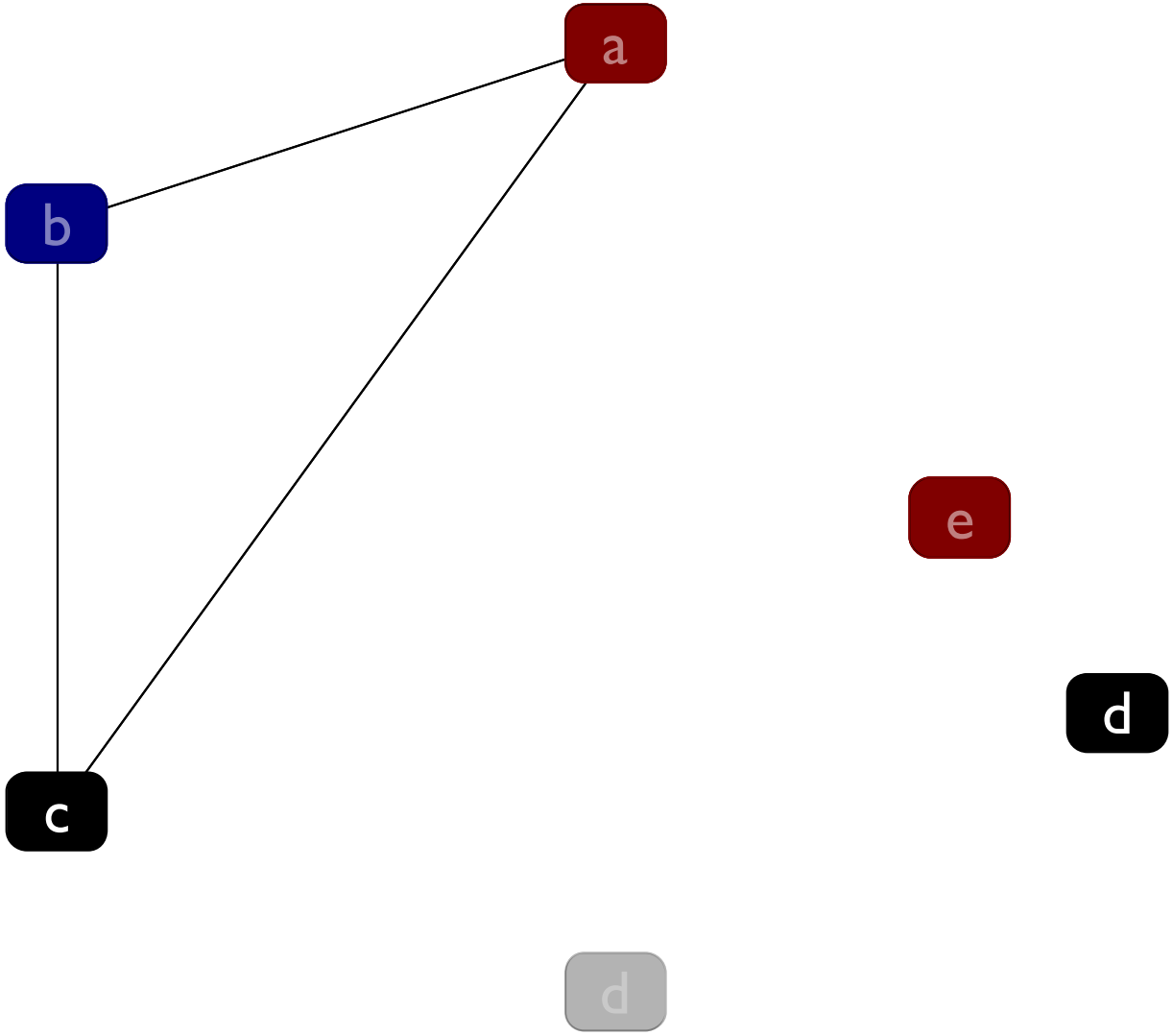

a
b
c
d
e
a
b
c
d

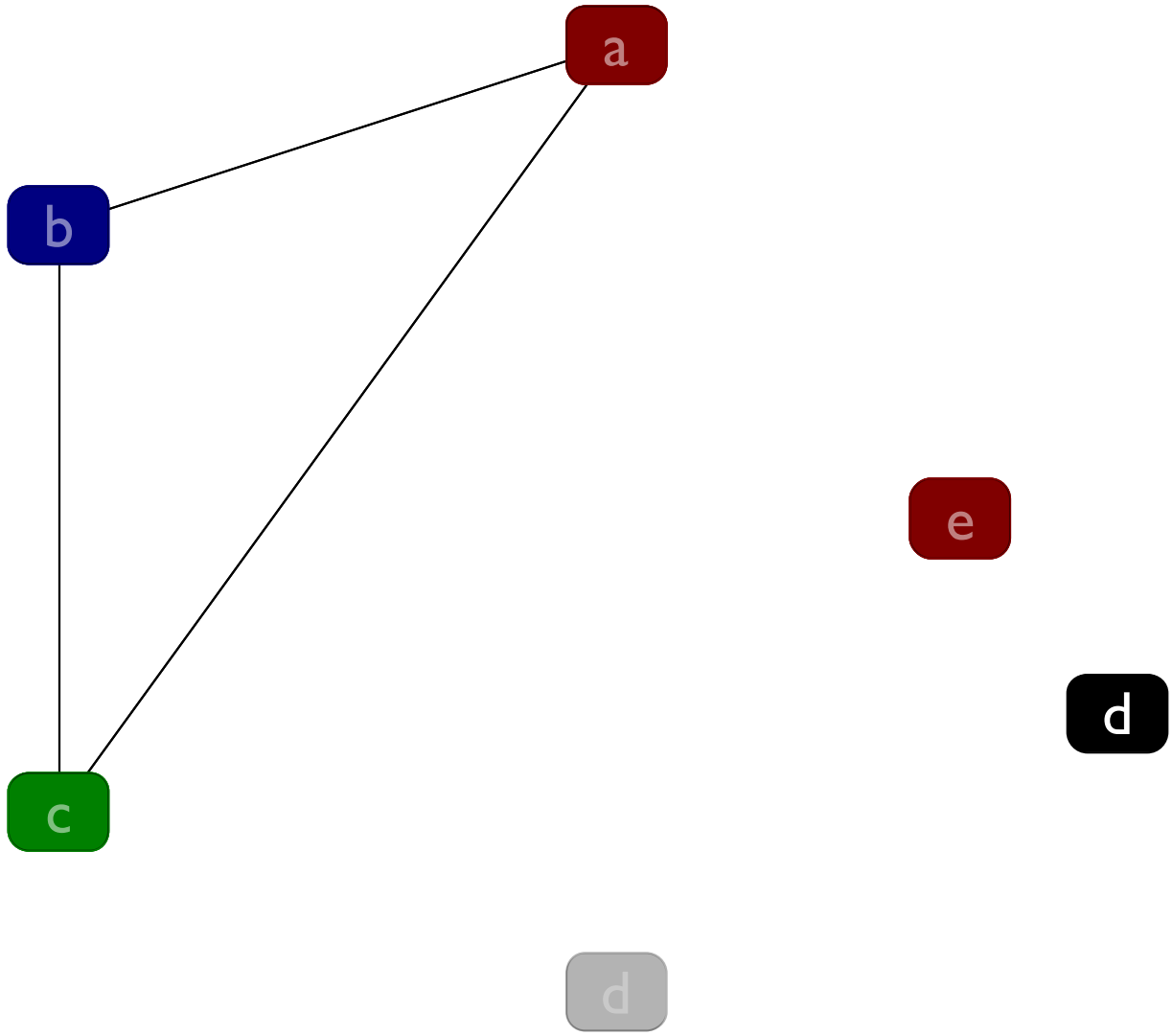

a
b
e
b
c
d
c
d

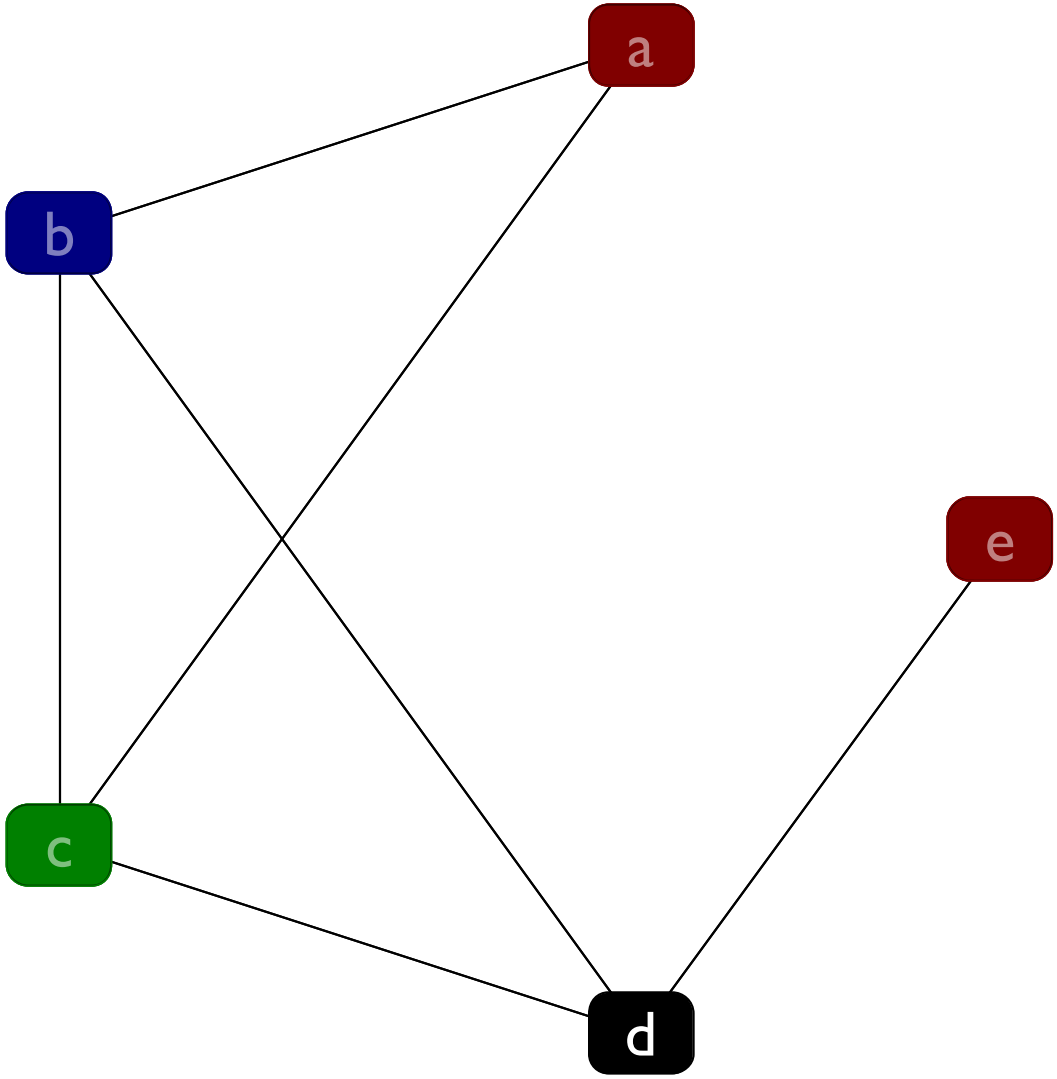

a
b
e
b
c
d
c
d

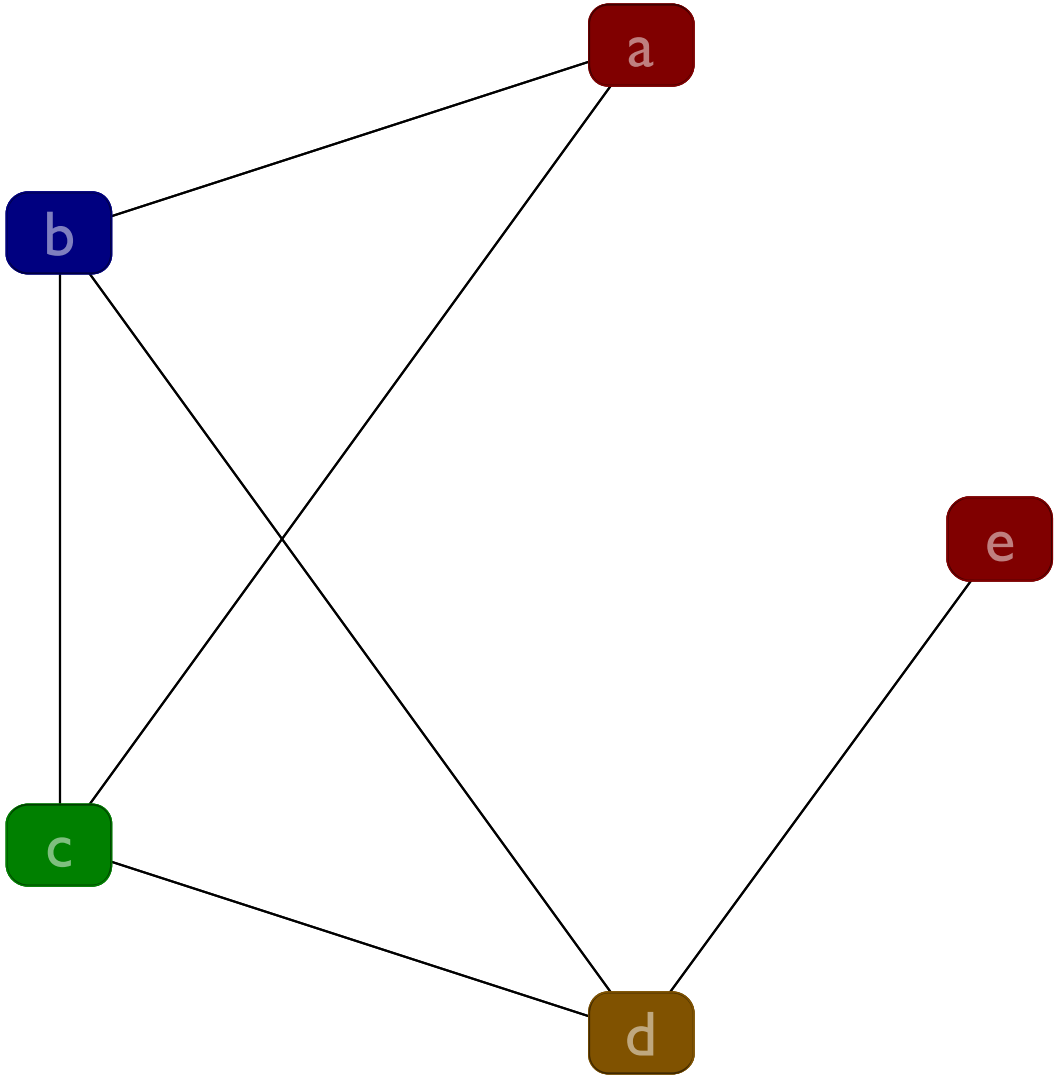

a
b
e
c
d
c
d


a
b
e
c
d
c
d


a
b
c
e
d
d


a
b
c
e
d
d


a
b
e
c
d


a
b
e
c
d

Heuristic: Remove the node with the most edges that's smaller than the number of colors (registers) you want to use