EEСS 483: Compiler Construction

Lecture 17:

Register Allocation Part 2: Graph Coloring and Code Generation

March 19
Winter Semester 2025

Announcements

- Assignment 4 to be released tonight. Due April 4th
- Exam grades to be released next week.

Register Allocation

3 Steps

1. **Liveness analysis**: identify when each variable's value is needed in the program
2. **Conflict analysis**: identify which variables interfere with each other
3. **Graph Coloring**: assign variables to registers so that interfering registers are assigned different registers.
    1. Spilling: if necessary, assign some variables to stack slots

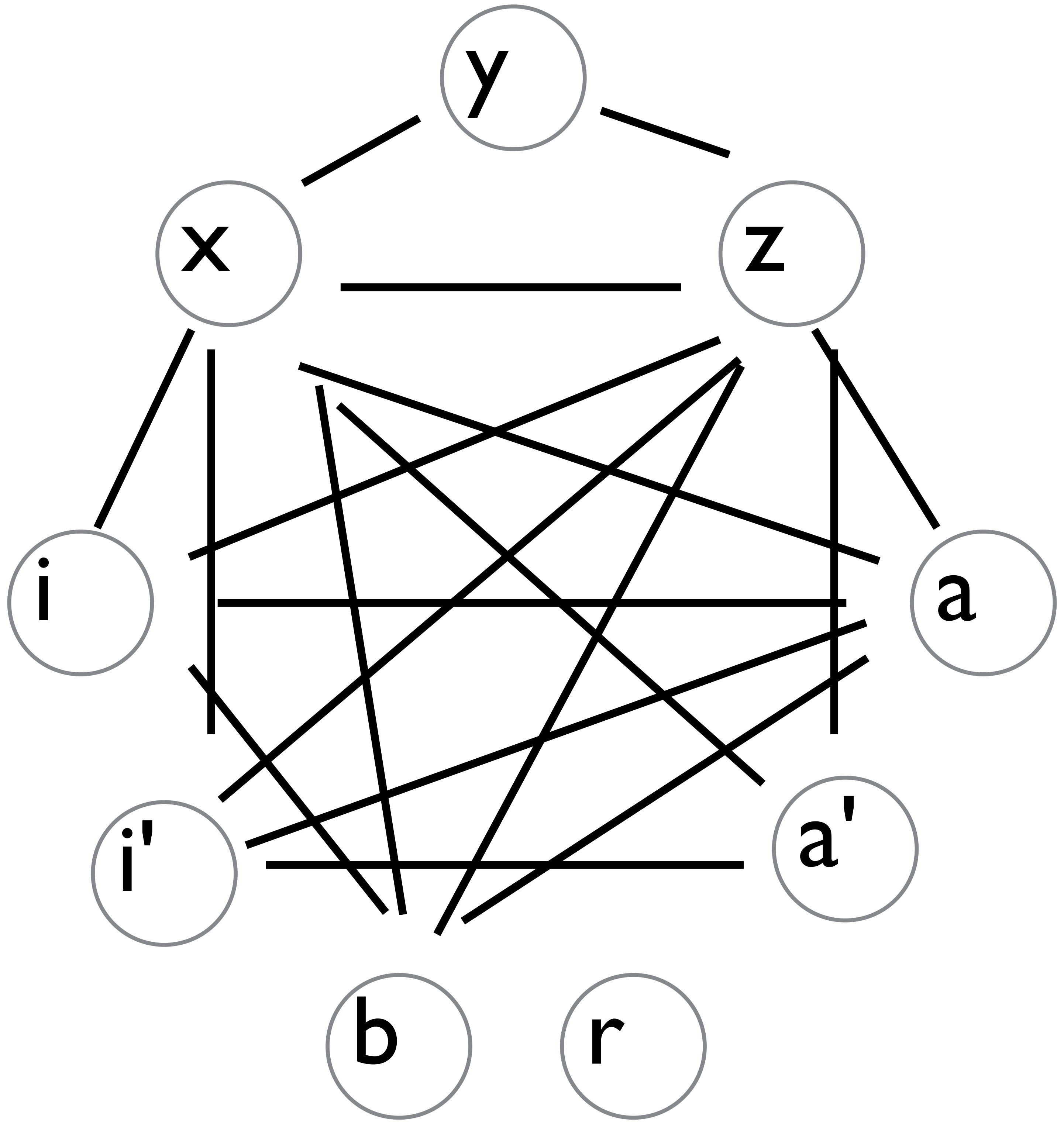
SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Liveness Info

1:	{x,z}
2:	{x,y,z}
3:	{a,i,x,z}
4:	{a,b,i,x,z}
5/6:	{a,z}
7:	{r}
8/9:	{a,i,x,z}
10:	{a,i',x,z}
11:	{a',i',x,z}

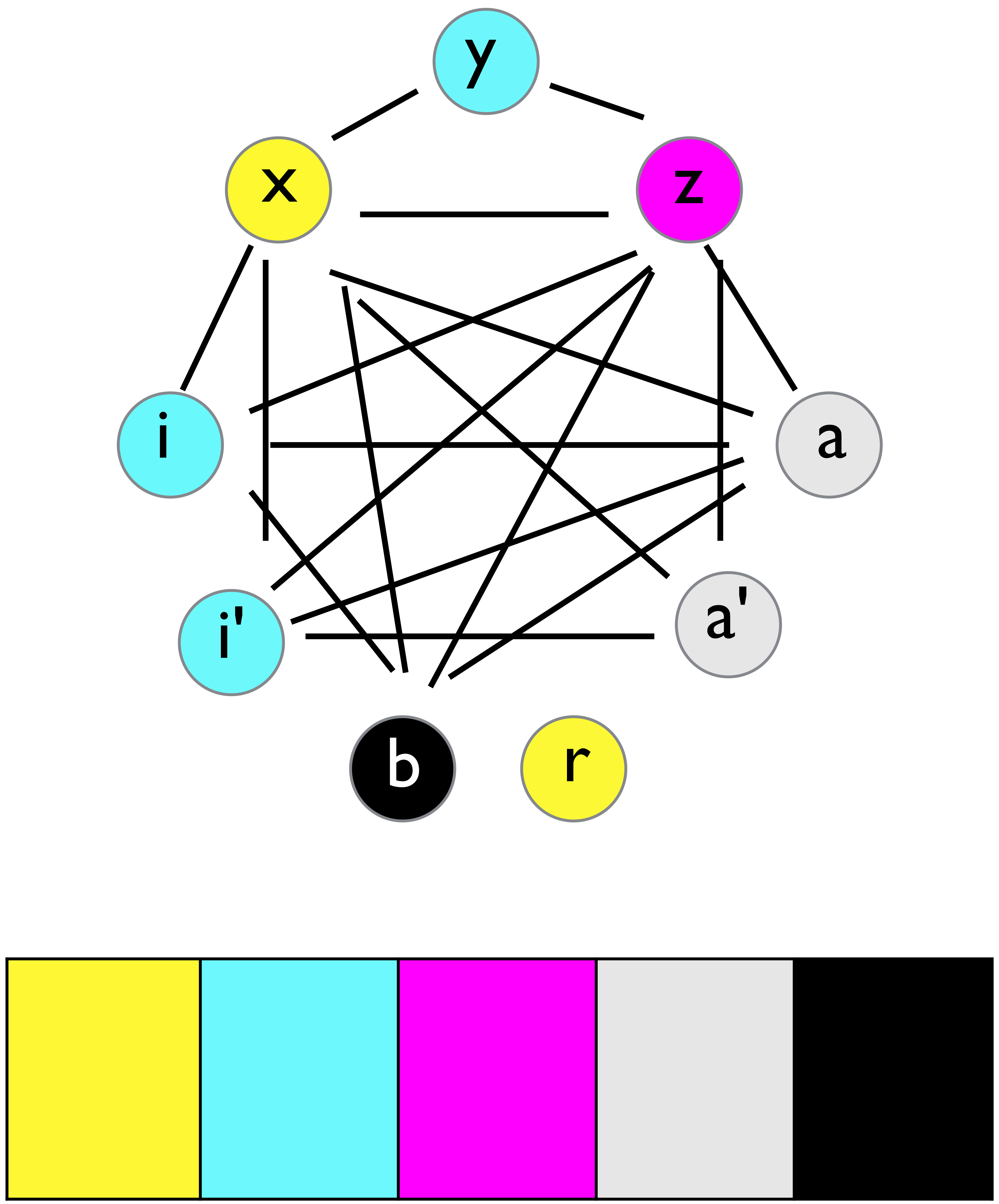
Interference Graph



SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

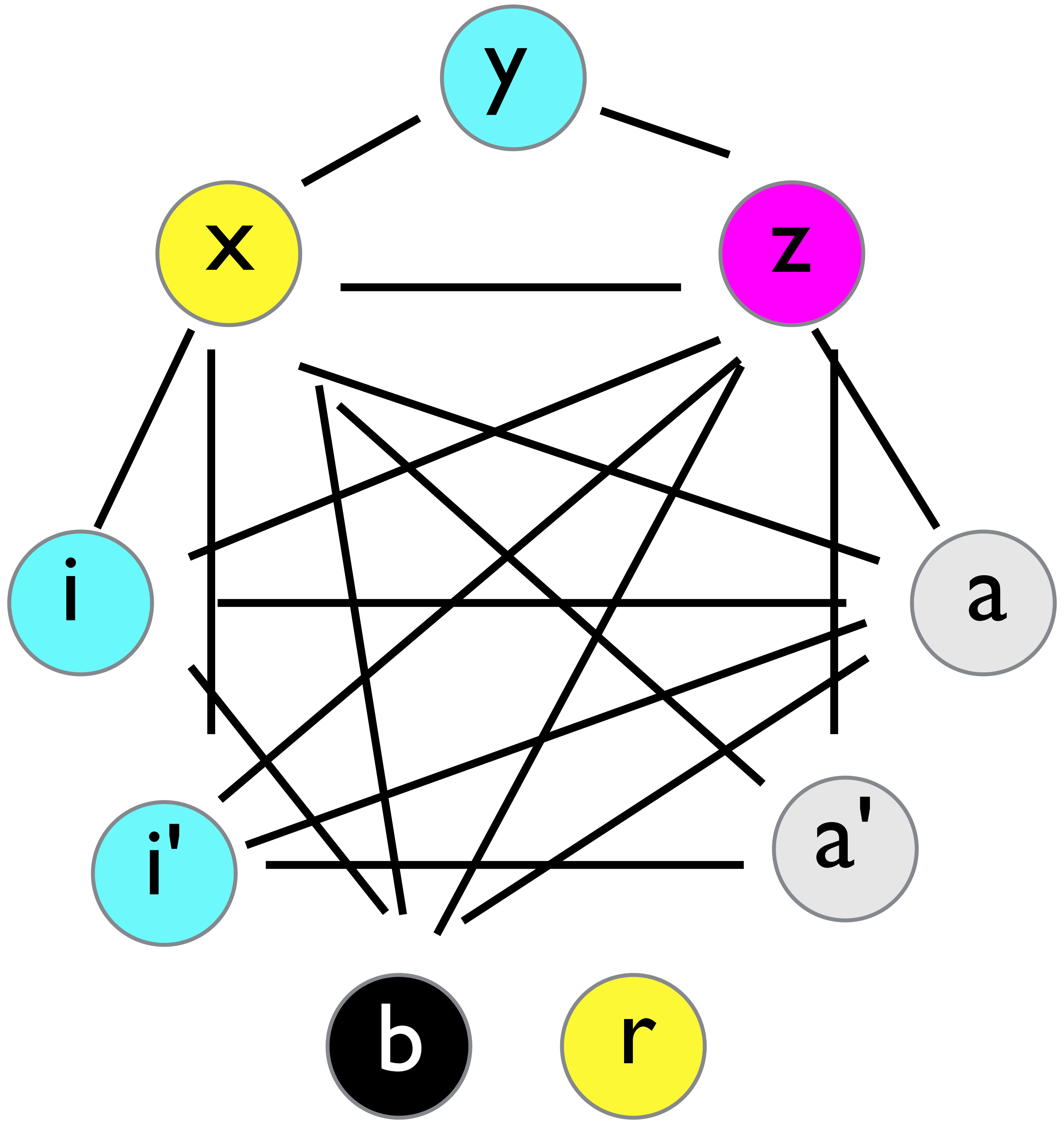
Interference Graph



SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Interference Graph



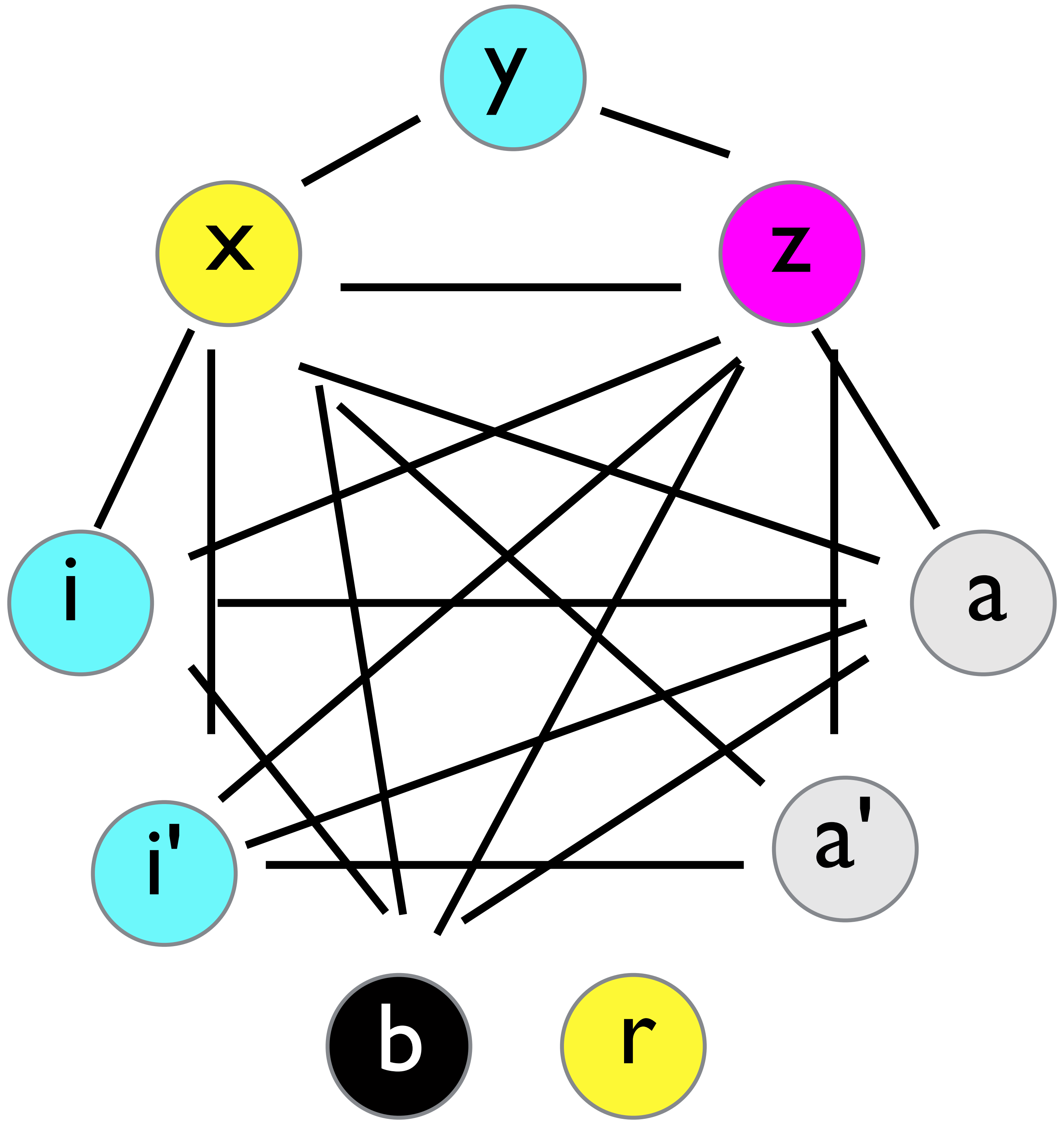
rax	rsi	rdi	rcx	rdx

SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

```
f:
  mov rcx, 0
  jmp loop
loop:
  cmp rsi, 0
  mov rdx, 0
  sete rdx
  cmp rdx 0
  jne thn
  jmp els
thn:
  mov rax, rcx
  imul rax, rdi
  ret
els:
  sub rsi, 1
  add rcx, rax
  jmp loop
```

Interference Graph



rax	rsi	rdi	rcx	rdx

Graph Coloring Register Allocation

Given our register interference graph, want to assign a register to each variable so that no interfering variables are assigned the same register.

Equivalent to graph coloring of the interference graph

- think of each register as a "color" and we want to paint each node so that no adjacent nodes are the same color.

Efficient algorithm for graph coloring -> efficient algorithm for graph coloring!

Graph Coloring is Hard

Determining a whether a graph is k-colorable is NP-complete for k > 2.

- So no polytime algorithm is known

Does that mean register allocation is NP-hard?

Is Register Allocation Hard?

Chaitin et al, "Register allocation via coloring", *Computer Languages* 1981

- Showed that the register allocation problem for a language with assignments and arbitrary control flow (goto) is **equivalent** to graph coloring
- every graph arises as the interference graph of some program

So register allocation of an imperative language with goto is NP complete.

- But our programs are more restrictive: SSA form...

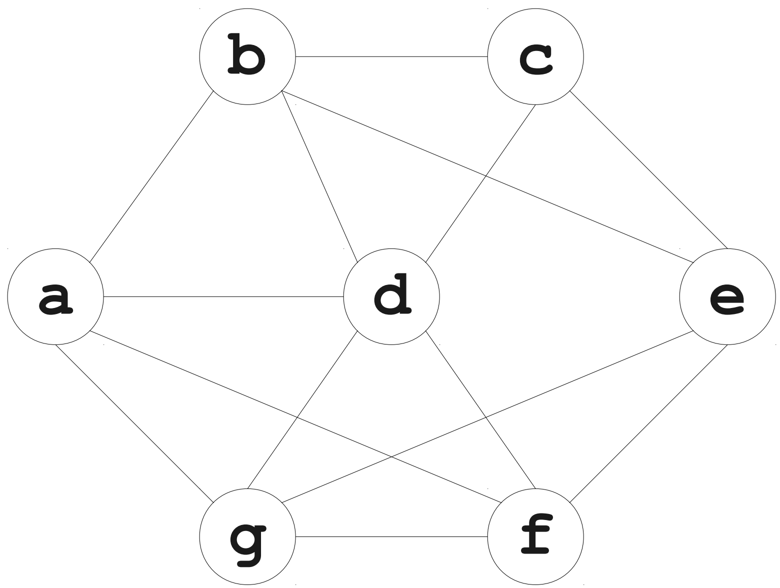
Chaitin's Algorithm

Chaitin's Algorithm

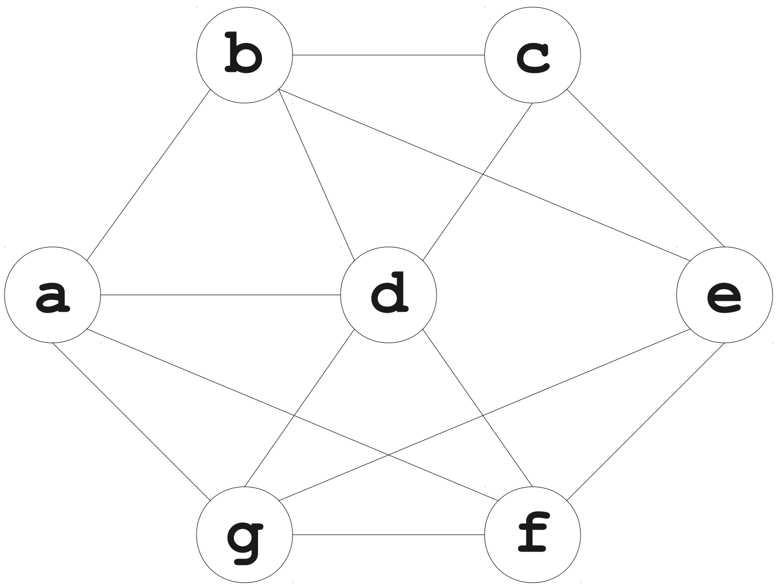
- Intuition:
  - Suppose we are trying to k-color a graph and find a node with fewer than k edges.
  - If we delete this node from the graph and color what remains, we can find a color for this node if we add it back in.
  - Reason: With fewer than k neighbors, some color must be left over.
- Algorithm:
  - Find a node with fewer than k outgoing edges.
  - Remove it from the graph.
  - Recursively color the rest of the graph.
  - Add the node back in.
  - Assign it a valid color.

Chaitin's Algorithm

Chaitin's Algorithm



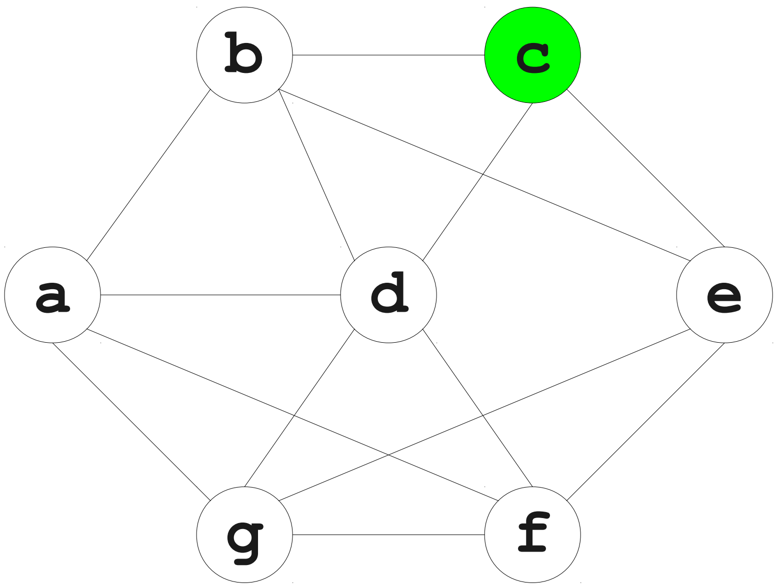
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

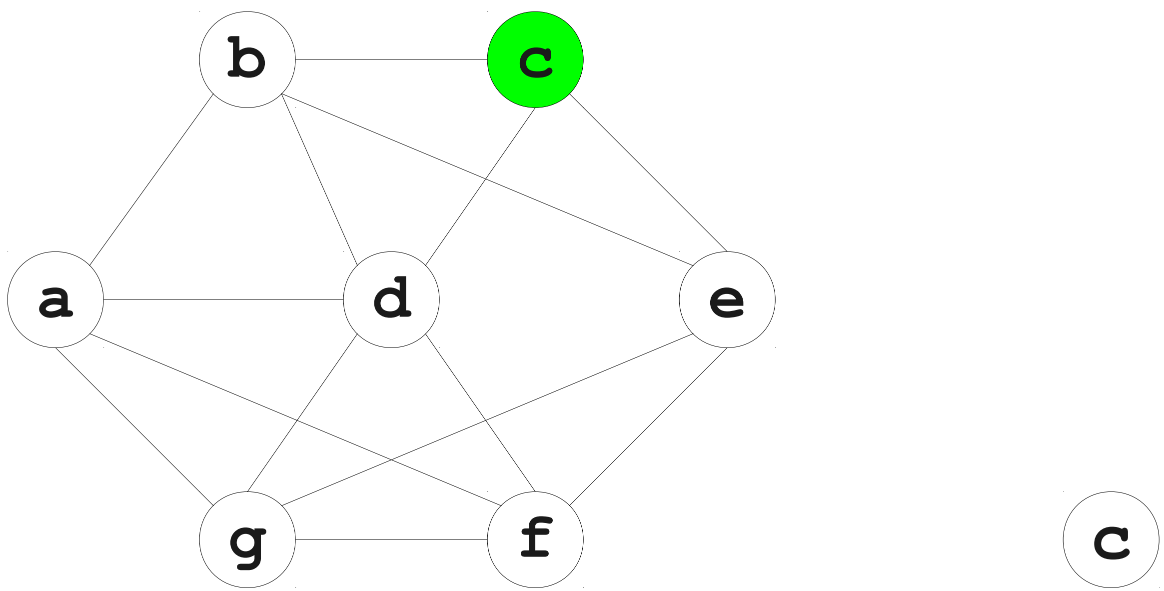
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

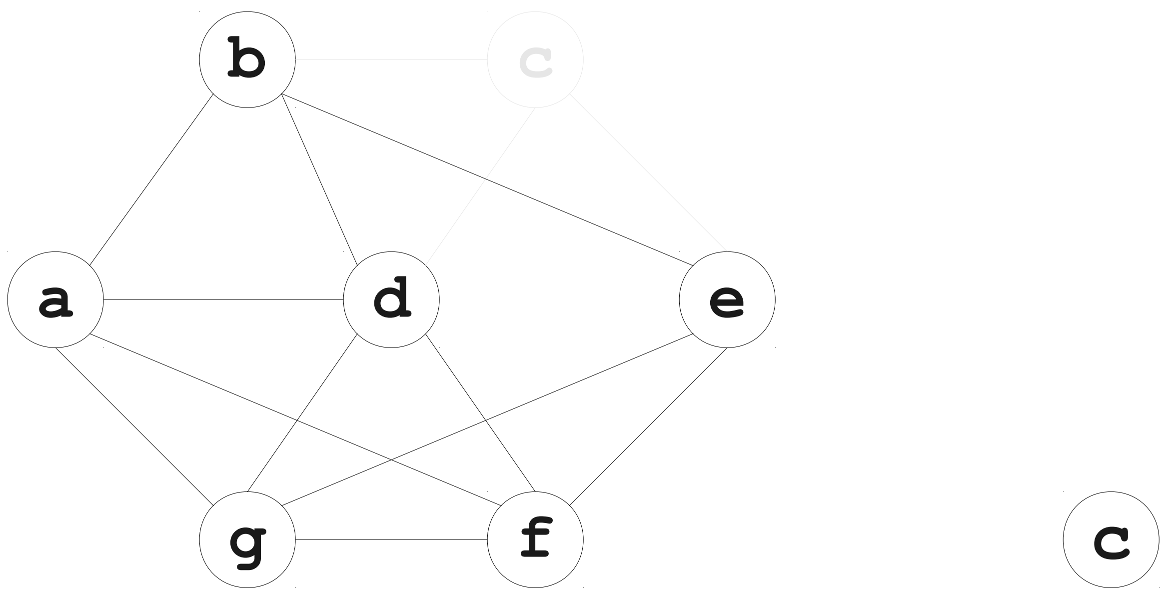
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

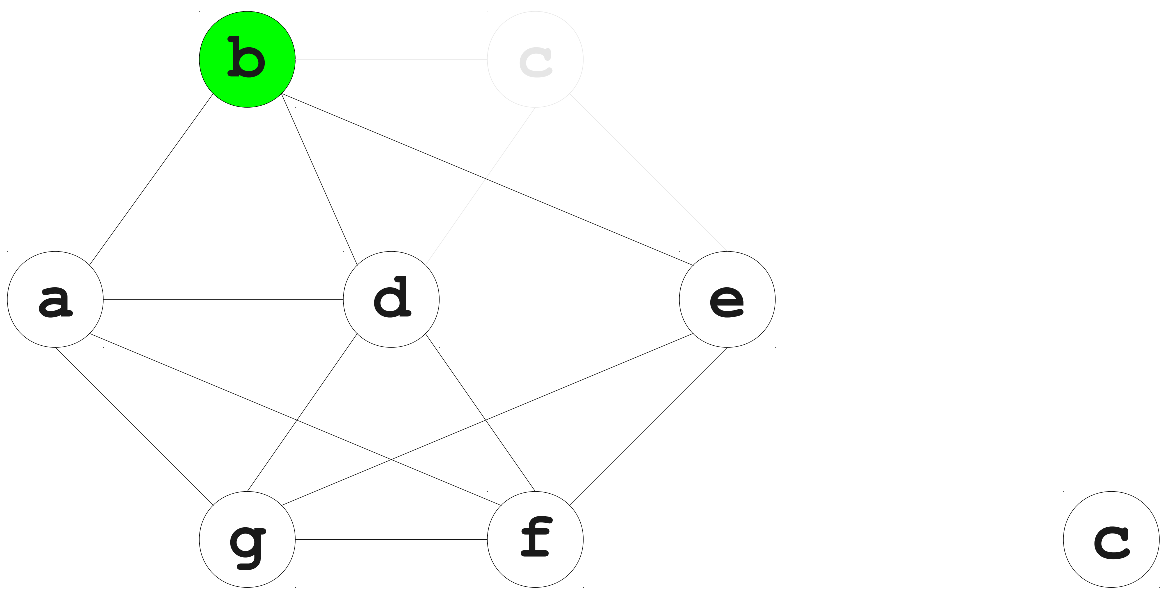
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

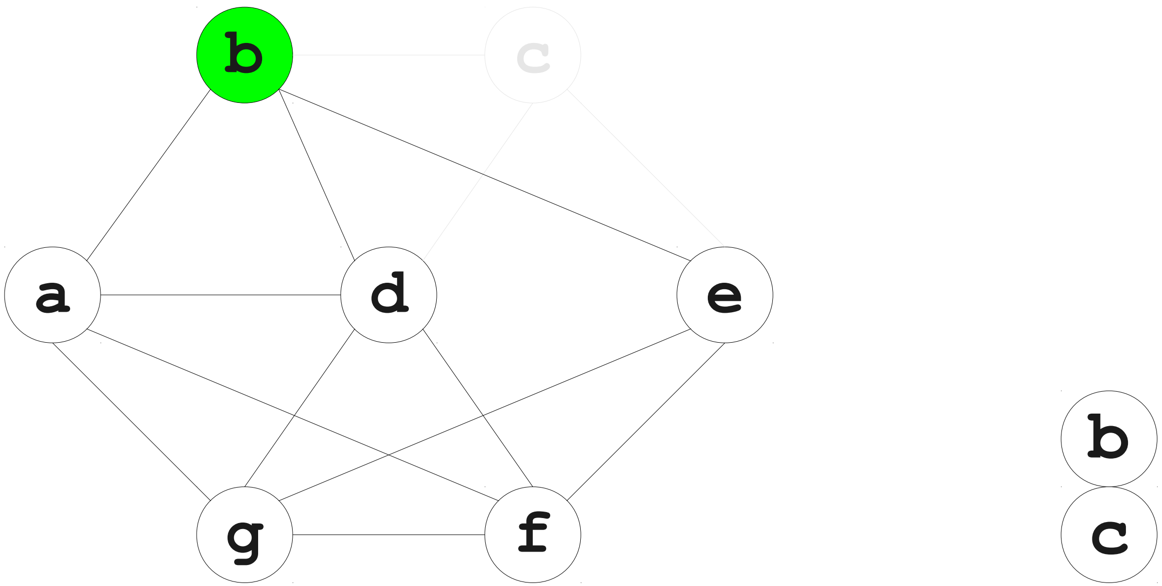
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

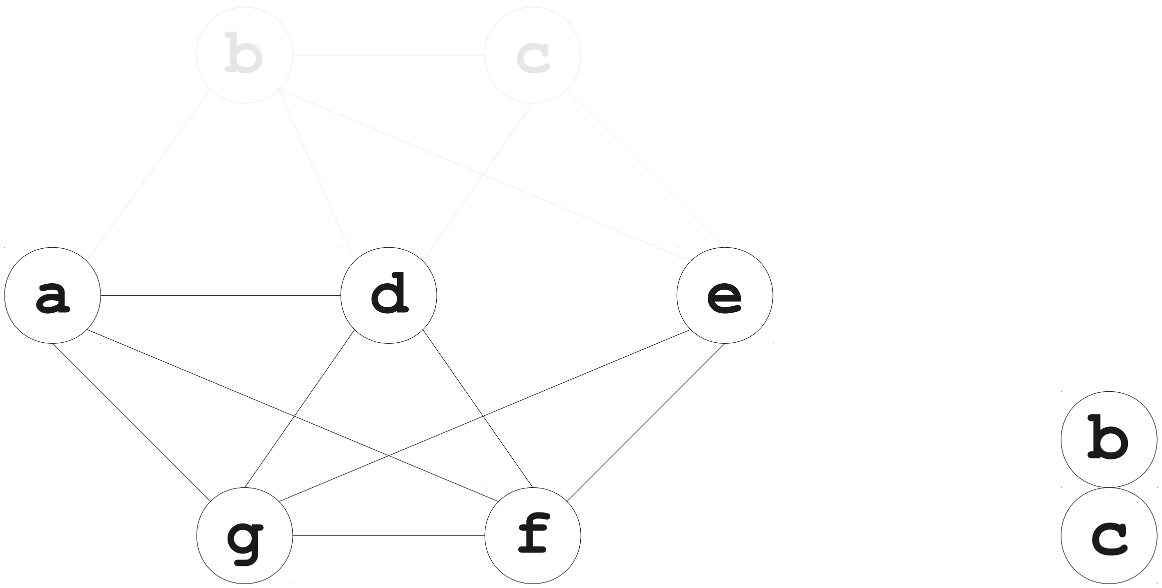
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

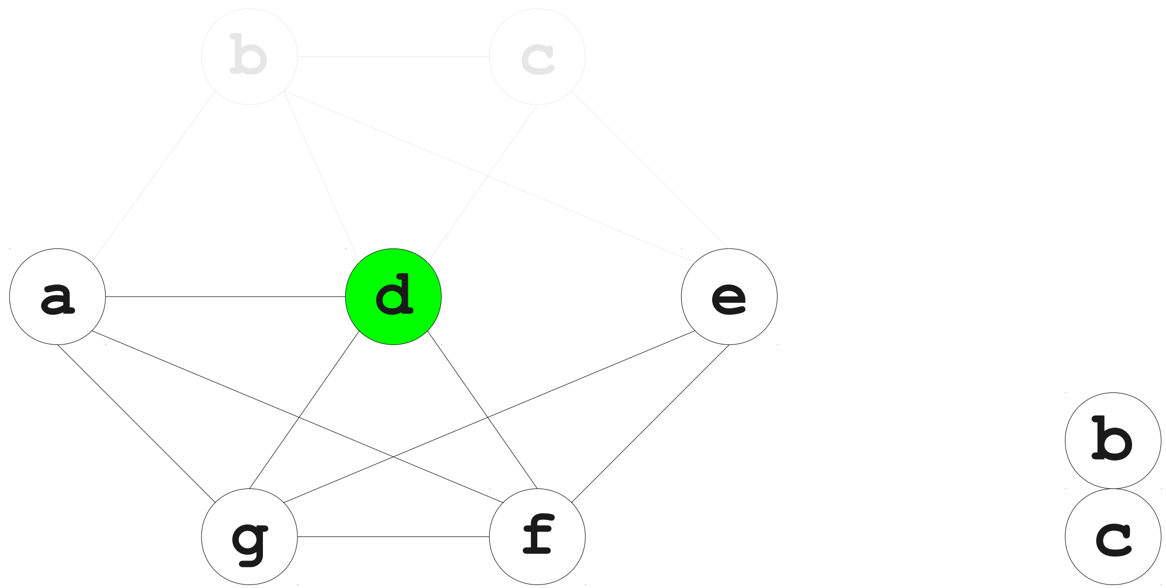
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

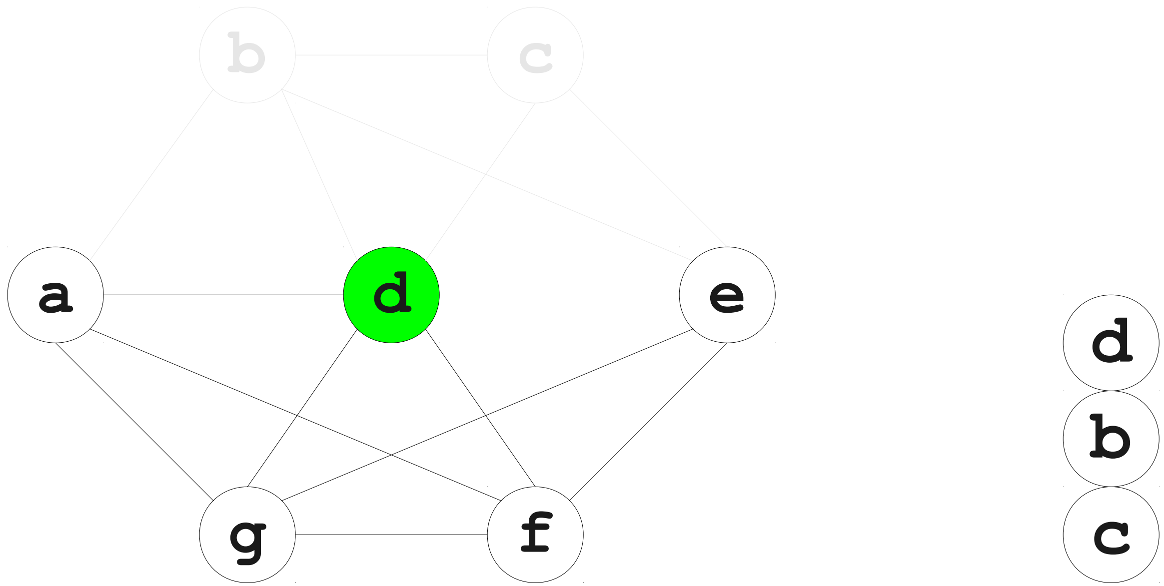
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

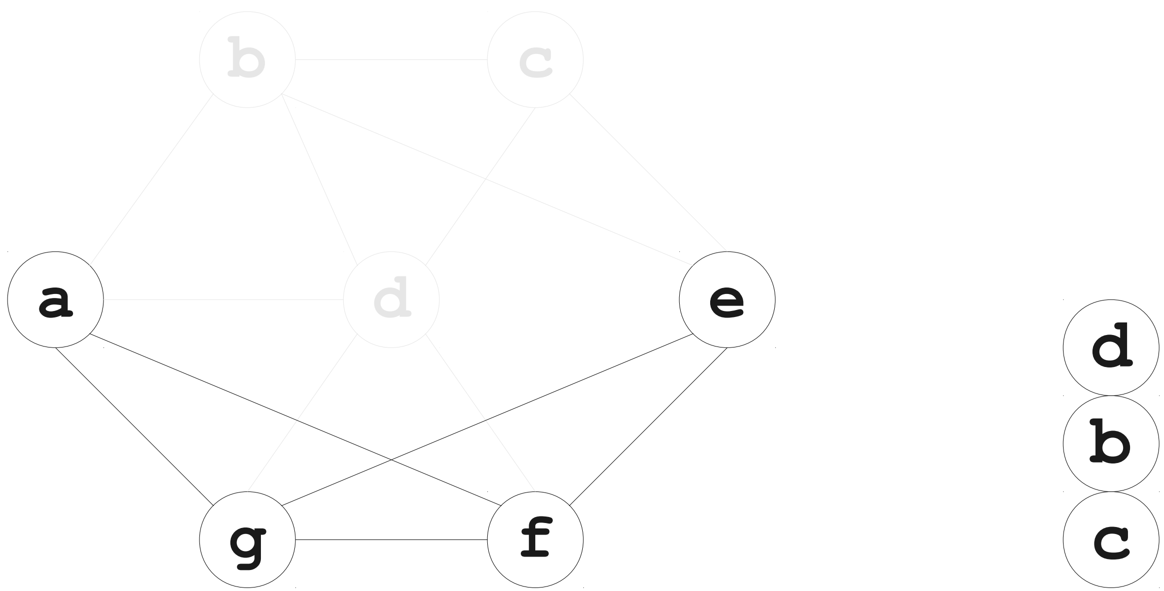
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

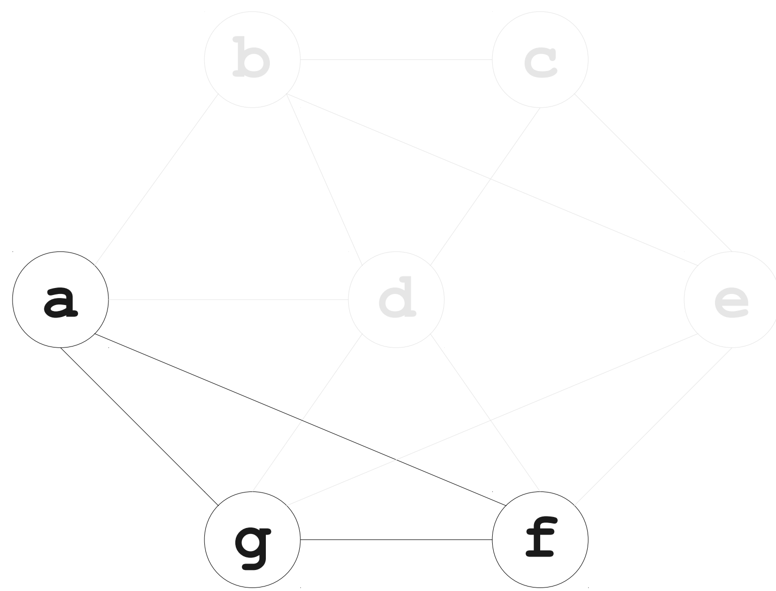
Chaitin's Algorithm

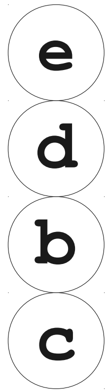


Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm

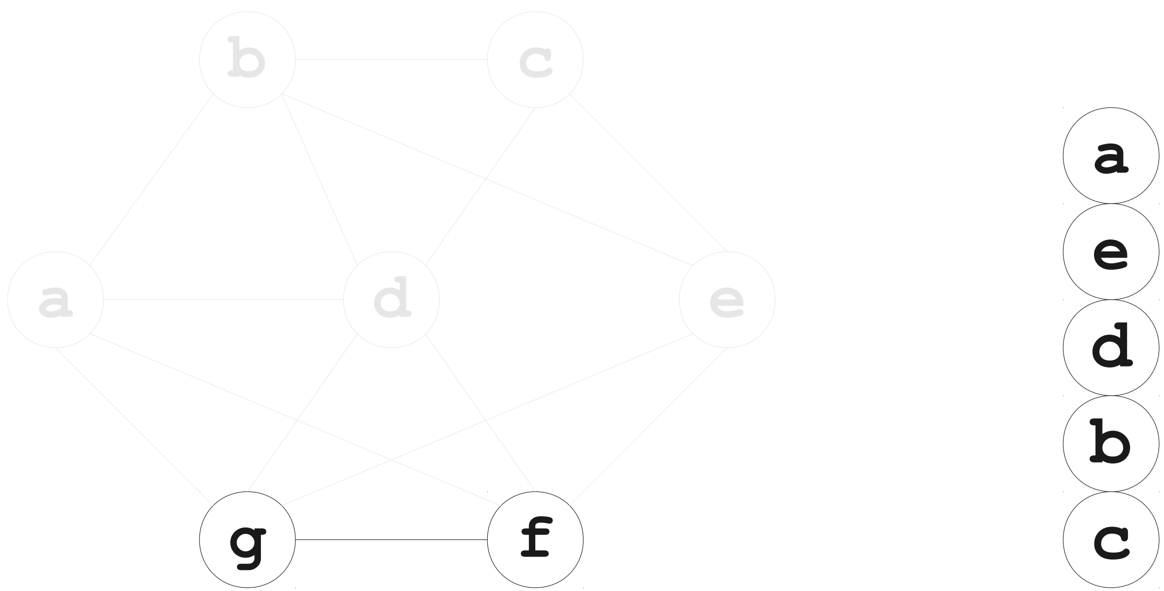




Registers

R_0	R_1	R_2	R_3

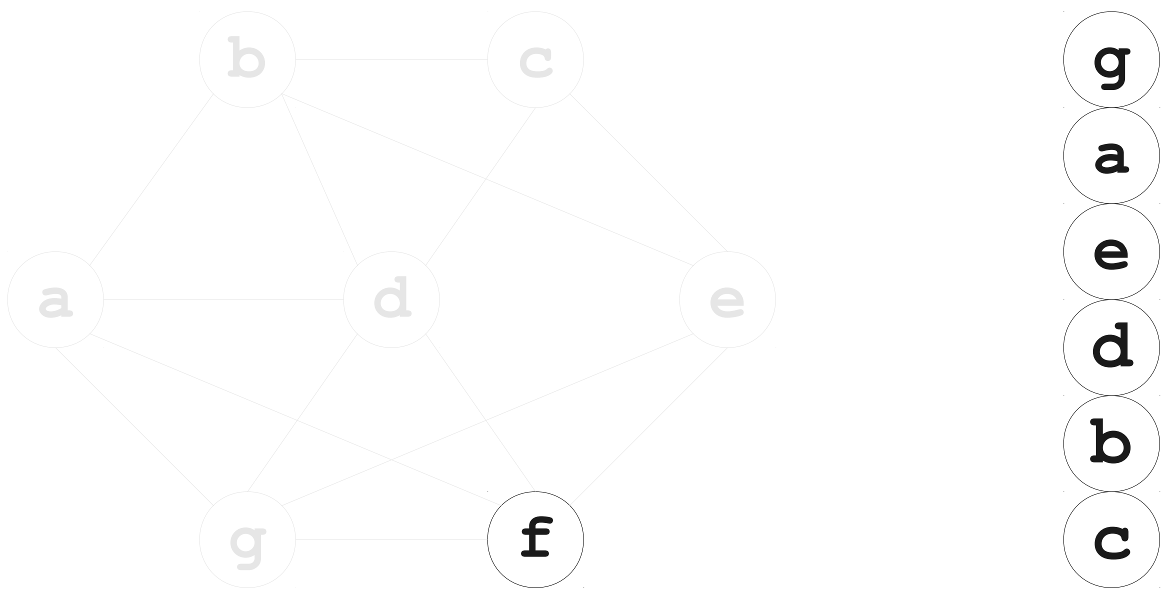
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



Registers

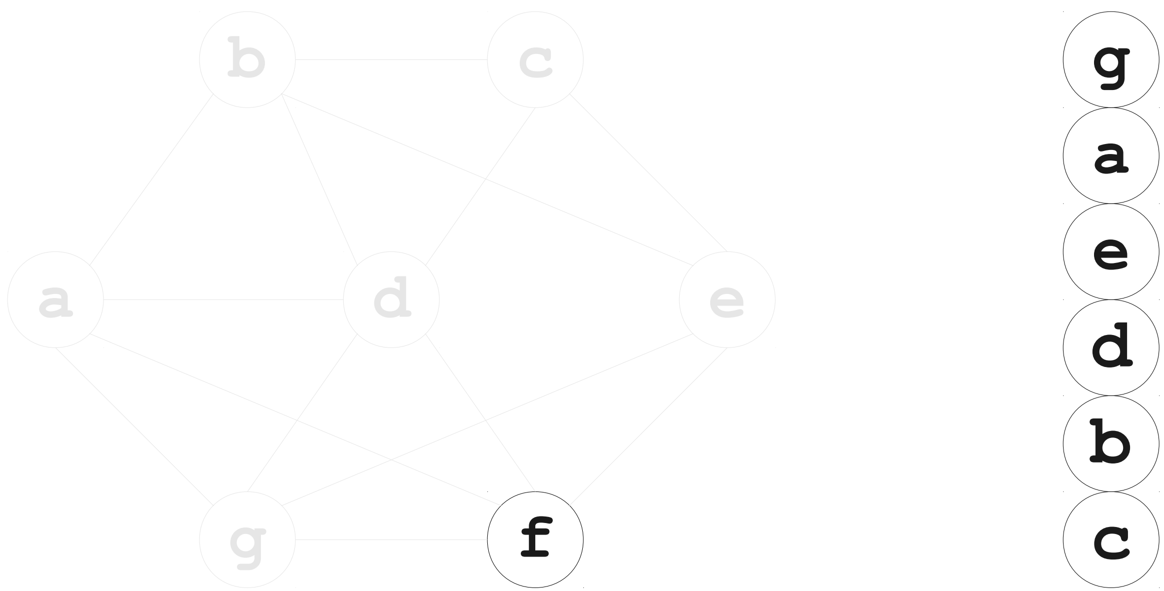
R_0	R_1	R_2	R_3

Chaitin's Algorithm

Registers

R_0	R_1	R_2	R_3

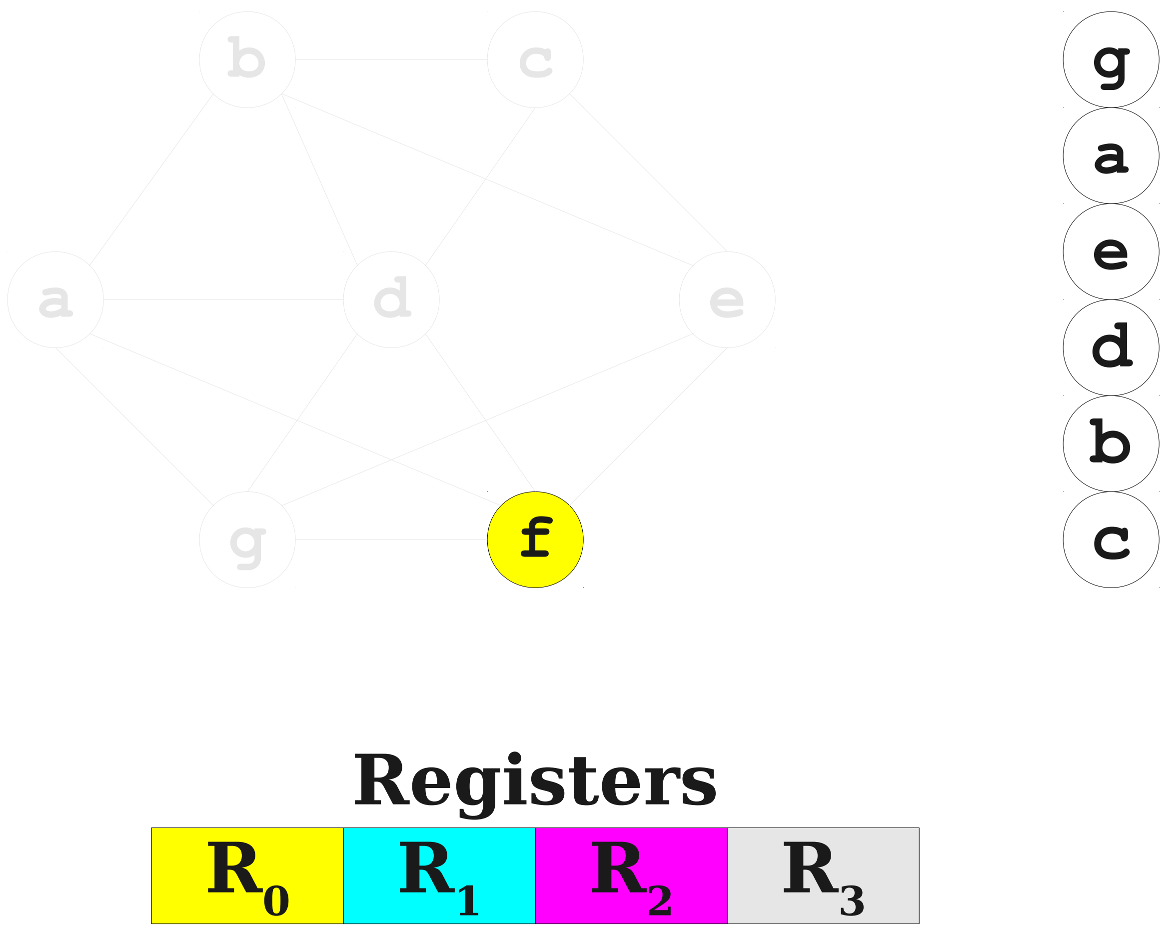
Chaitin's Algorithm



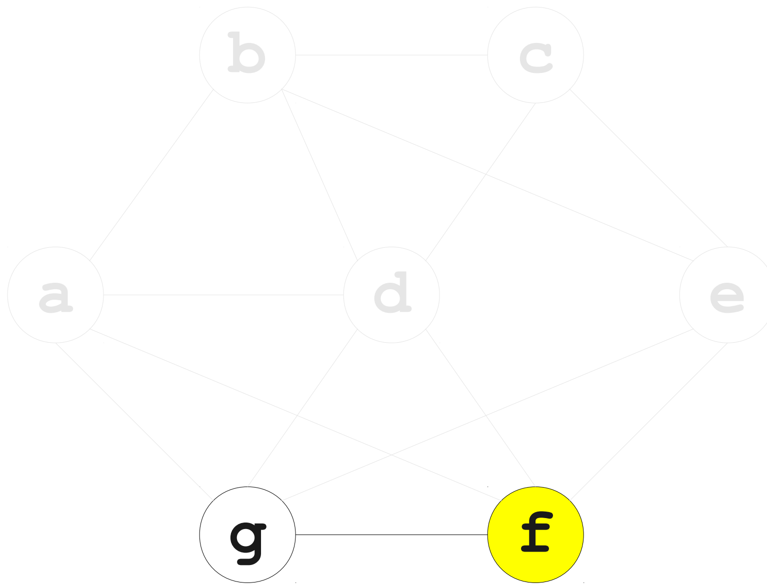
Registers

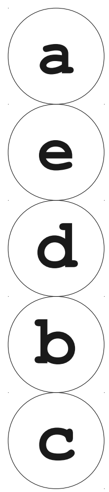
R_0	R_1	R_2	R_3

Chaitin's Algorithm



Chaitin's Algorithm

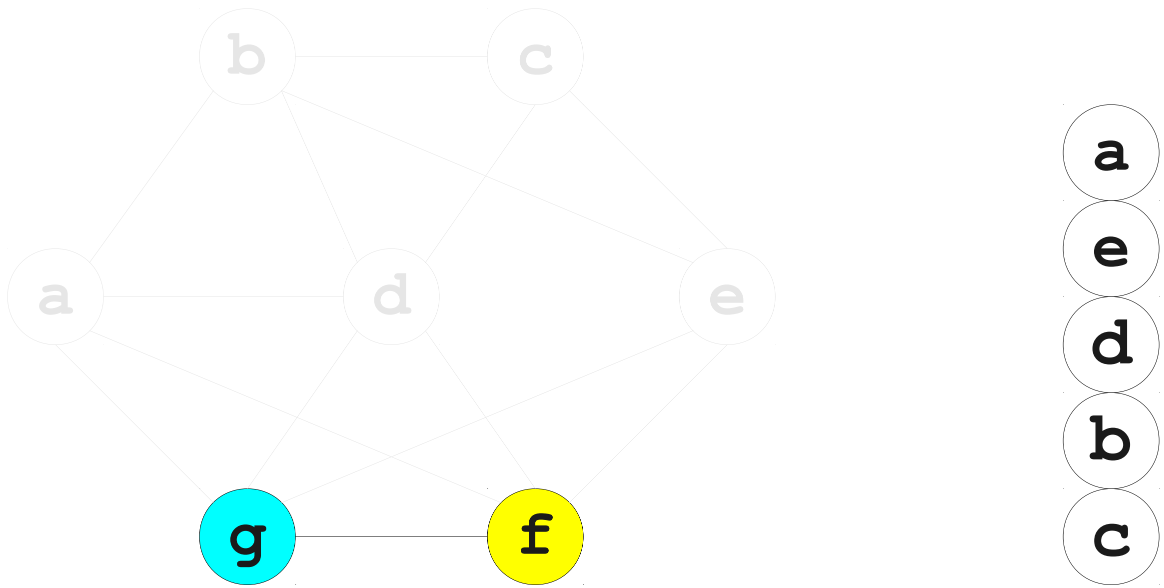




Registers

R_0	R_1	R_2	R_3

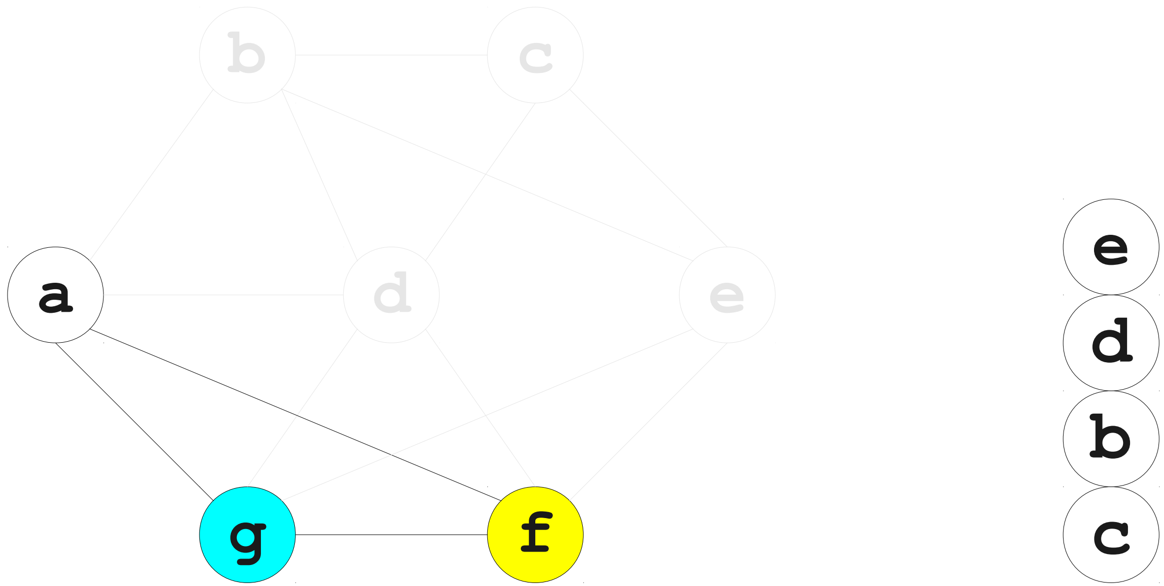
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

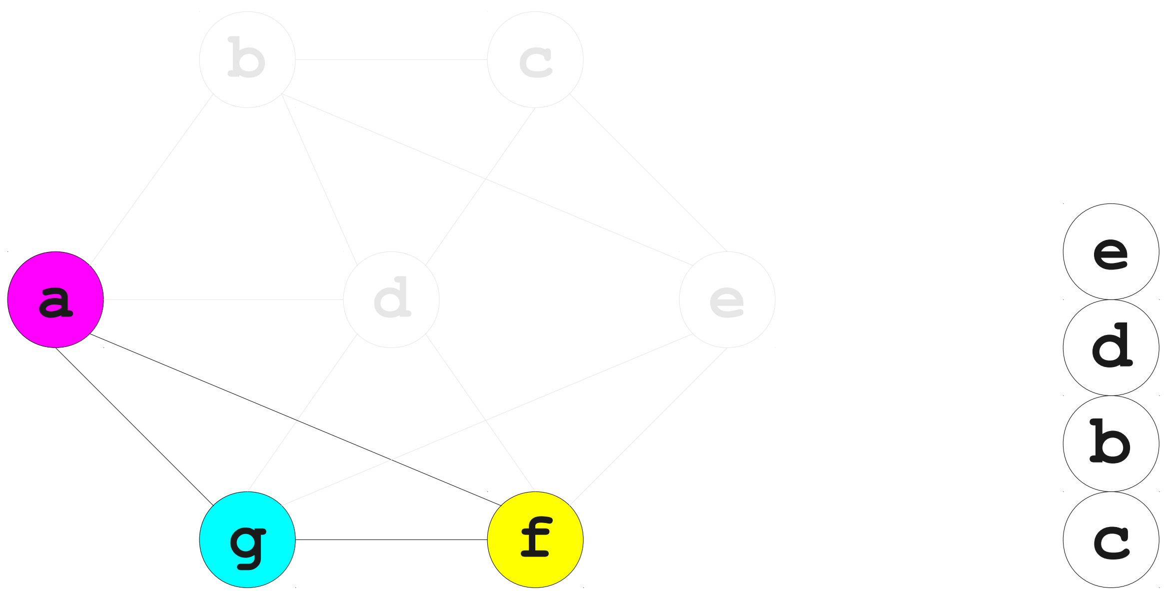
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

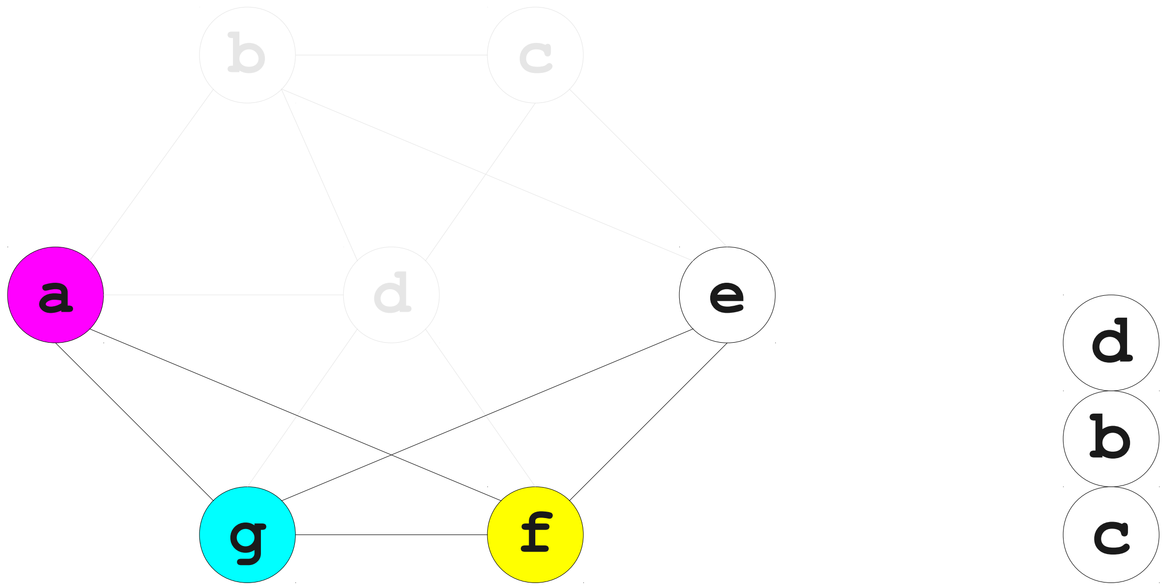
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

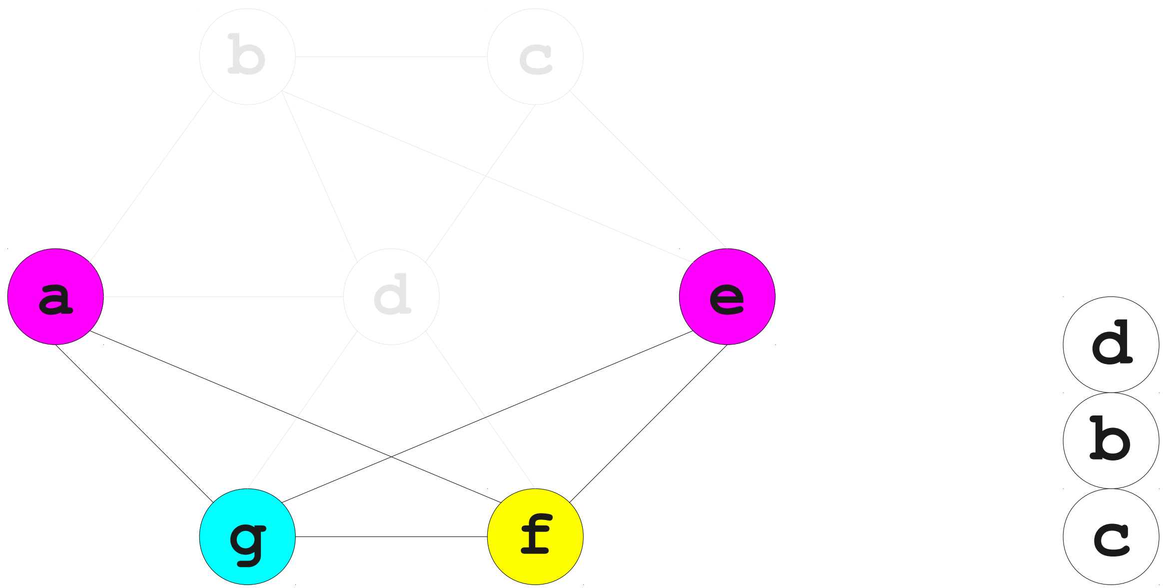
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

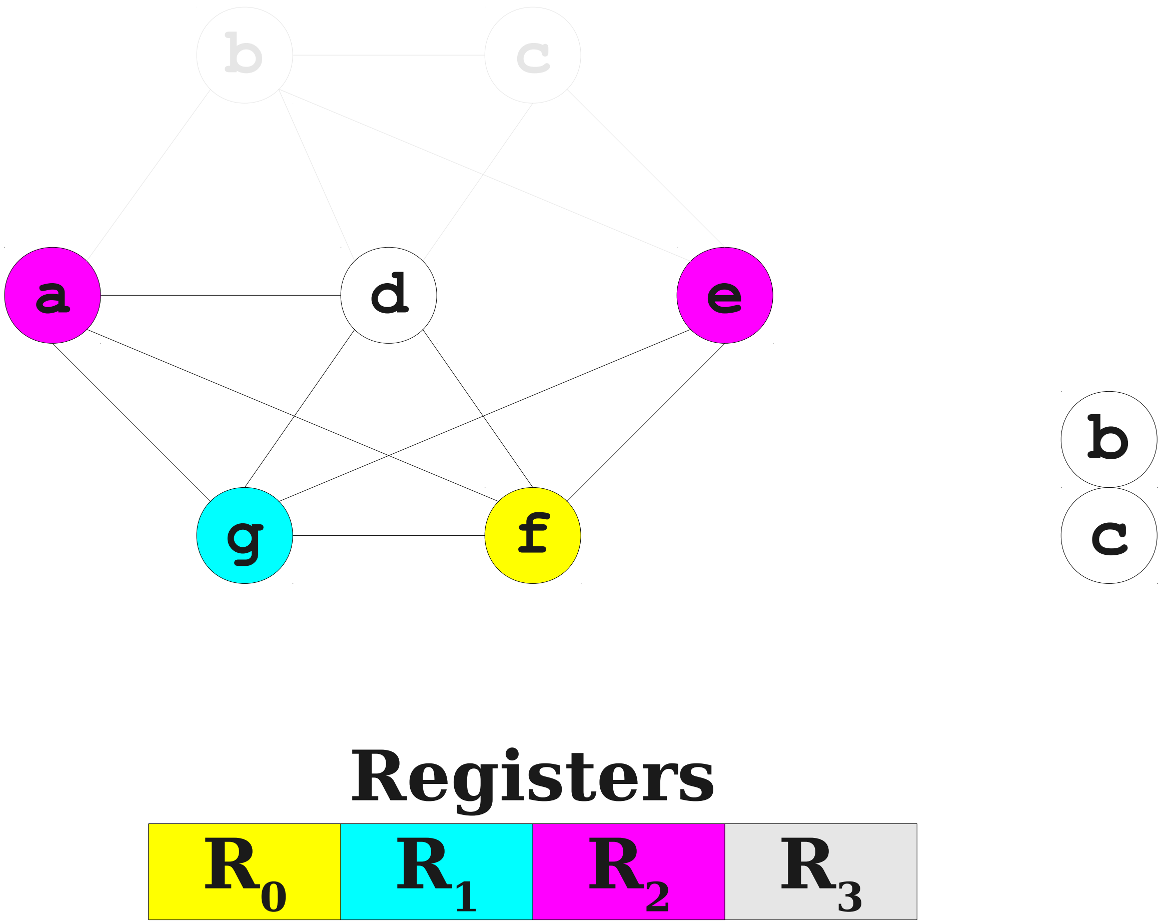
Chaitin's Algorithm



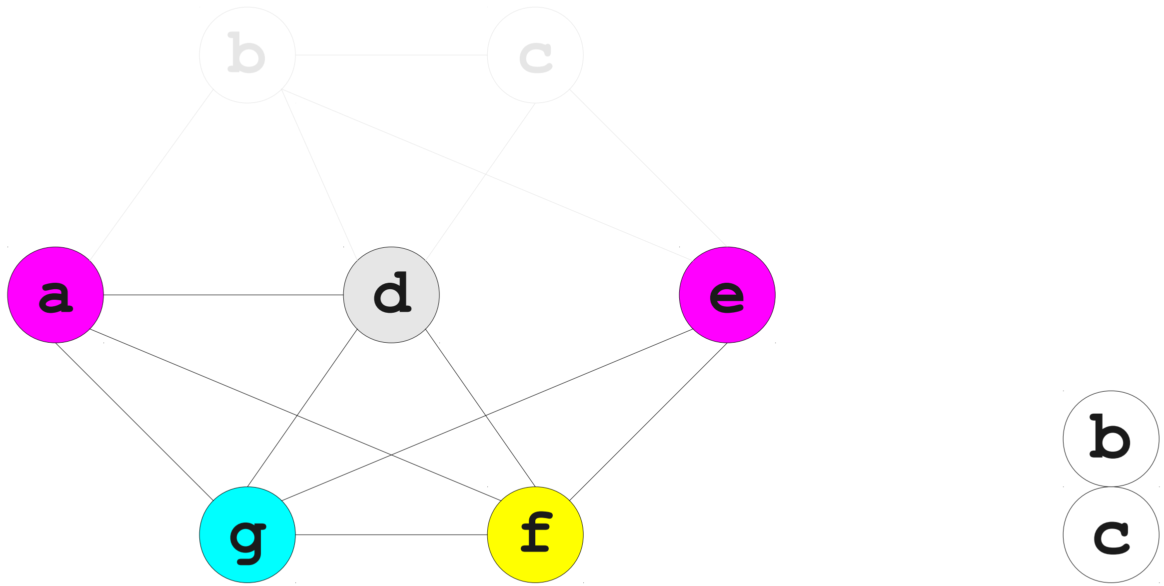
Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



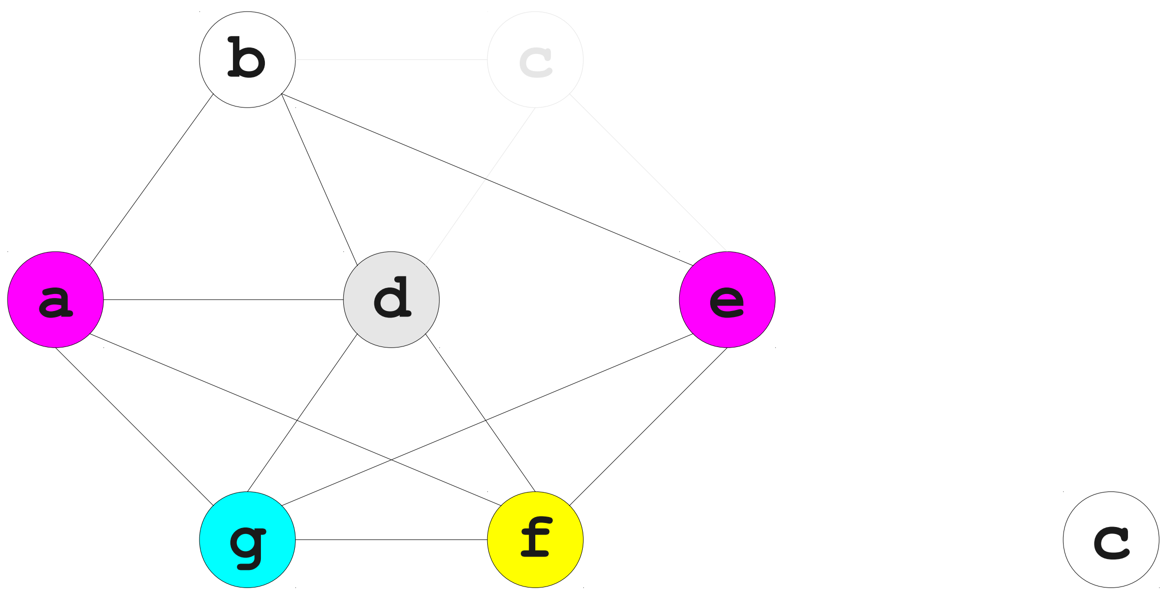
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

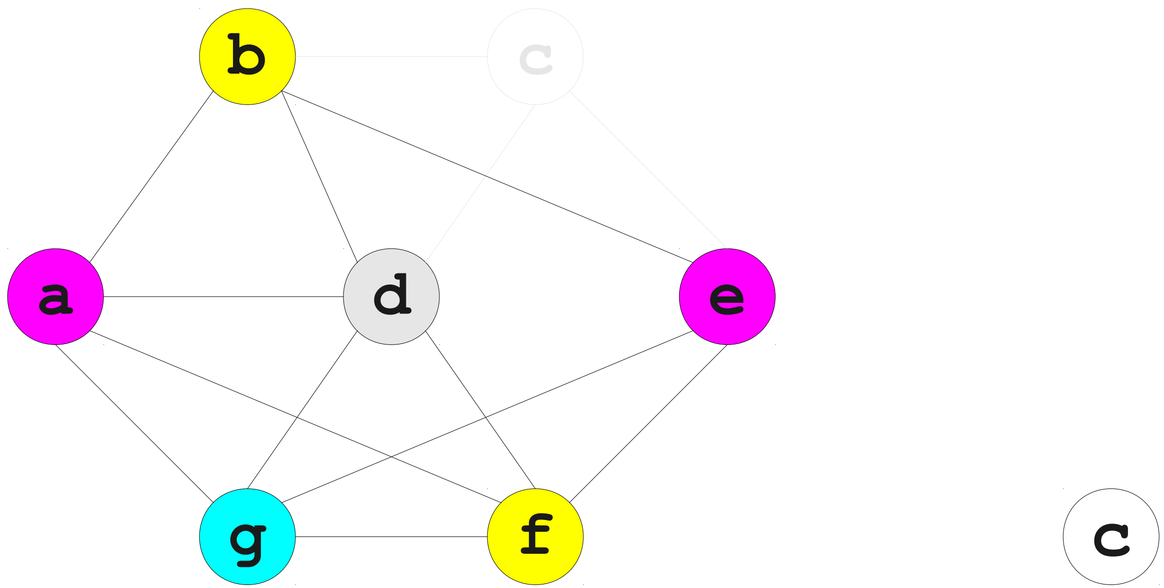
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

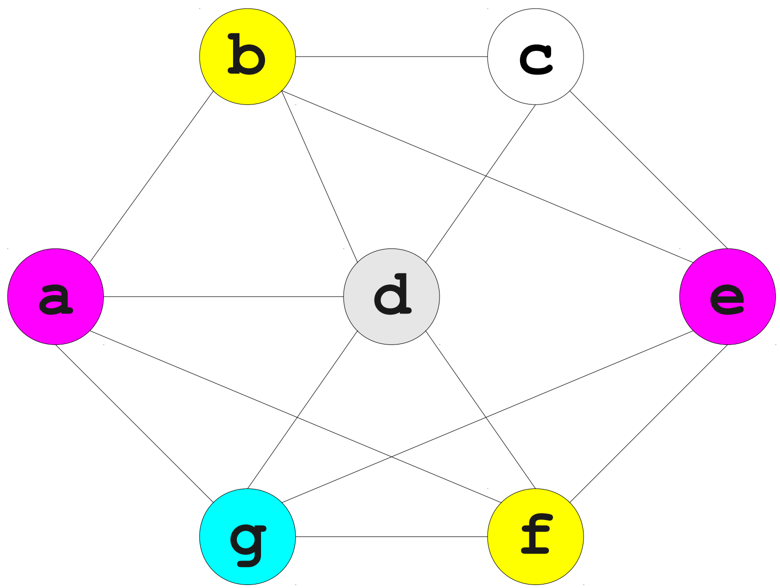
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

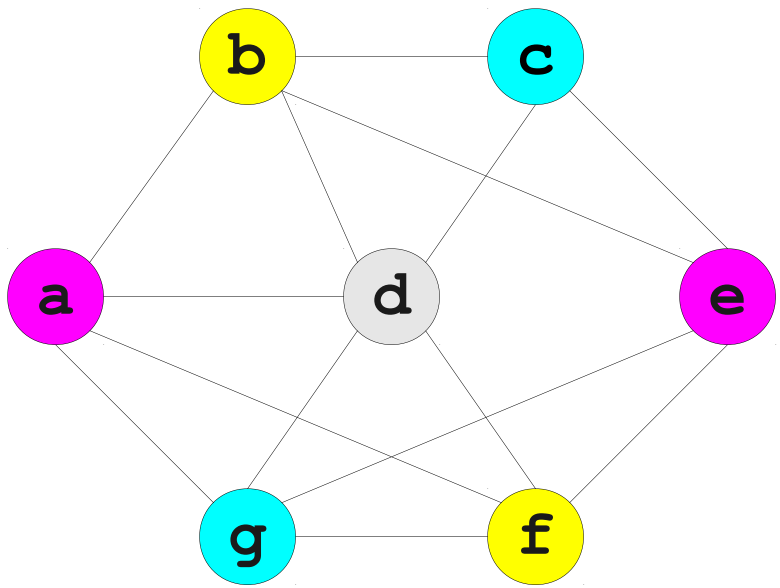
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



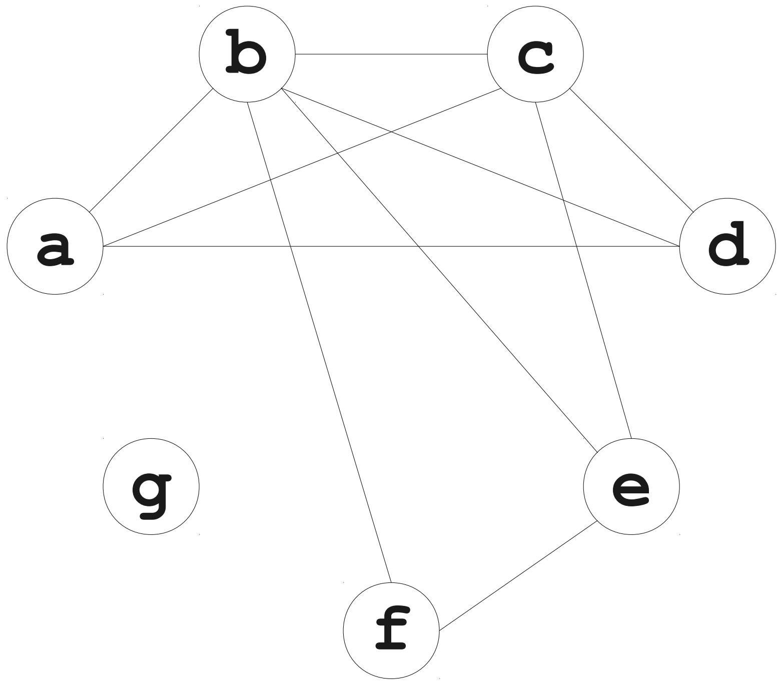
Registers

R_0	R_1	R_2	R_3

One Problem

- What if we can't find a node with fewer than *k* neighbors?
- Choose and remove an arbitrary node, marking it “troublesome.”
  - Use heuristics to choose which one.
- When adding node back in, it may be possible to find a valid color.
- Otherwise, we have to spill that node.

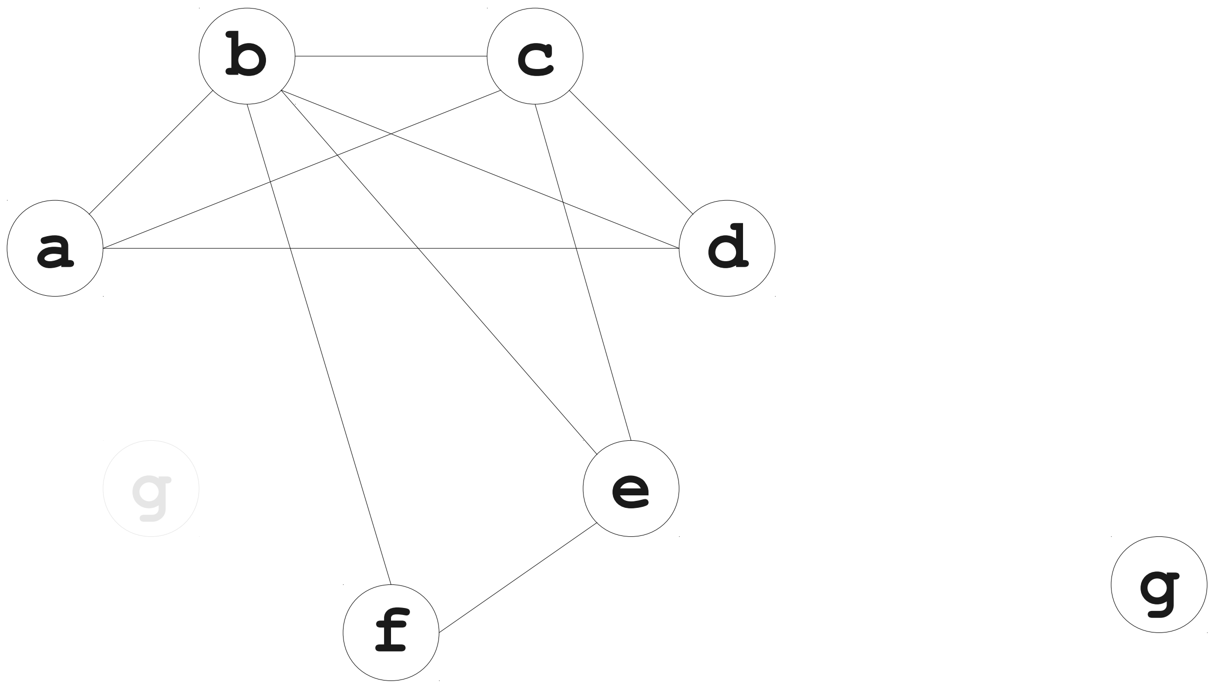
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

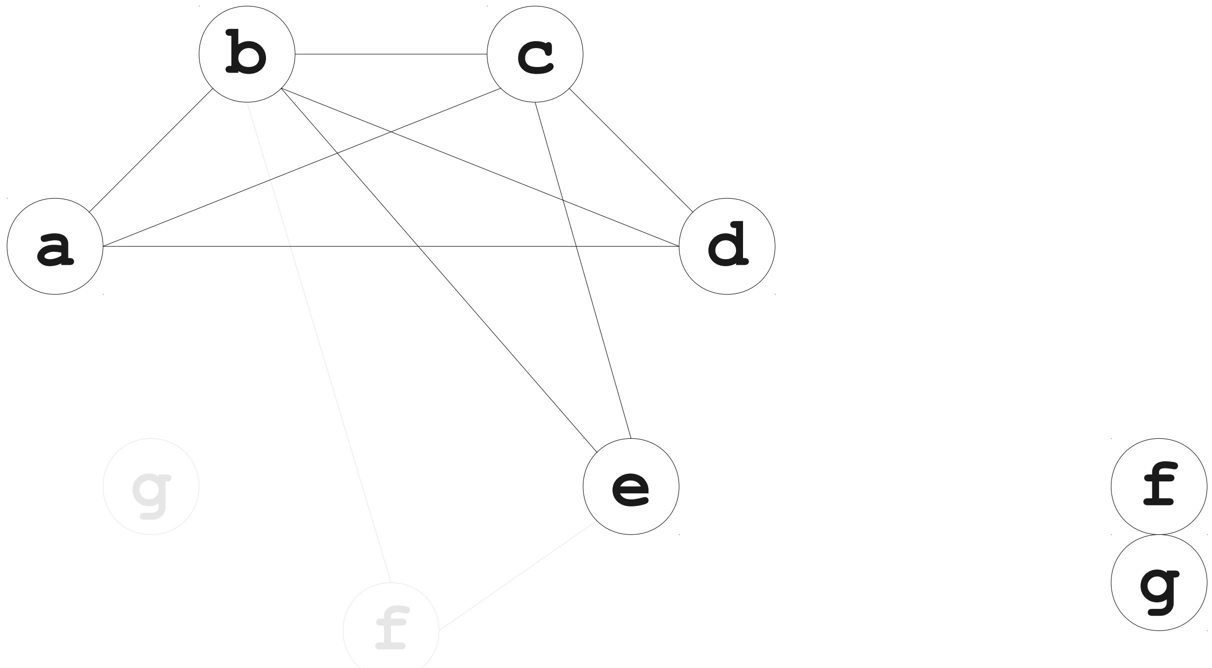
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

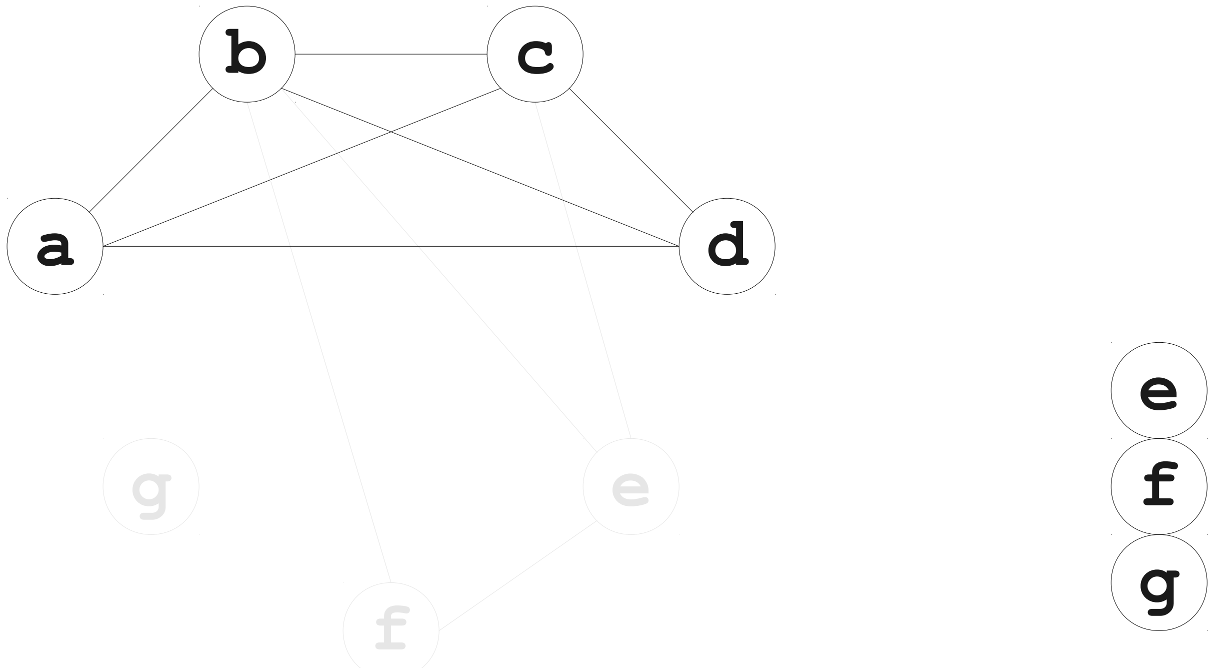
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

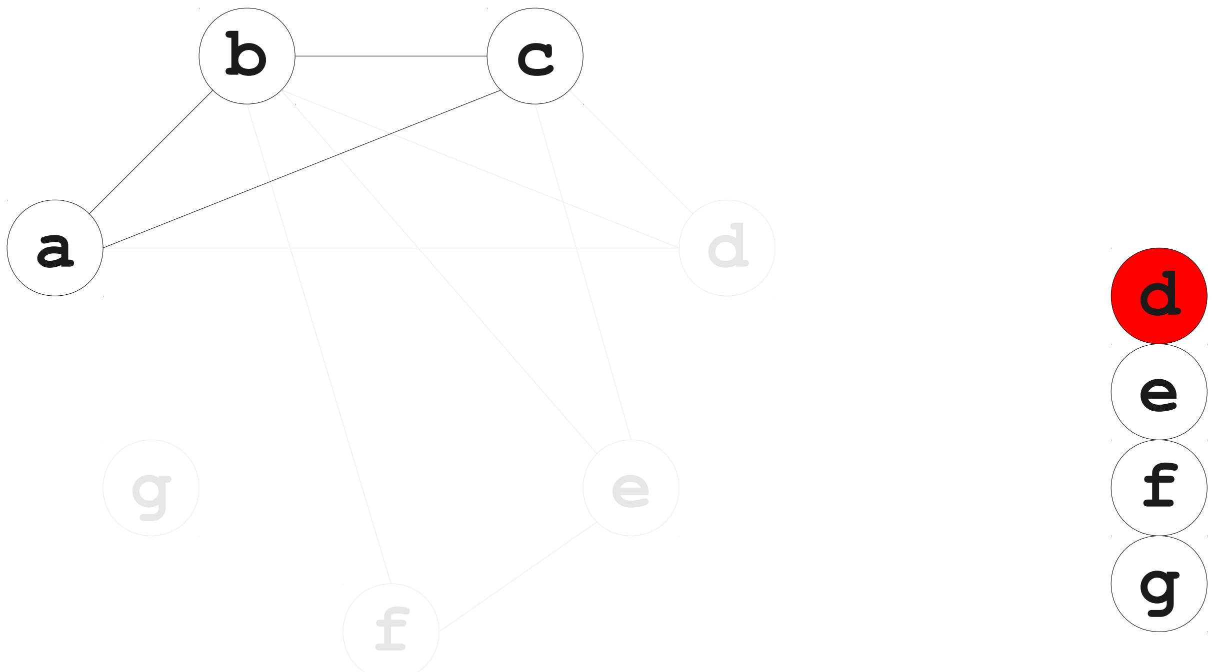
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



Registers

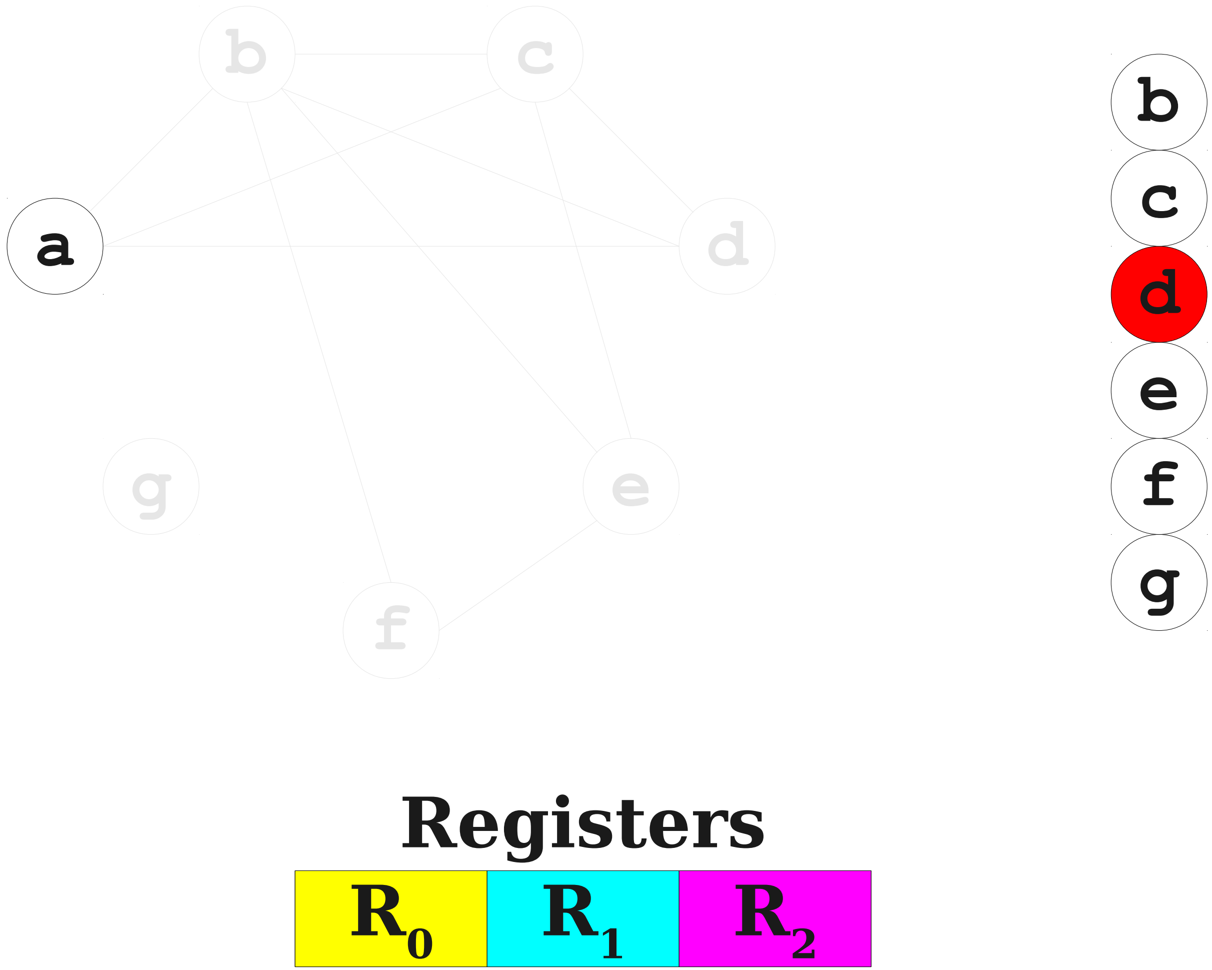
R_0	R_1	R_2

Chaitin's Algorithm Reloaded

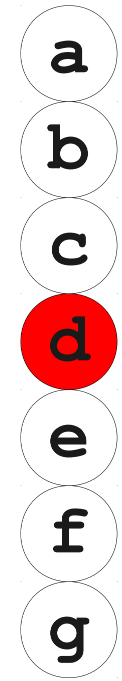
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



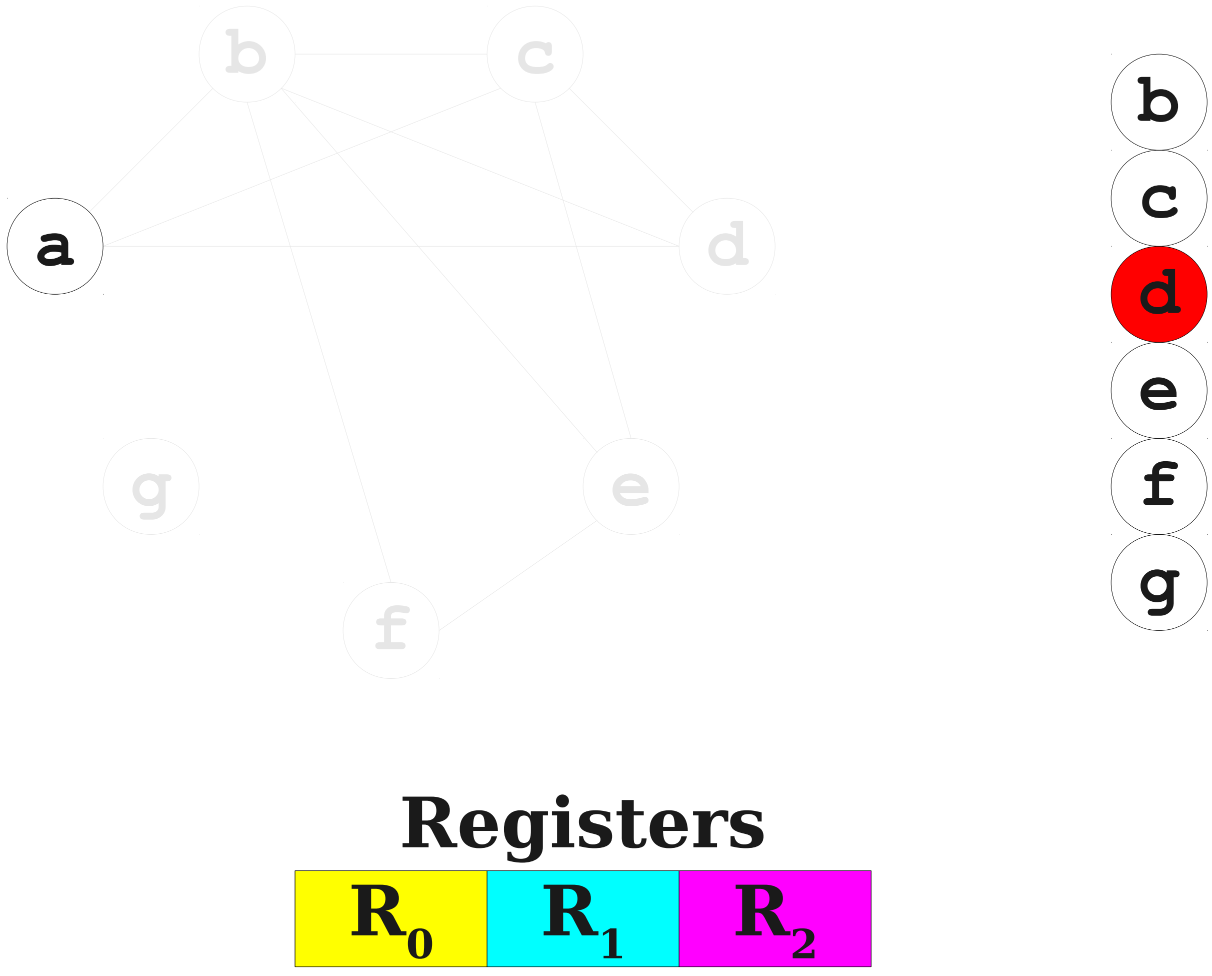
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



Chaitin's Algorithm Reloaded

Registers

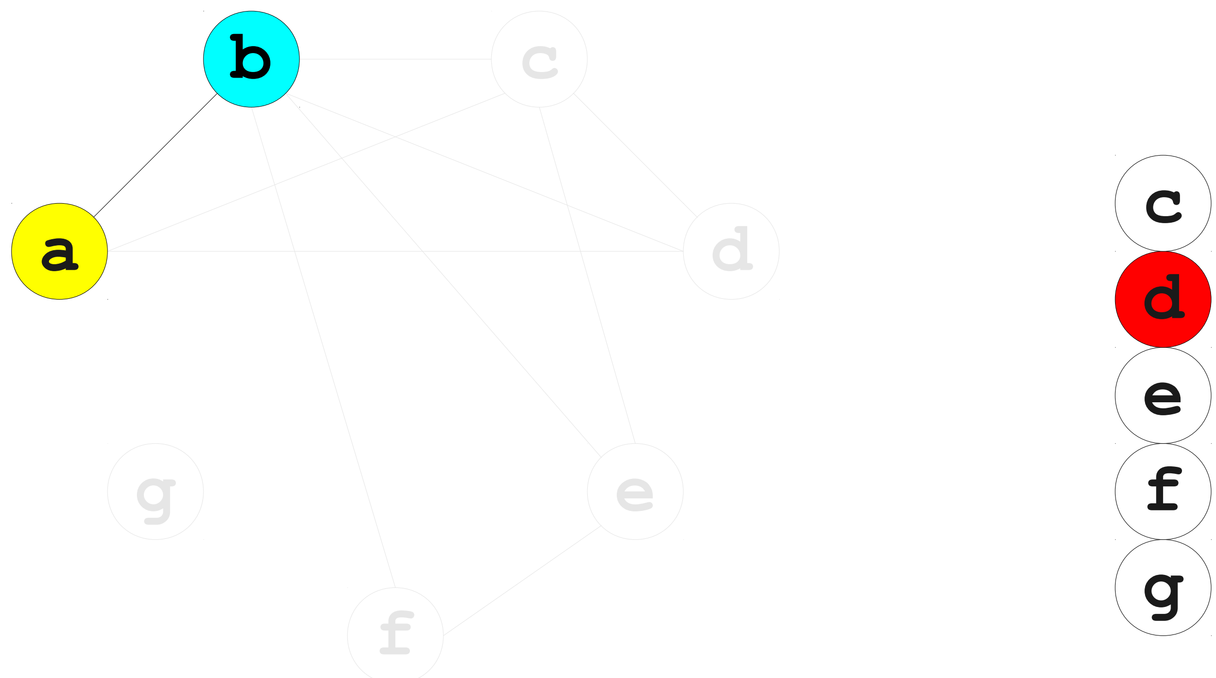
R_0	R_1	R_2

Chaitin's Algorithm Reloaded

Registers

R_0	R_1	R_2

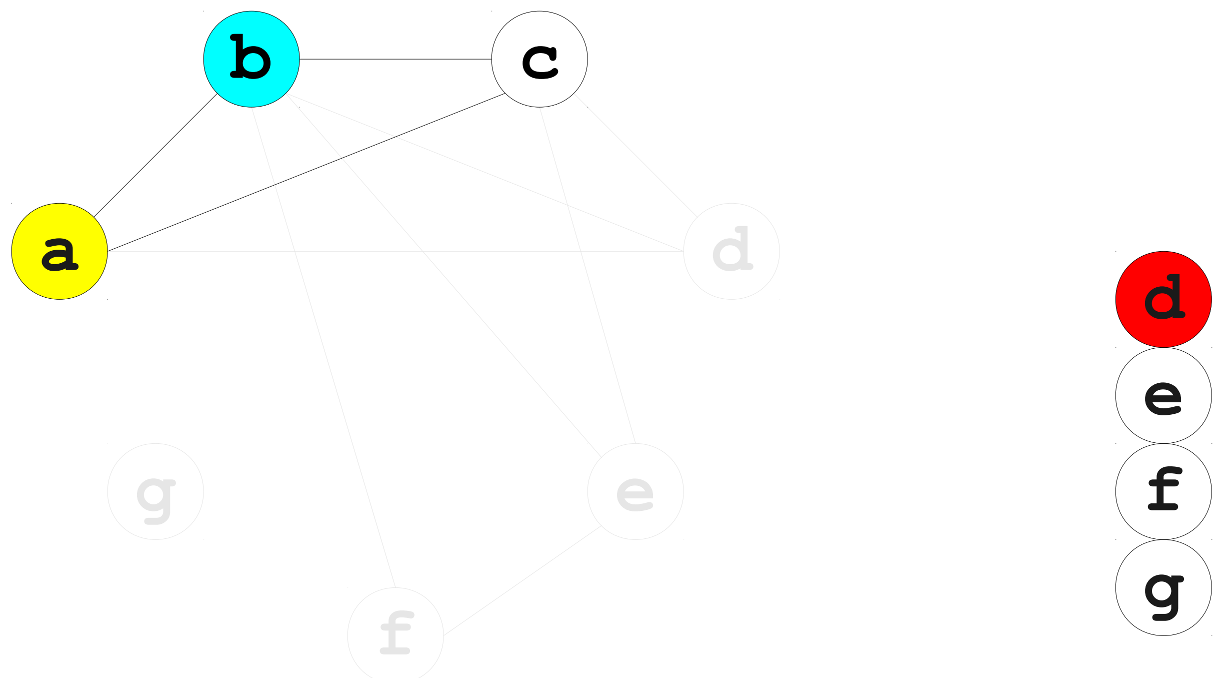
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

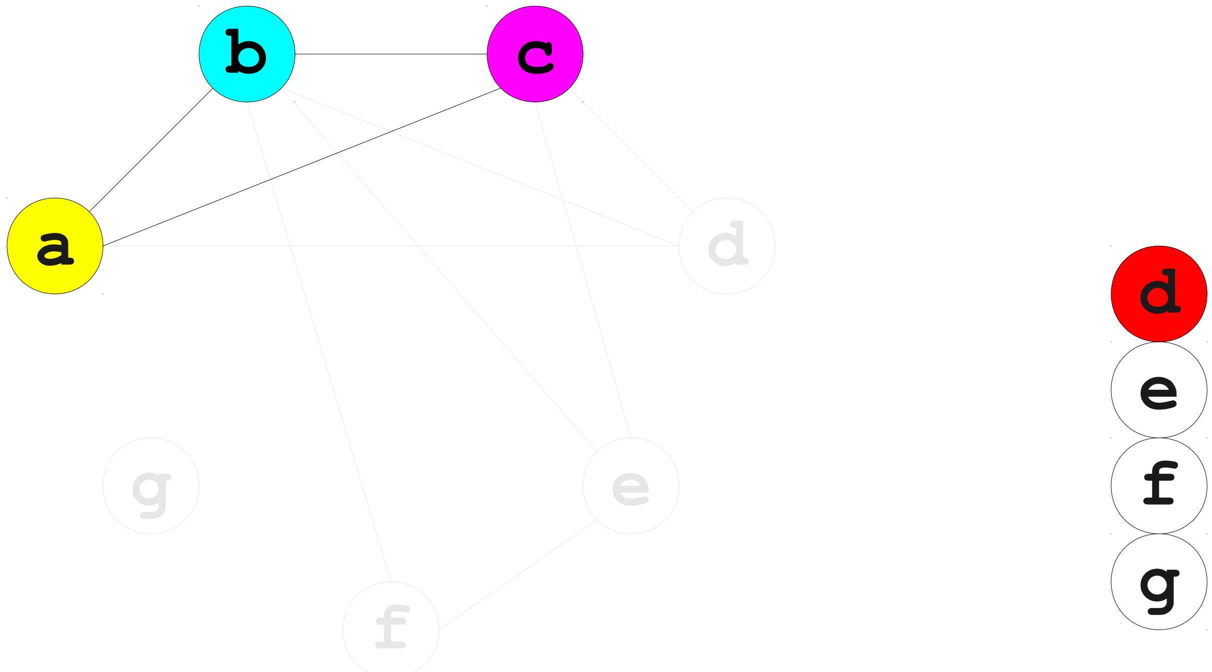
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

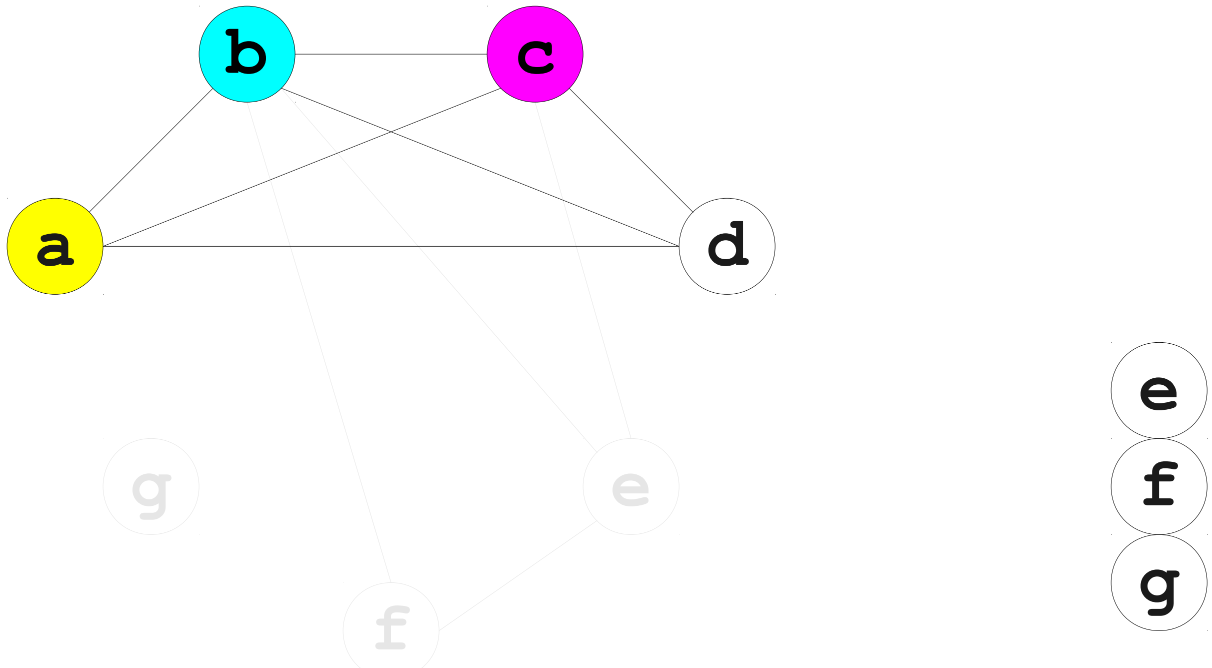
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

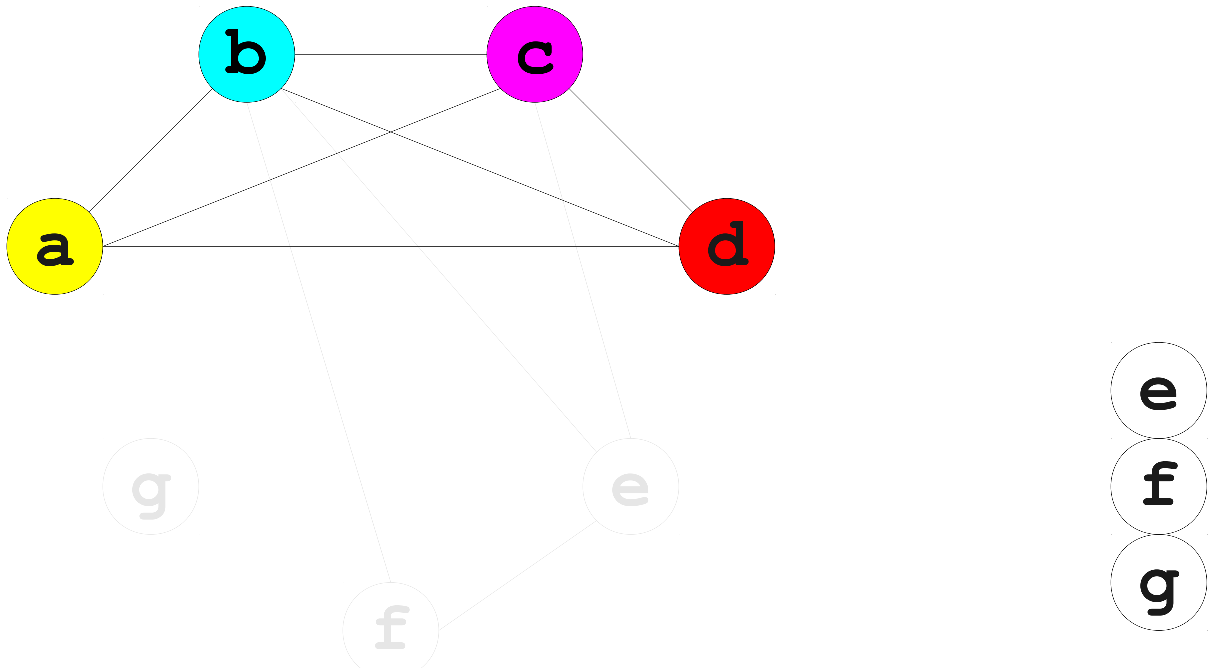
Chaitin's Algorithm Reloaded



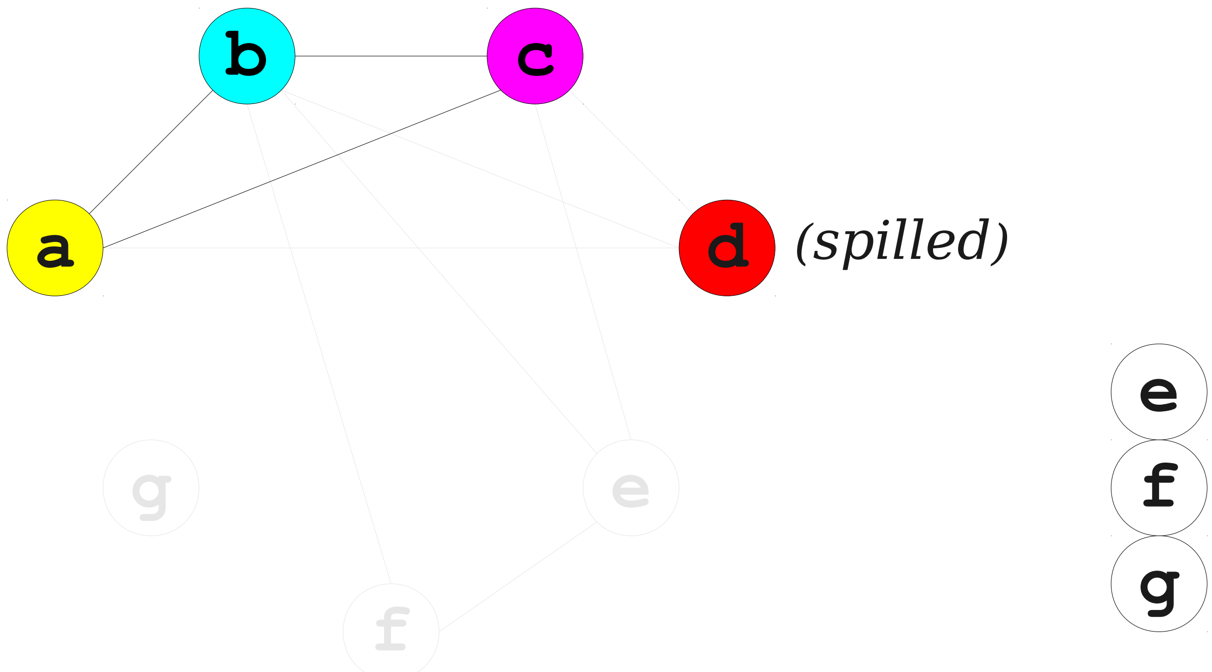
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



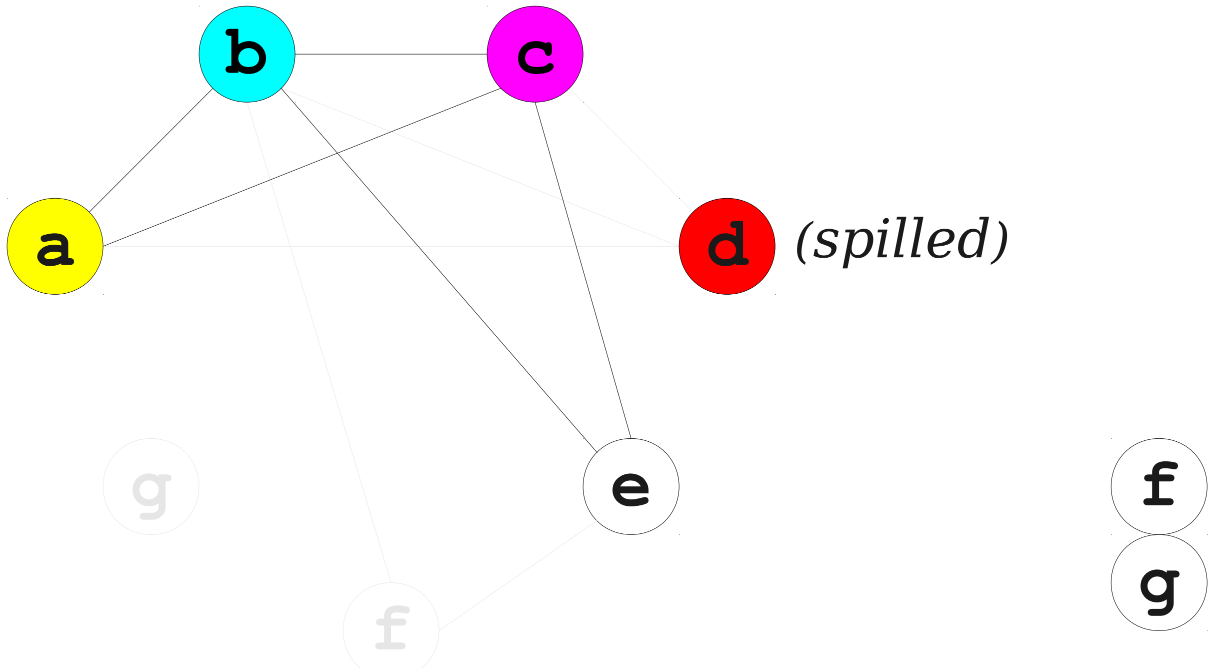
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

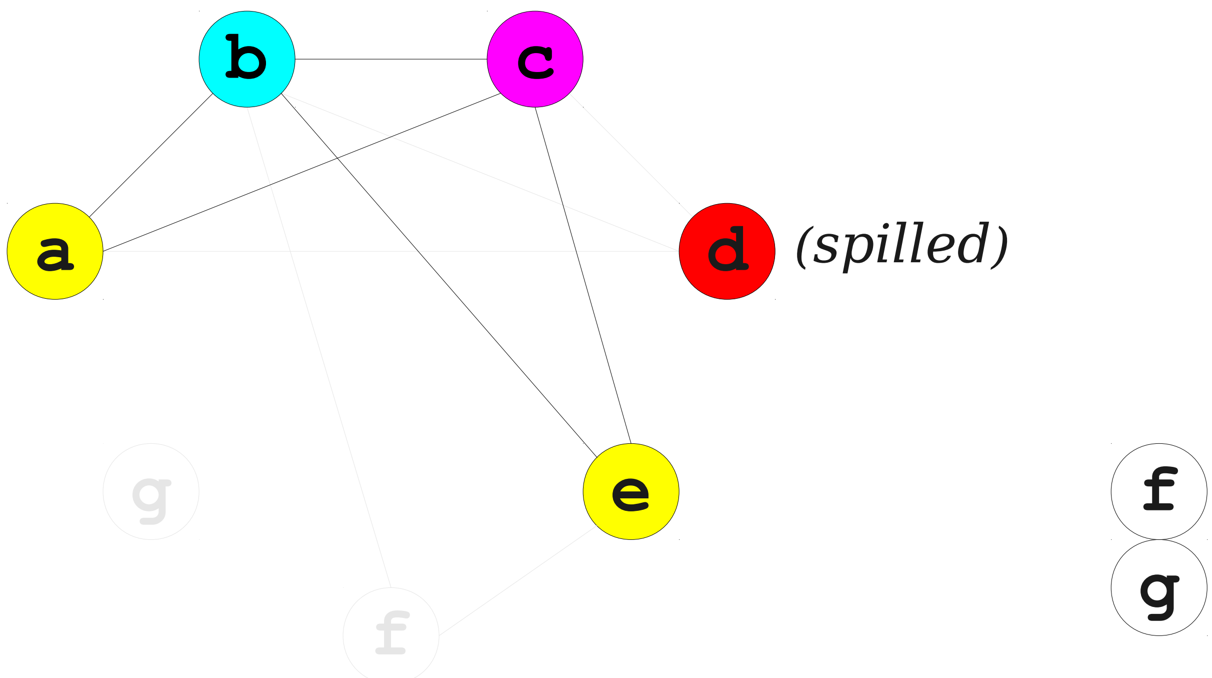
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

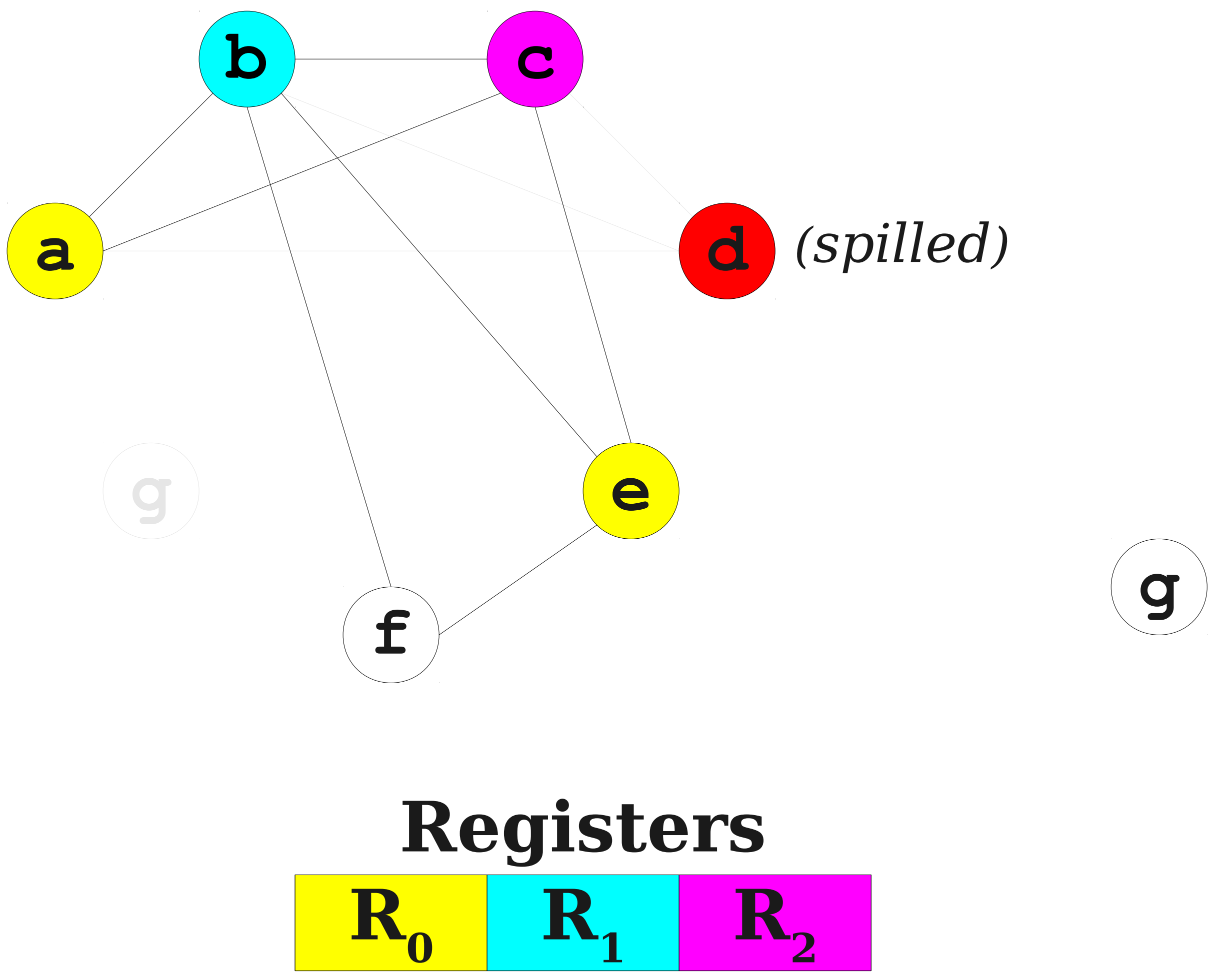
Chaitin's Algorithm Reloaded



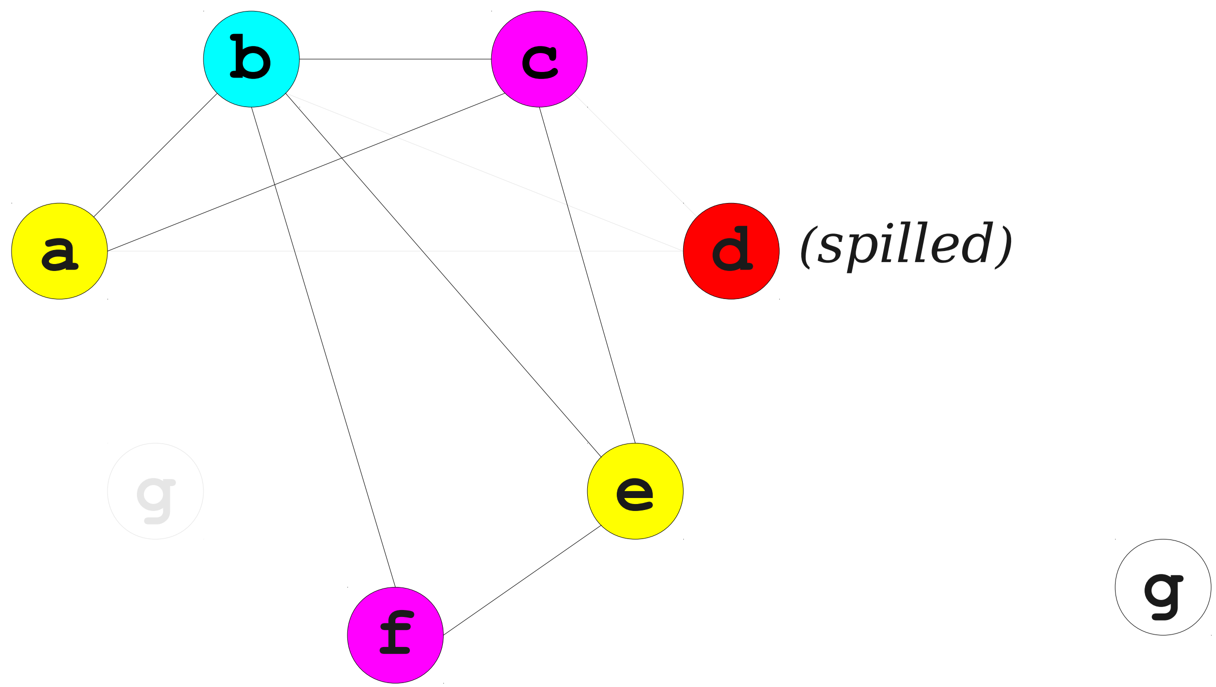
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded



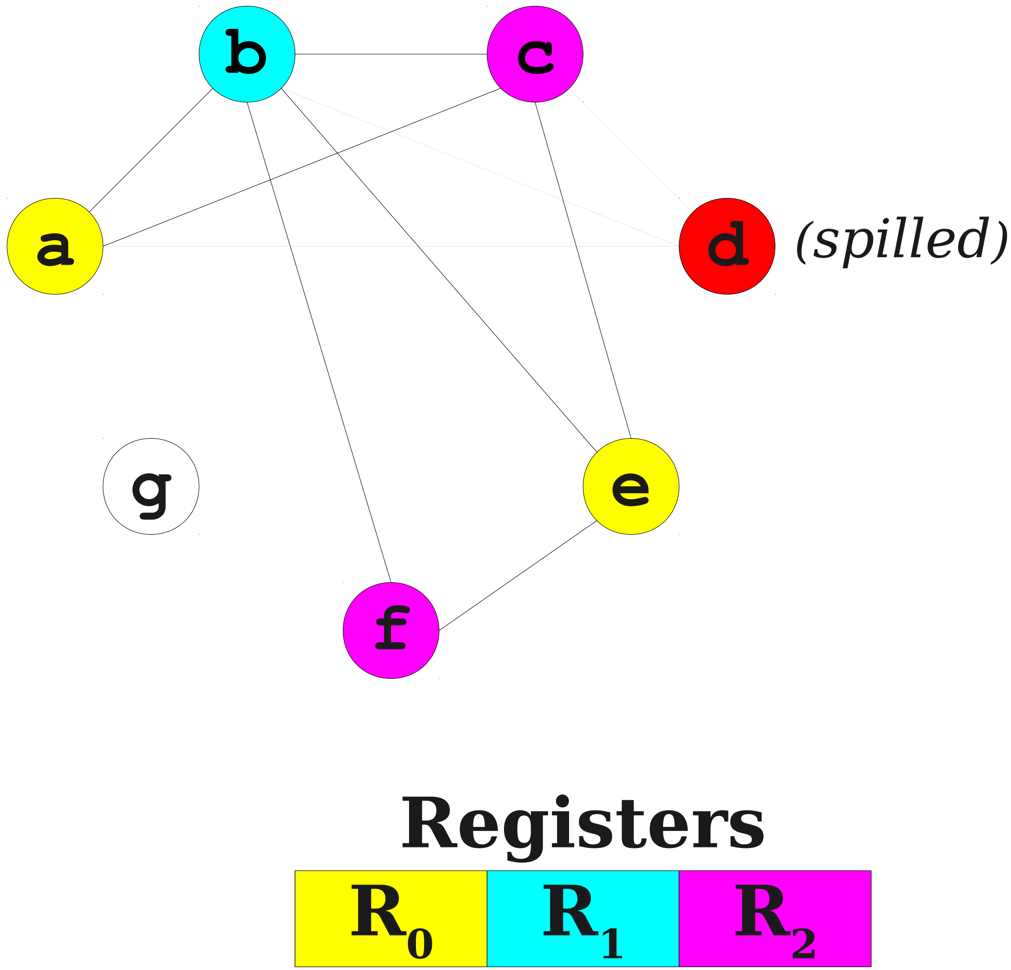
Chaitin's Algorithm Reloaded



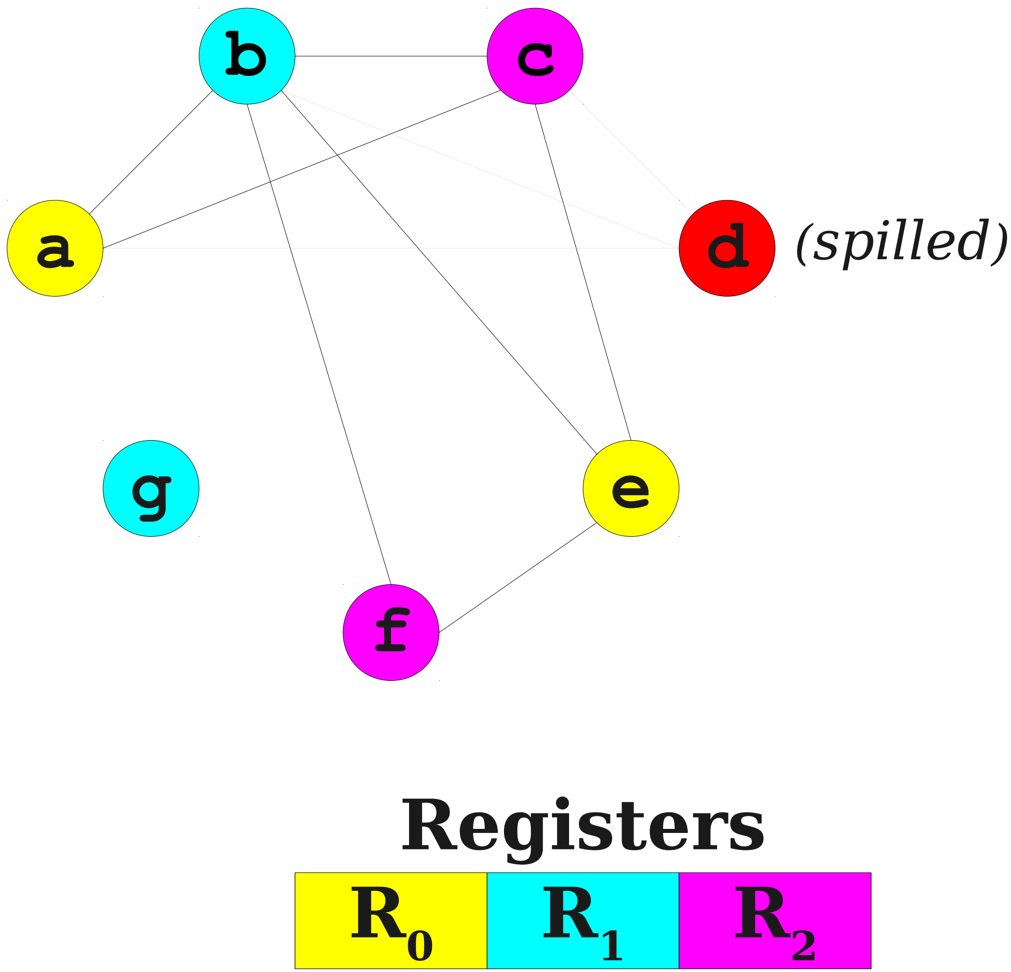
Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded

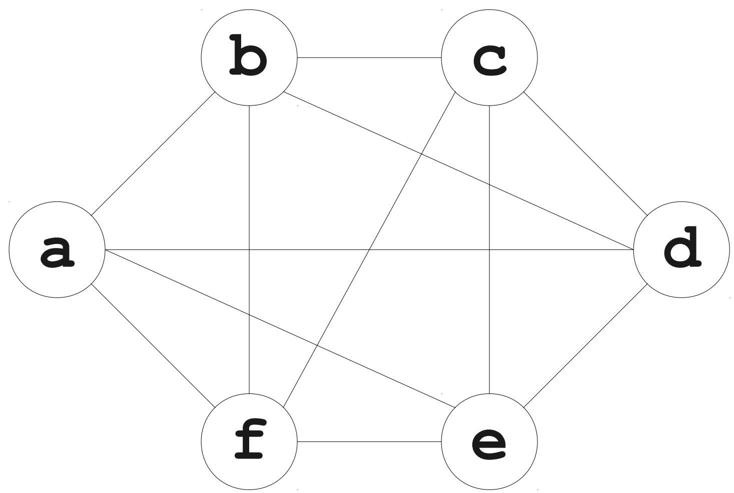


Chaitin's Algorithm Reloaded

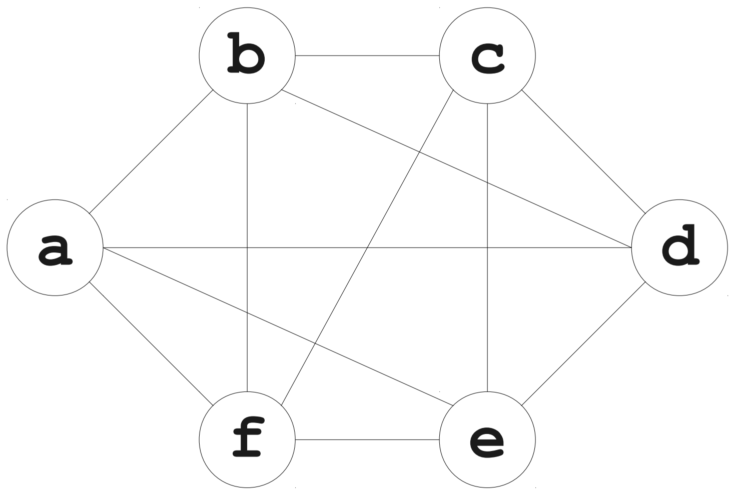


Another Example

Another Example



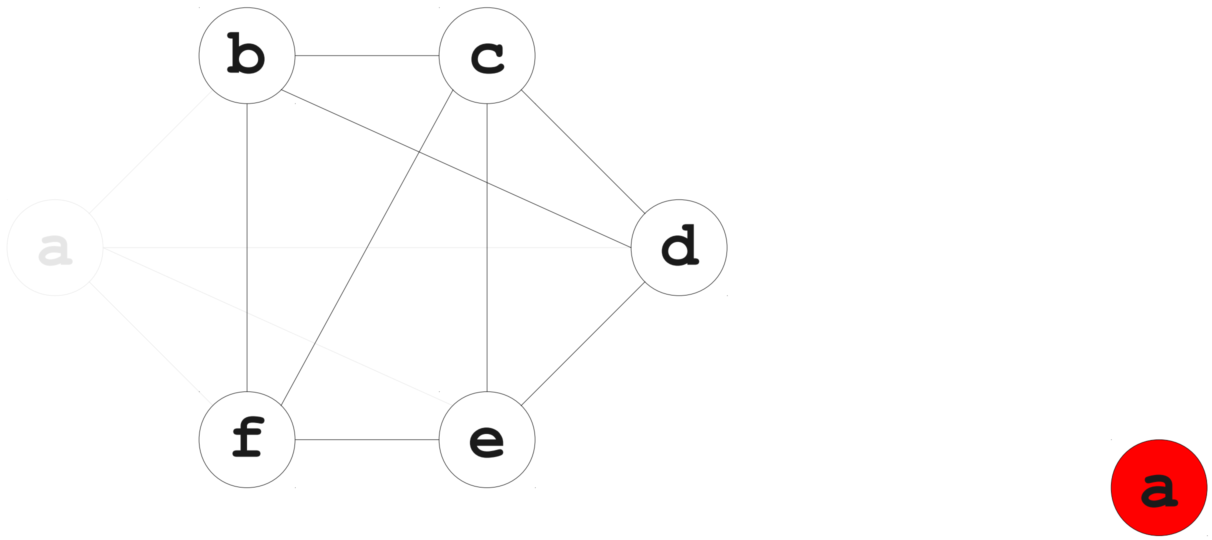
Another Example



Registers

R_0	R_1	R_2

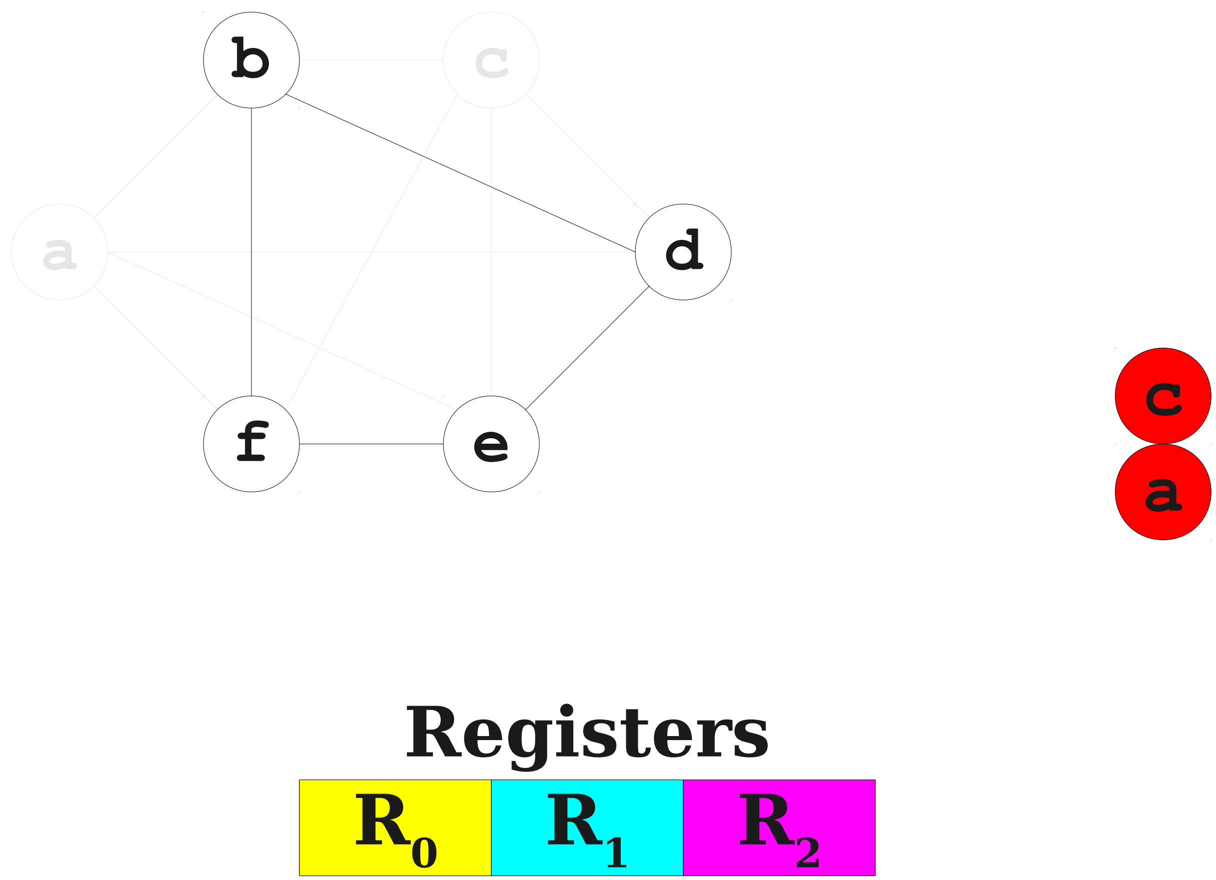
Another Example



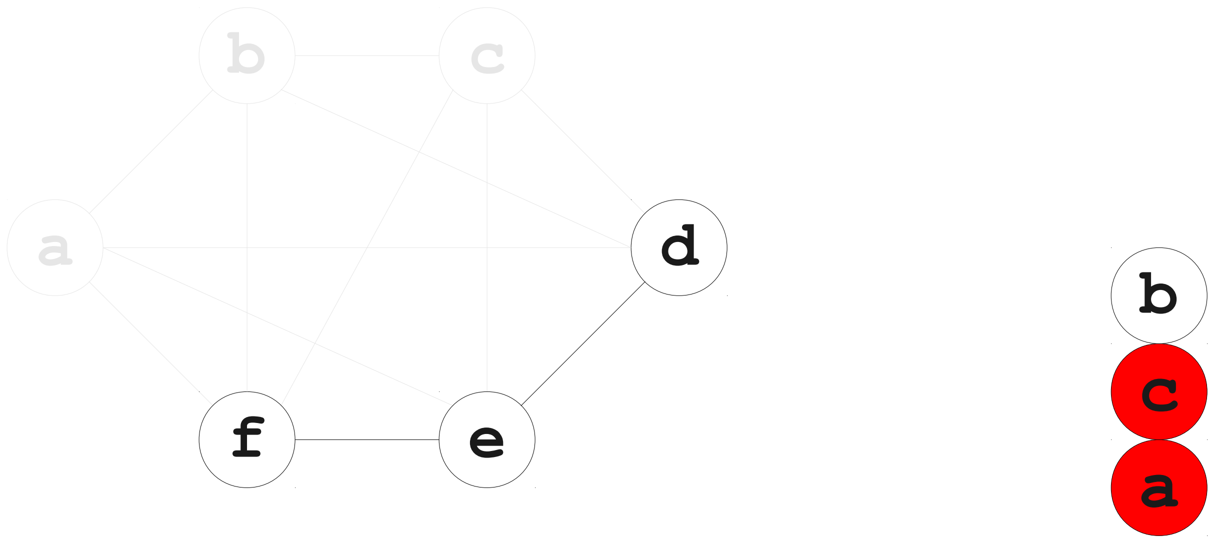
Registers

R_0	R_1	R_2

Another Example



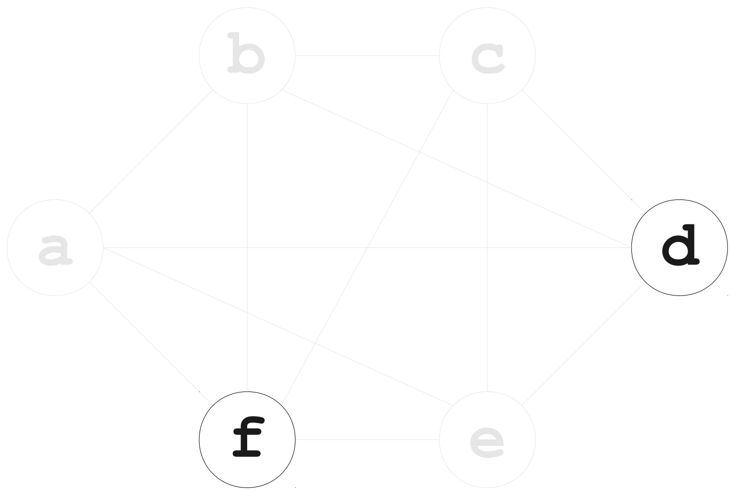
Another Example

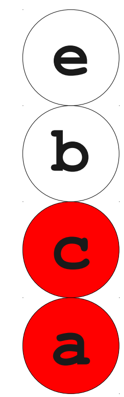


Registers

R_0	R_1	R_2

Another Example

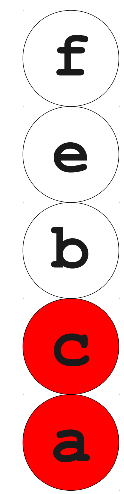




Registers

R_0	R_1	R_2

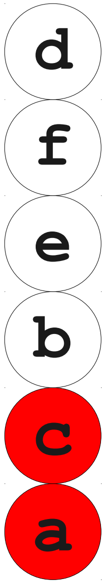
Another Example



Registers

R_0	R_1	R_2

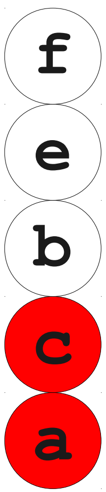
Another Example



Registers

R_0	R_1	R_2

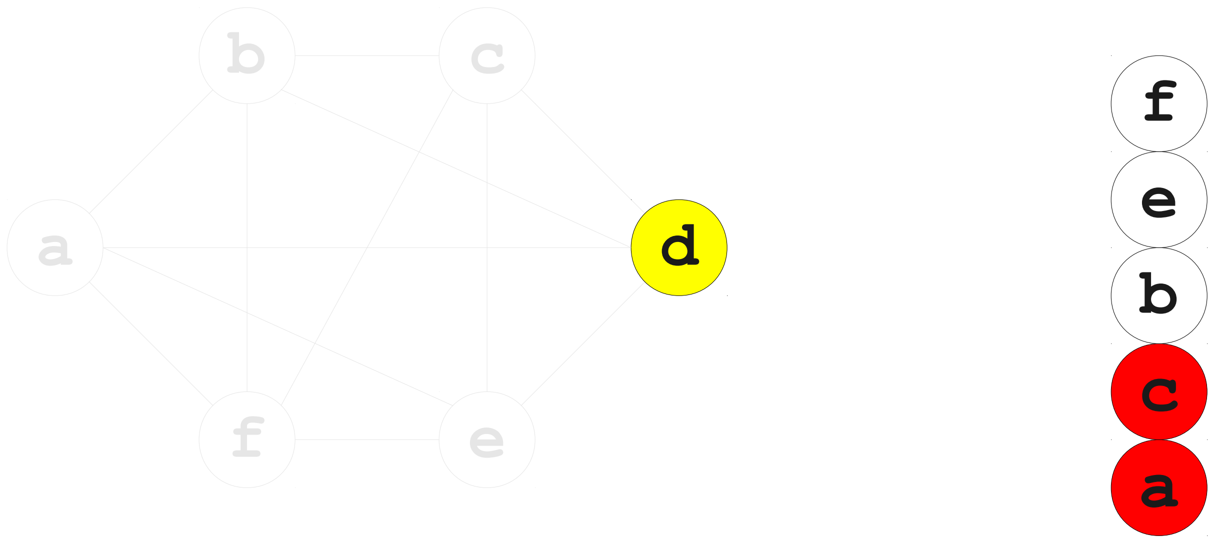
Another Example



Registers

R_0	R_1	R_2

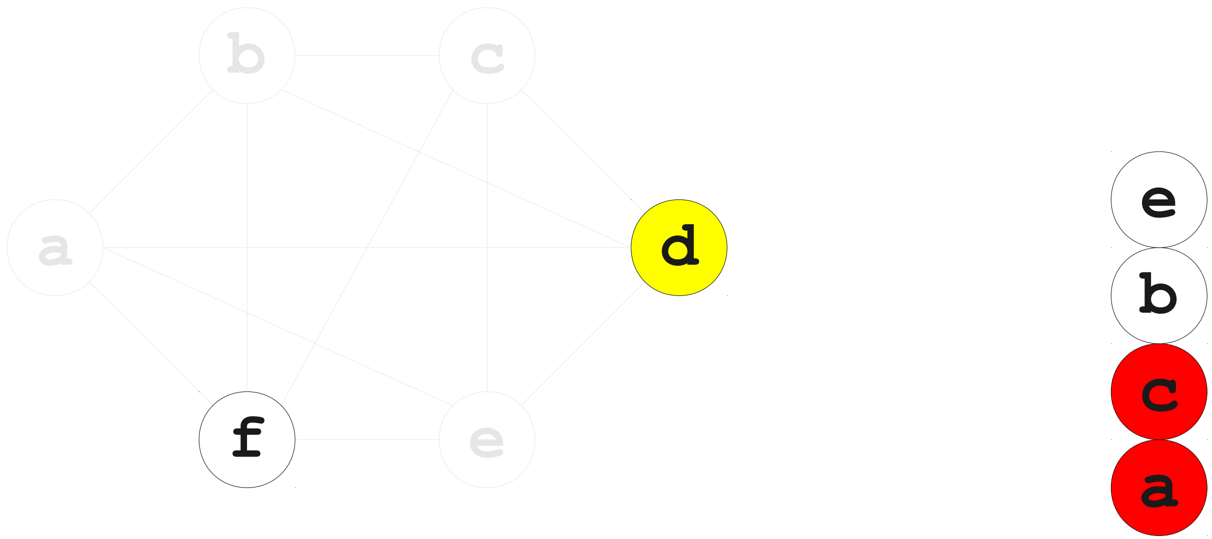
Another Example



Registers

R_0	R_1	R_2

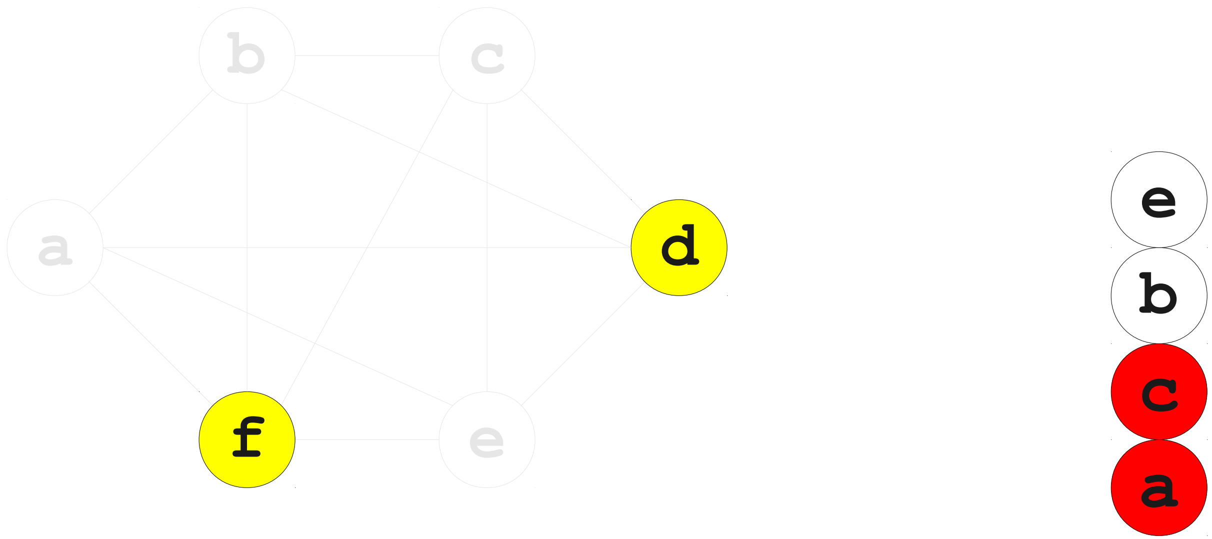
Another Example



Registers

R_0	R_1	R_2

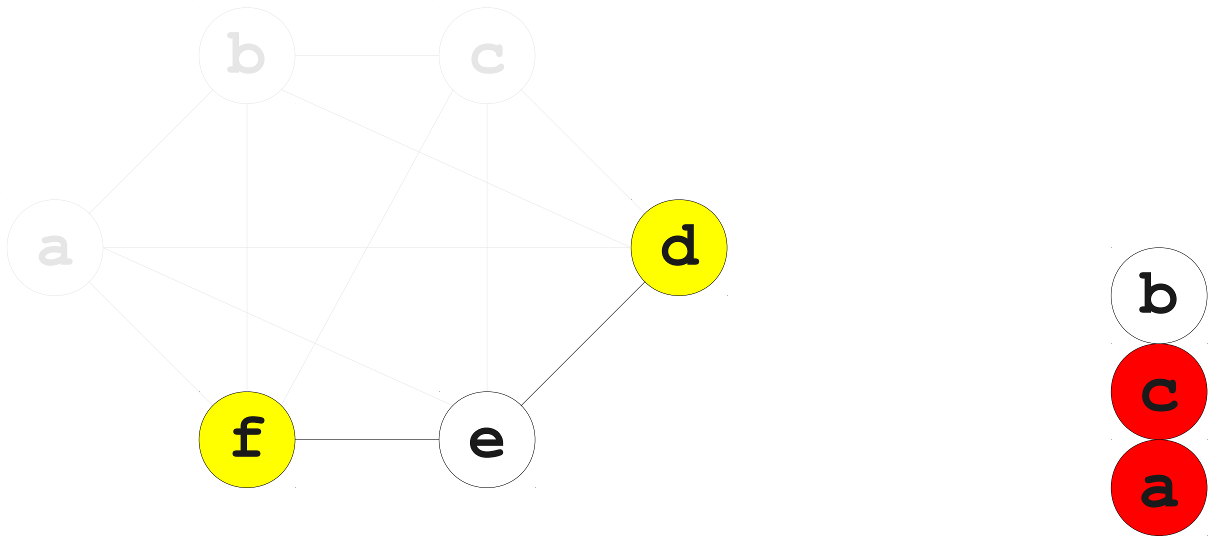
Another Example



Registers

R_0	R_1	R_2

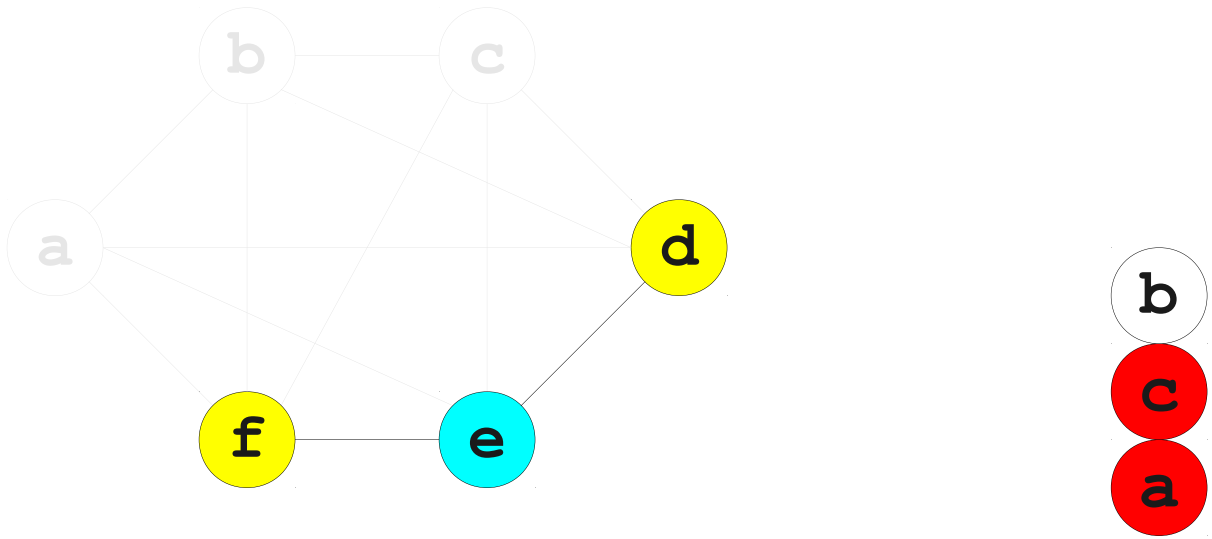
Another Example



Registers

R_0	R_1	R_2

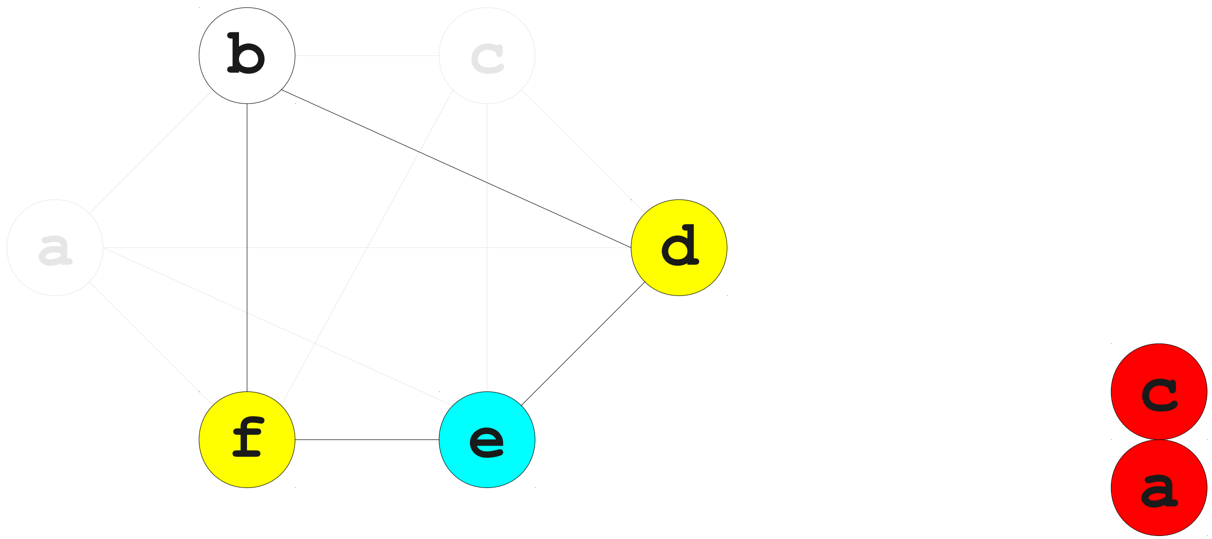
Another Example



Registers

R_0	R_1	R_2

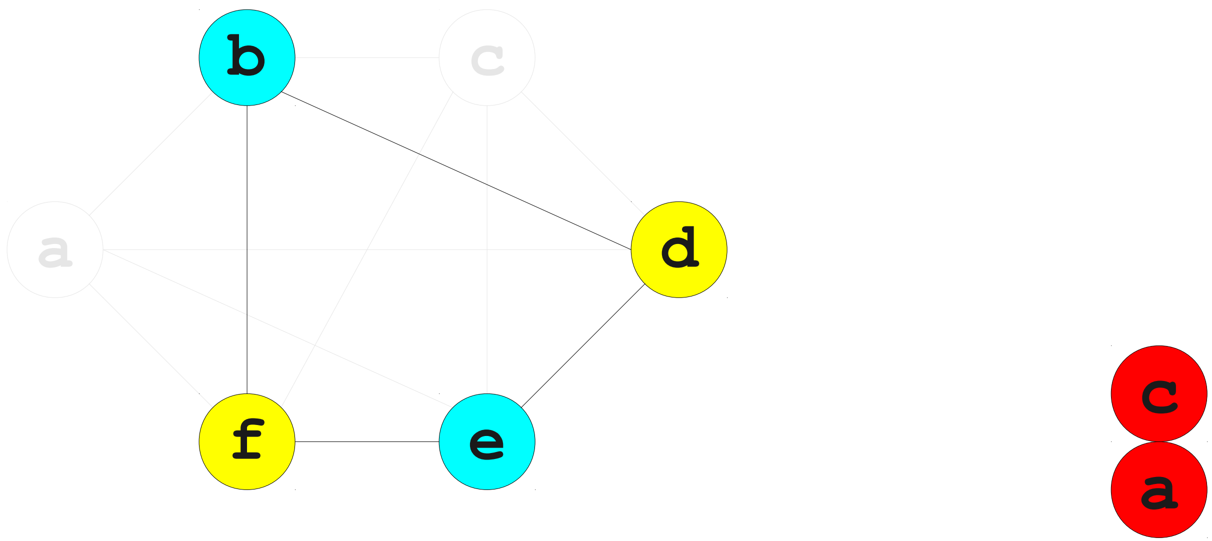
Another Example



Registers

R_0	R_1	R_2

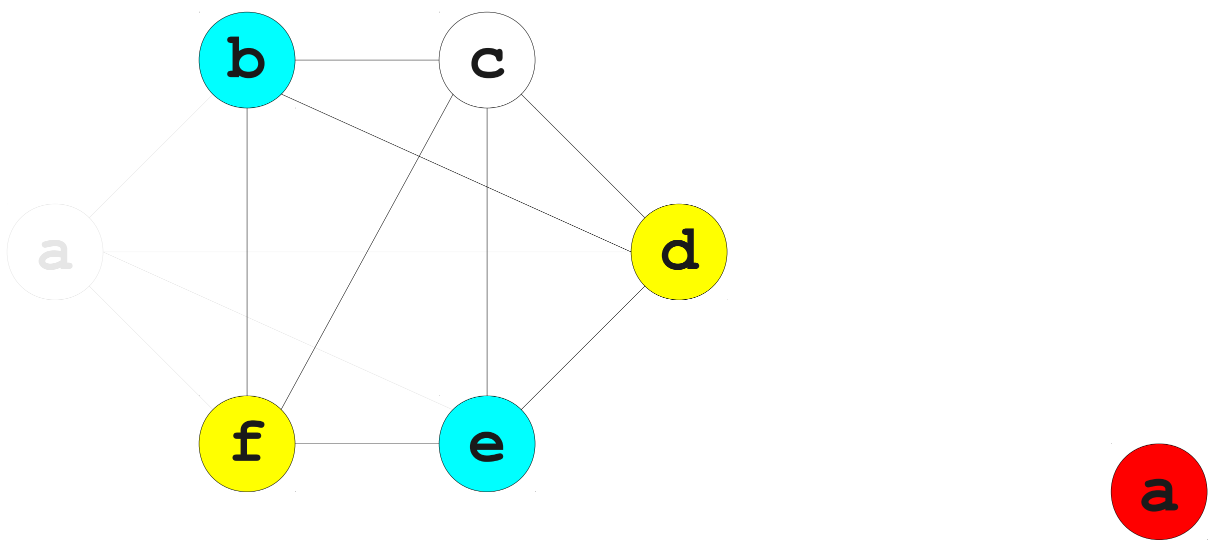
Another Example



Registers

R_0	R_1	R_2

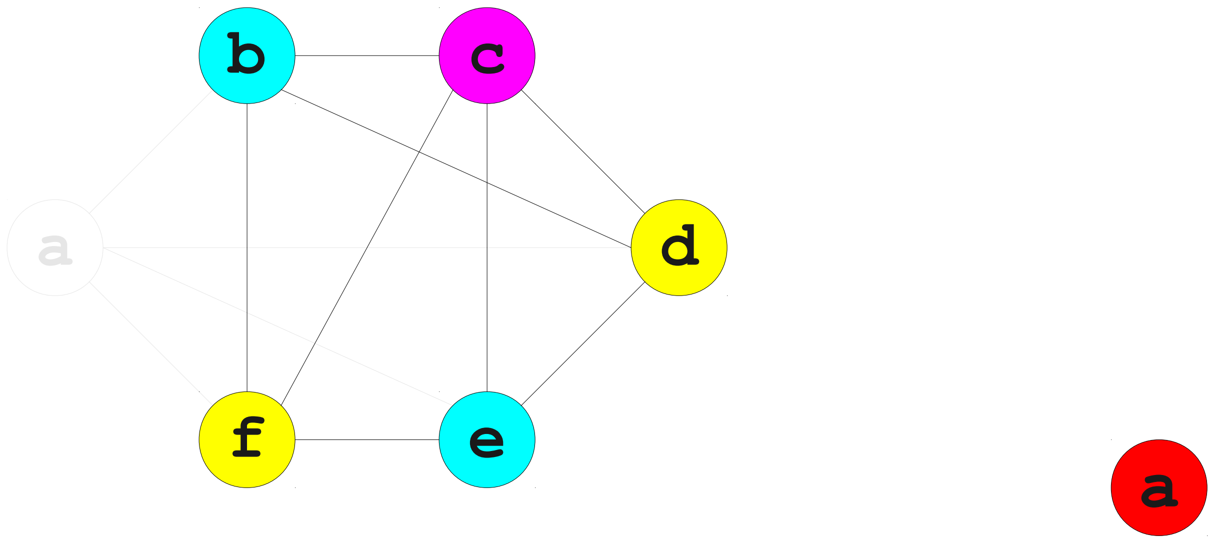
Another Example



Registers

R_0	R_1	R_2

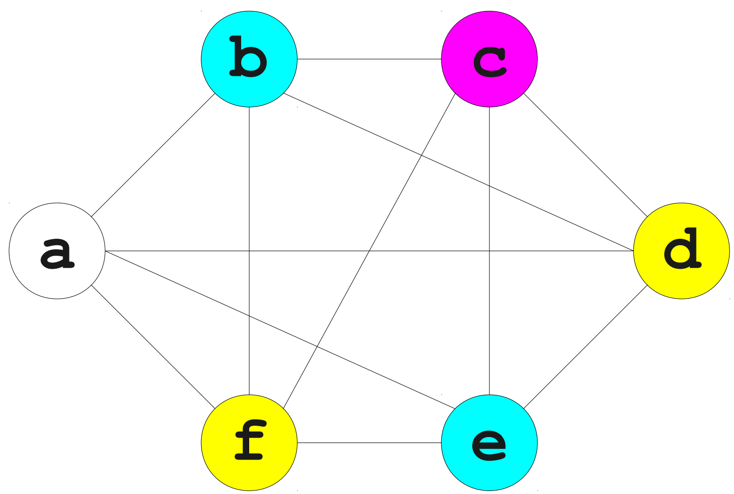
Another Example



Registers

R_0	R_1	R_2

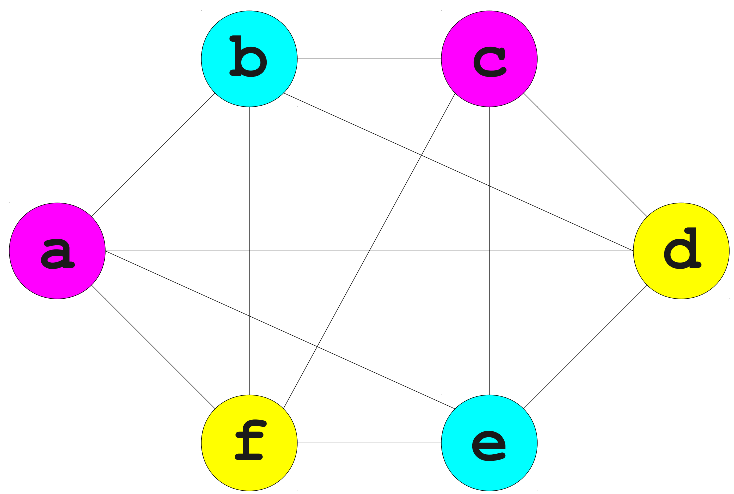
Another Example



Registers

R_0	R_1	R_2

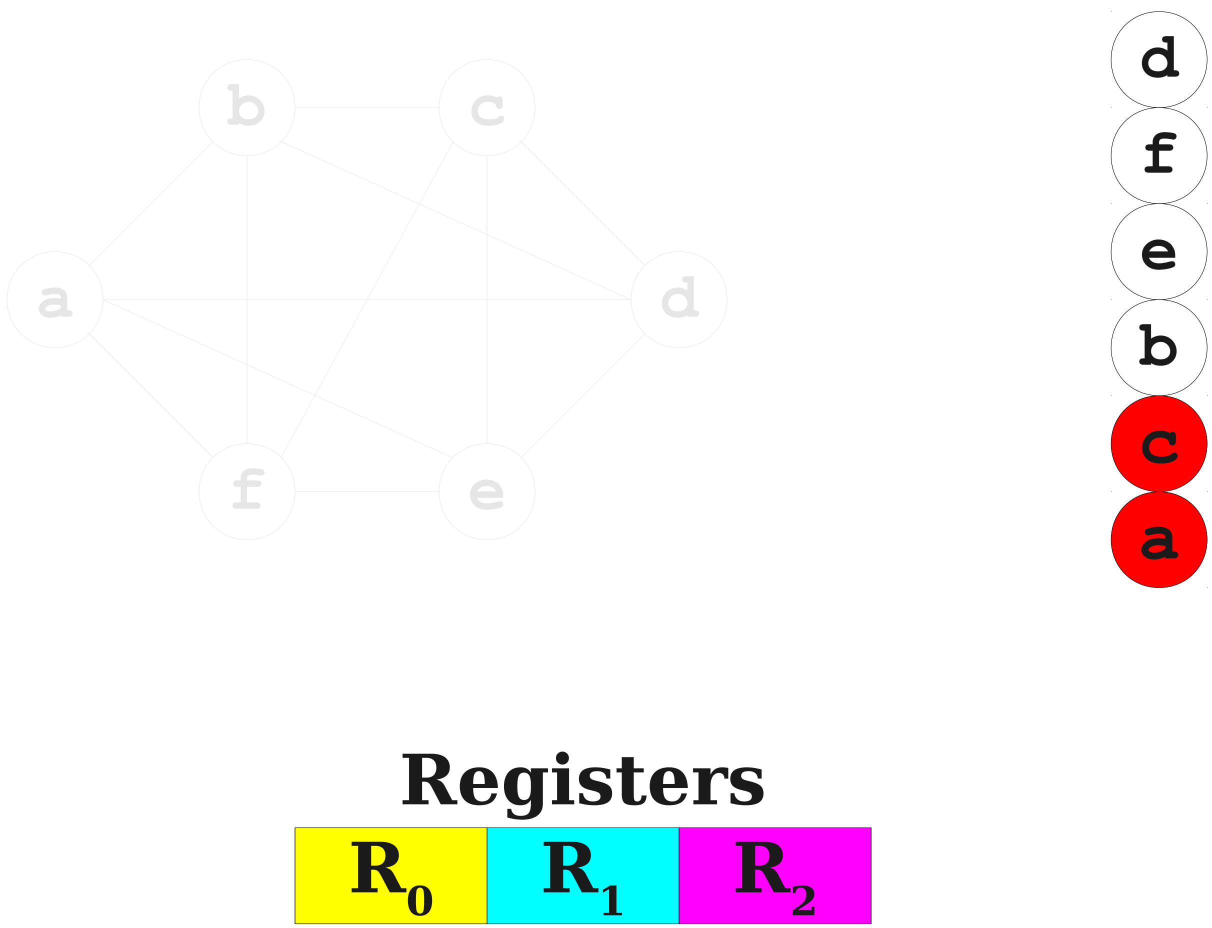
Another Example



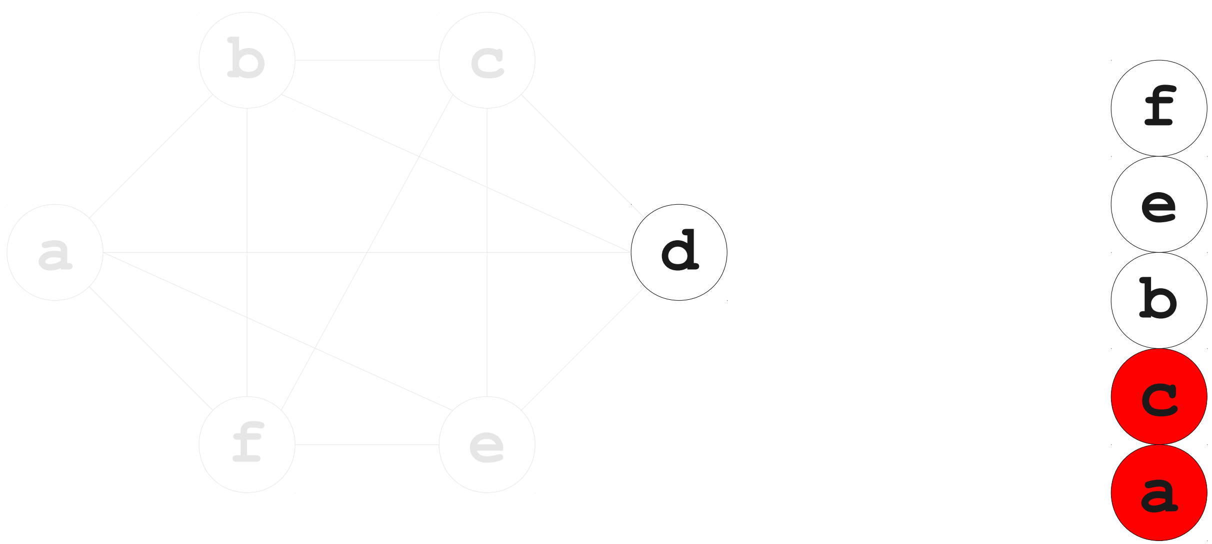
Registers

R_0	R_1	R_2

Another Example



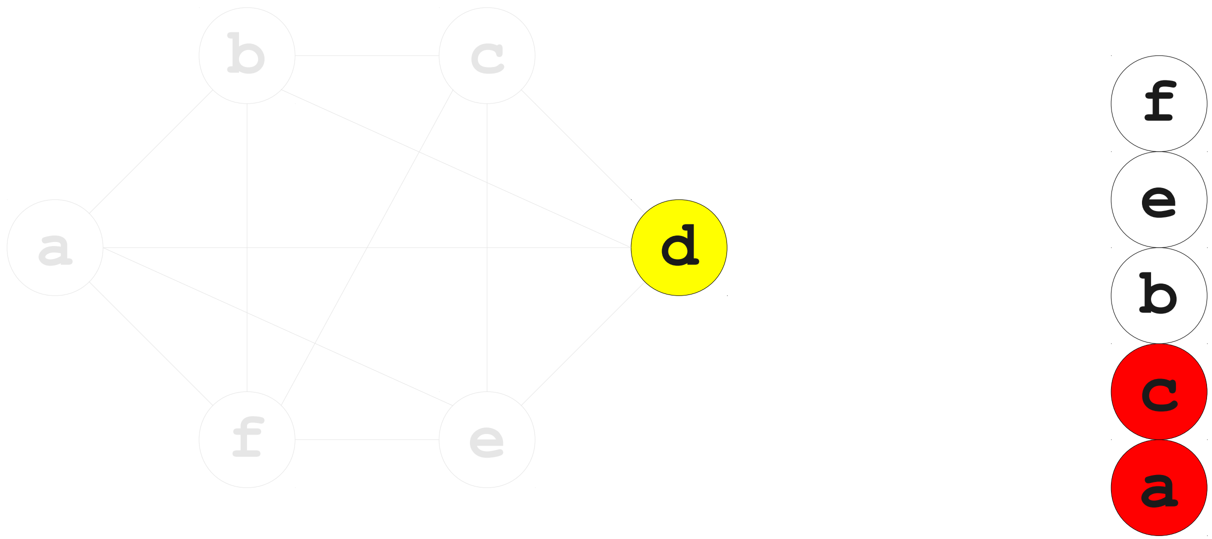
Another Example



Registers

R_0	R_1	R_2

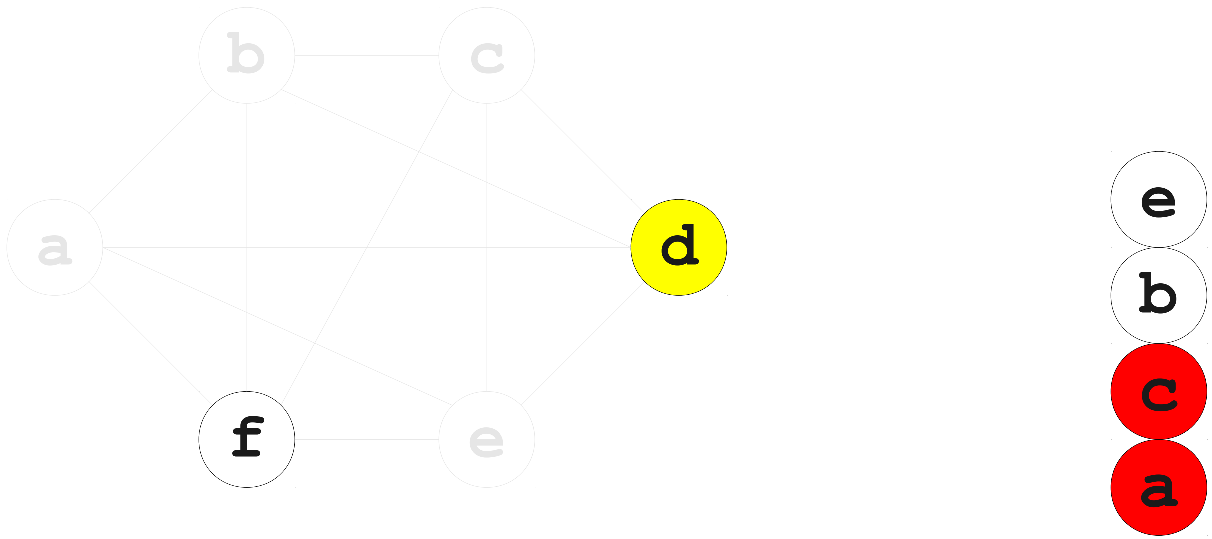
Another Example



Registers

R_0	R_1	R_2

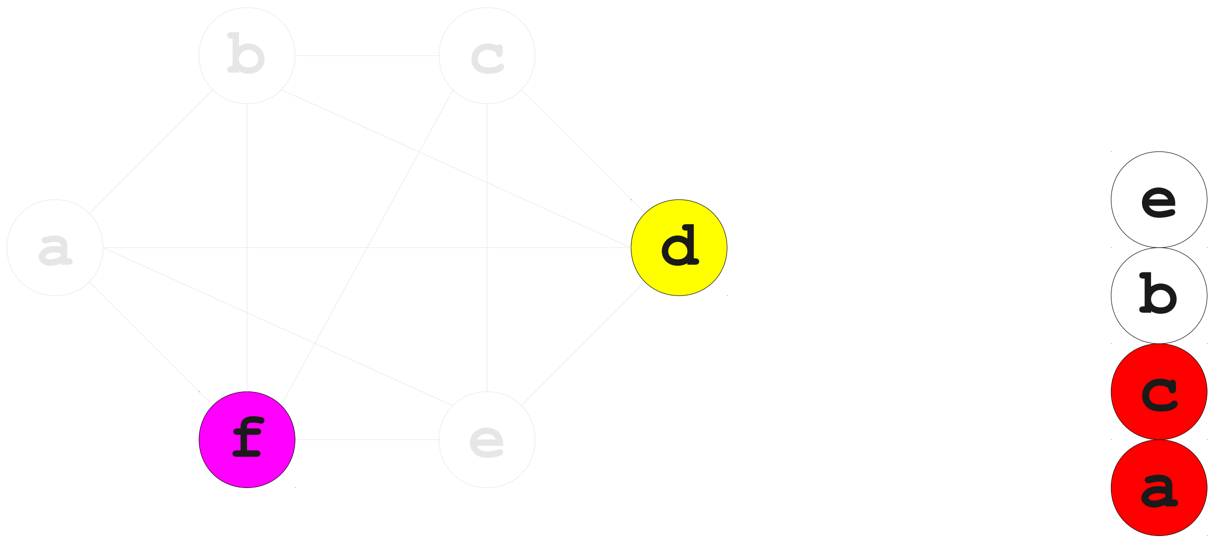
Another Example



Registers

R_0	R_1	R_2

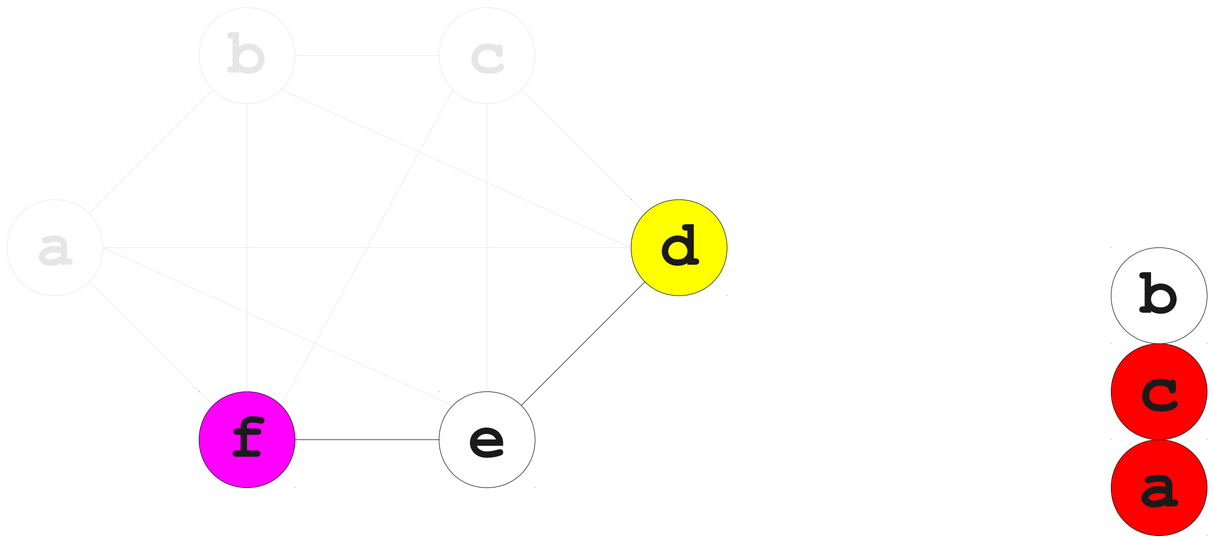
Another Example



Registers

R_0	R_1	R_2

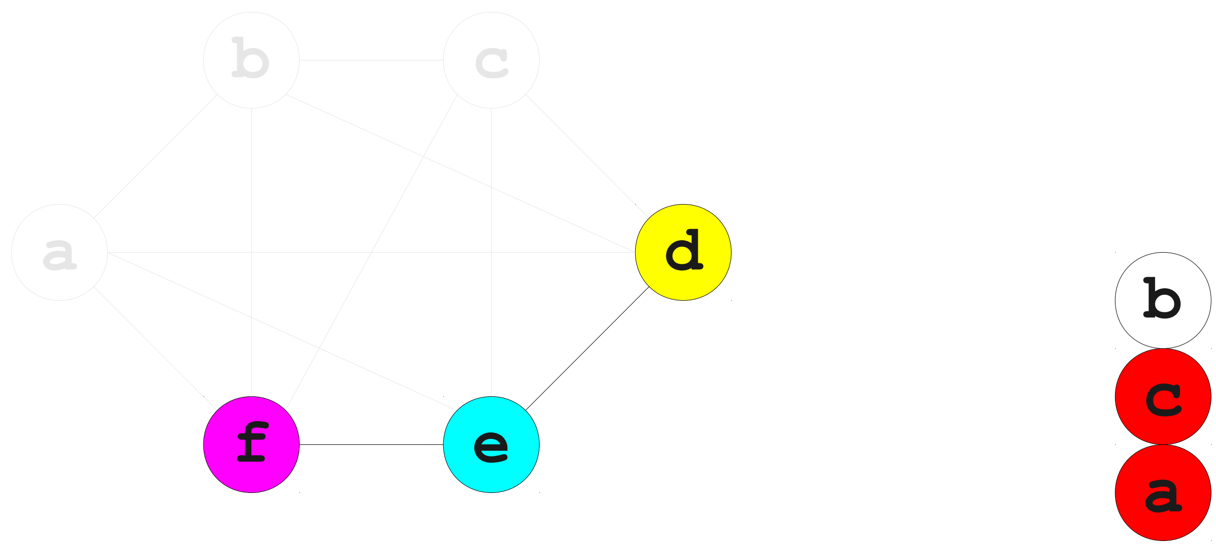
Another Example



Registers

R_0	R_1	R_2

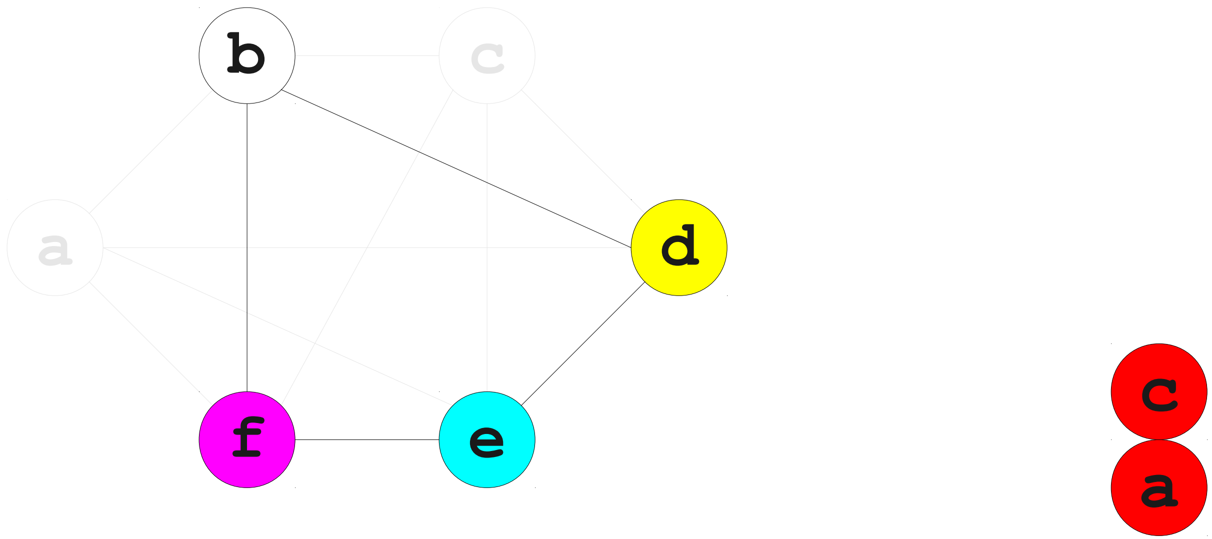
Another Example



Registers

R_0	R_1	R_2

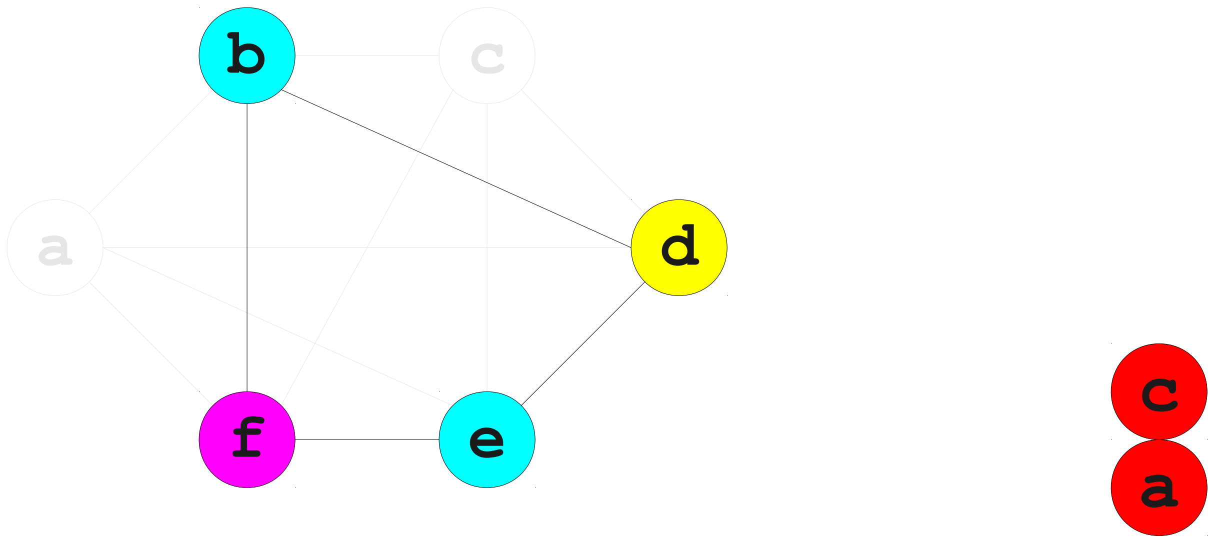
Another Example



Registers

R_0	R_1	R_2

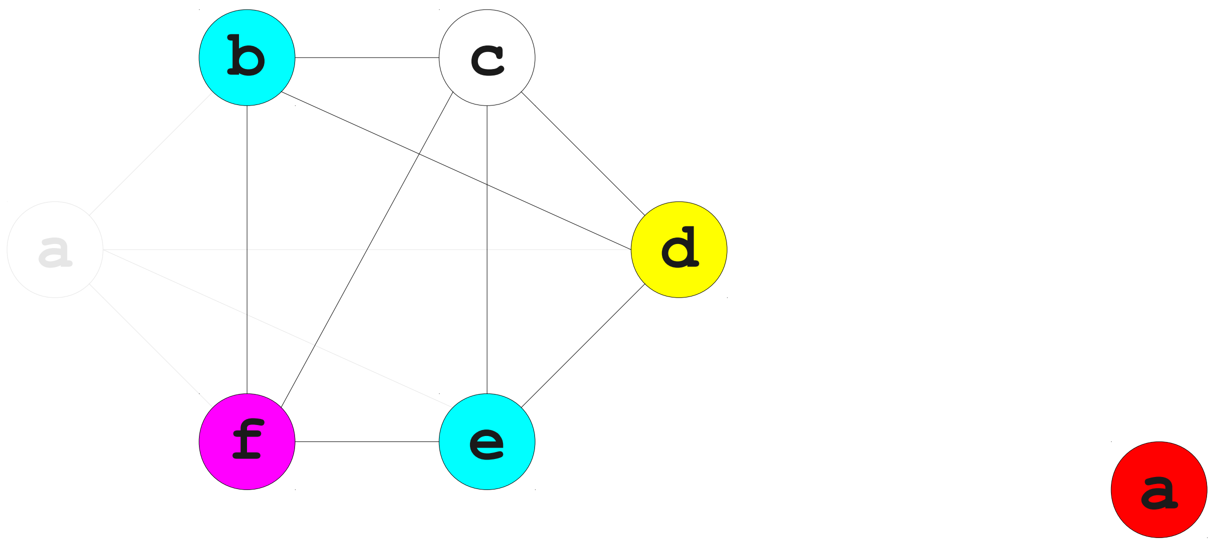
Another Example



Registers

R_0	R_1	R_2

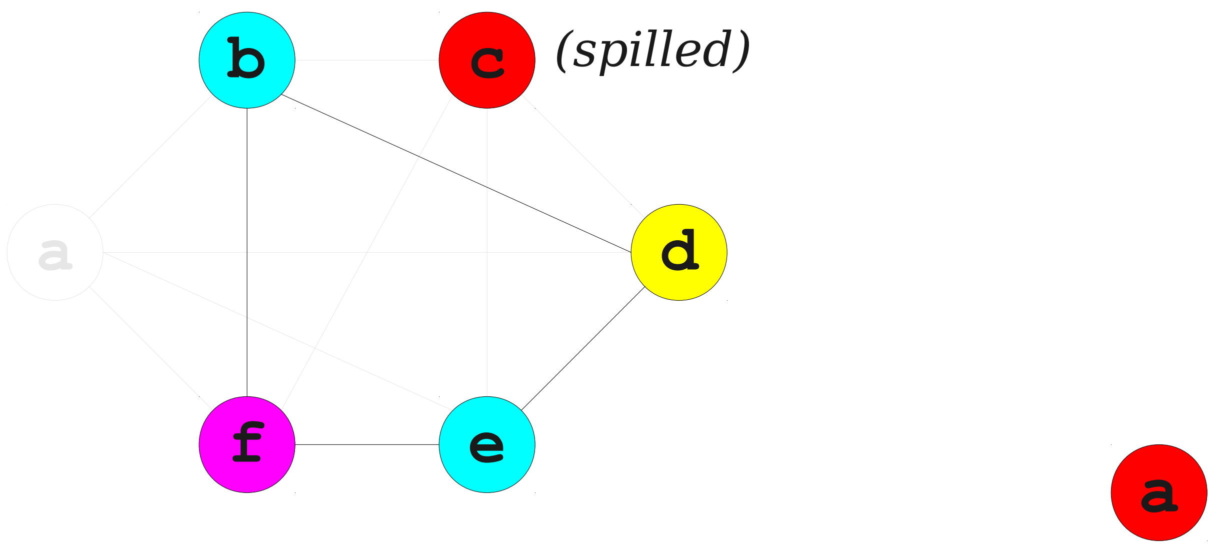
Another Example



Registers

R_0	R_1	R_2

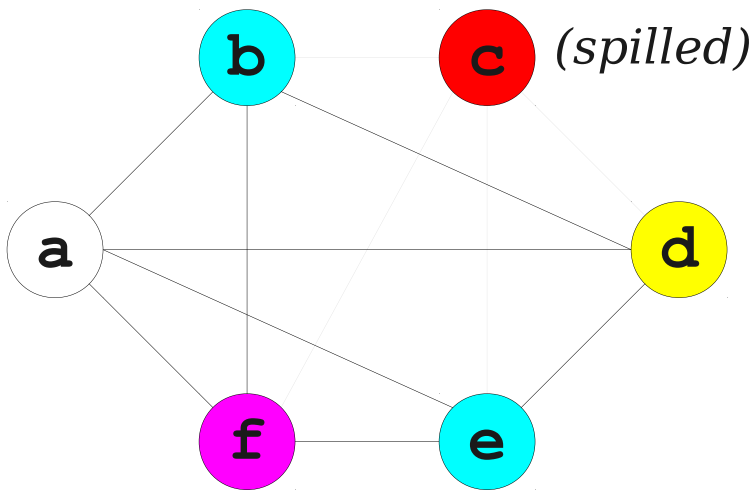
Another Example



Registers

R_0	R_1	R_2

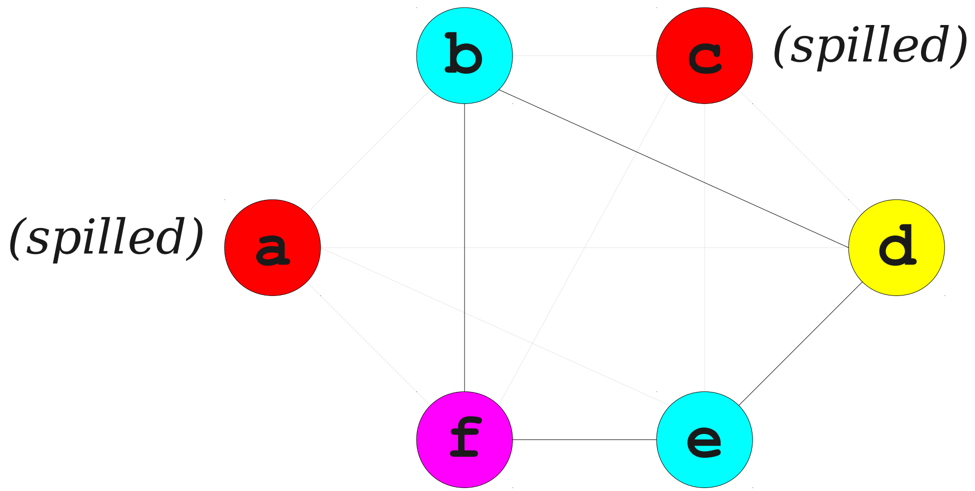
Another Example



Registers

R_0	R_1	R_2

Another Example



Registers

R_0	R_1	R_2

Chaitin's Algorithm

Chaitin's algorithm is efficient (O(|V| + |E|), simple to implement

Chaitin's Algorithm

Chaitin's algorithm is efficient (O(|V| + |E|), simple to implement

- How good the coloring is depends on the order we color the nodes to the graph
  - called the **elimination ordering**

Chaitin's Algorithm

Chaitin's algorithm is efficient ($O(|V| + |E|)$, simple to implement

- How good the coloring is depends on the order we color the nodes to the graph
  - called the **elimination ordering**
- For every graph, there is a elimination ordering such that Chaitin's algorithm produces an optimal coloring
  - therefore finding this optimal elimination ordering for a general graph is NP-complete

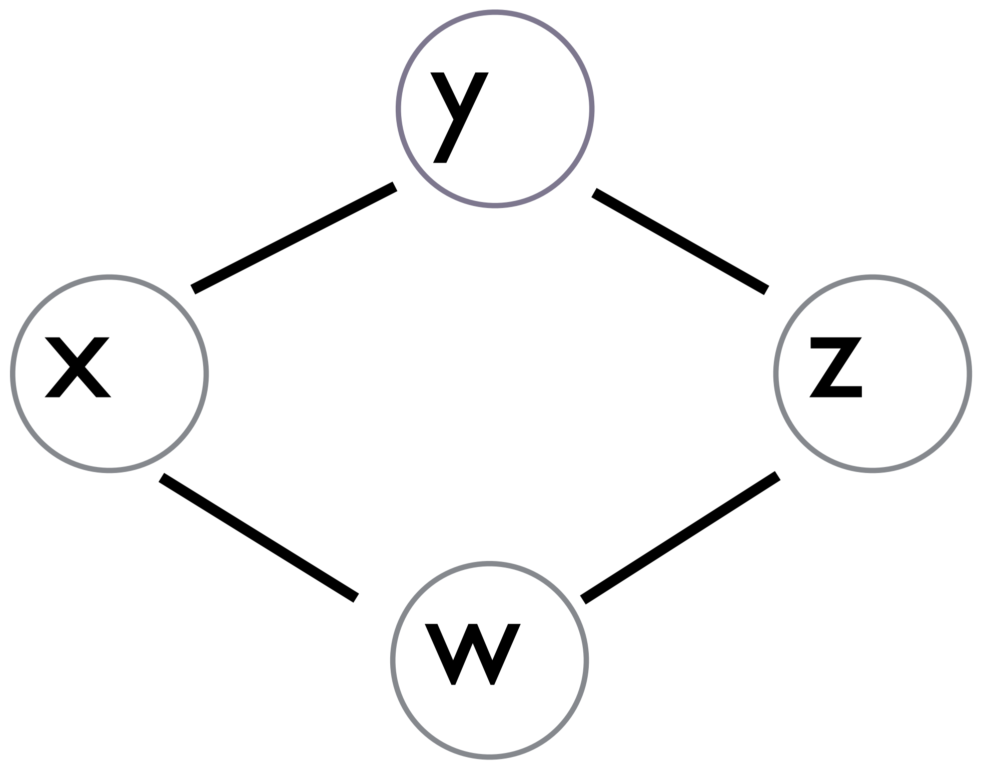
Graph Coloring SSA Programs

Hack et al, "Register Allocation for Programs in SSA-Form", *Compiler Construction* 2006

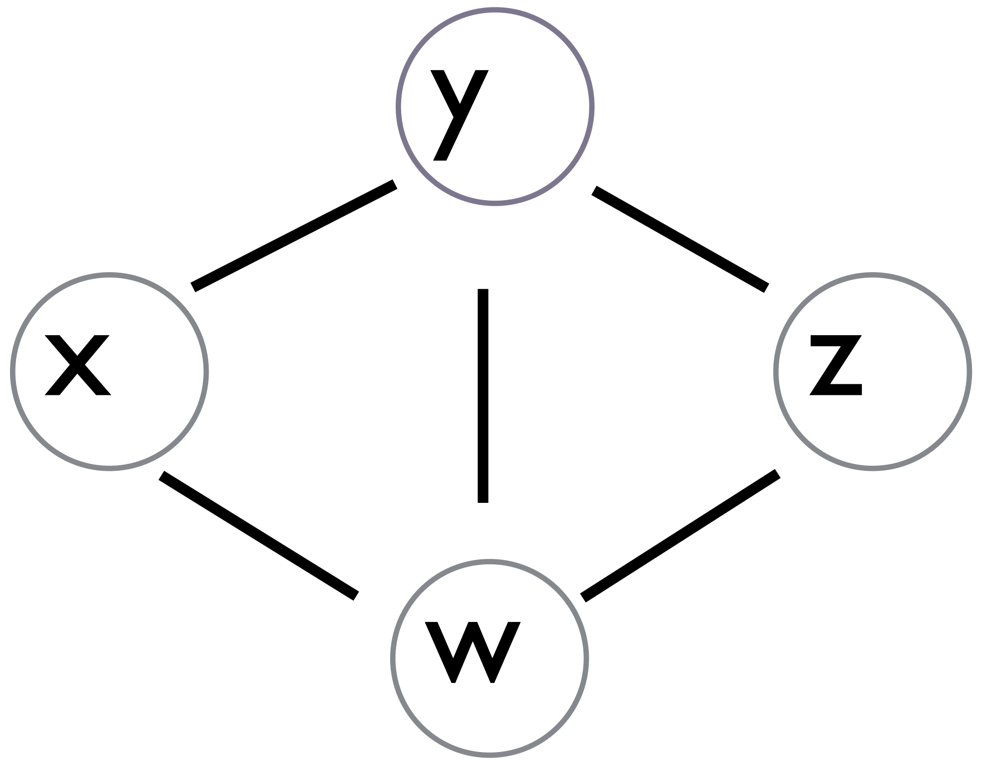
Graph Coloring SSA Programs

Hack et al, "Register Allocation for Programs in SSA-Form", *Compiler Construction* 2006

- The interference graphs of an SSA program are all **chordal**
  - Every cycle >= 4 nodes has a **chord**



Not chordal



chordal

Coloring Chordal Graphs

Theorem: Every chordal graph has a **perfect elimination ordering**

Coloring Chordal Graphs

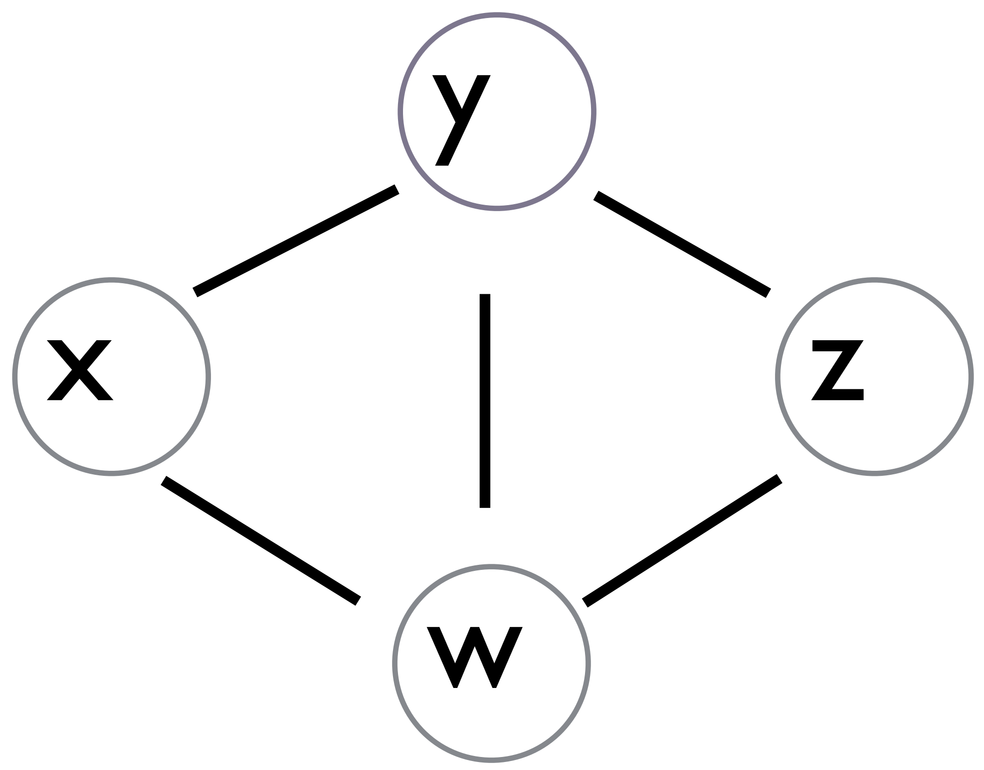
Theorem: Every chordal graph has a **perfect elimination ordering**

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors earlier in the order
- Chaitin's algo produces an optimal coloring if we use a PEO

Coloring Chordal Graphs

Theorem: A graph is chordal iff it has a **perfect elimination ordering** (PEO)

- a total ordering of nodes v1,v2,v3,... such that for each vi, vi forms a clique with all its neighbors earlier in the order
- Chaitin's algo produces an optimal coloring if we use a PEO



x,y,z,w
not perfect: N(w) non-clique

w,x,y,z
perfect

Coloring Chordal Graphs

Theorem: A graph is chordal iff it has a **perfect elimination ordering (**PEO)

Theorem: A graph is chordal iff it is the intersection graph of a group of subtrees of a tree

- In an SSA program, each variable's liveness is a subtree of the AST
- The interference graph is exactly the intersection graph of those subtrees
- Therefore, the interference graph of an SSA program is chordal!

SSA Interference Graphs are Chordal!

Every SSA Interference Graph is chordal

Chaitin's algorithm computes optimal coloring given a PEO

So we can color SSA interference graphs if we can find a PEO

SSA programs have a perfect elimination ordering that is easy to compute:

> "in-scope" or "dominance" relation
>
> a variable x dominates y if x in scope when y is defined (includes simultaneous binding)

- x's definition is "closer to the root" of the AST than y
- easy to compute: pre-order traversal of the nodes

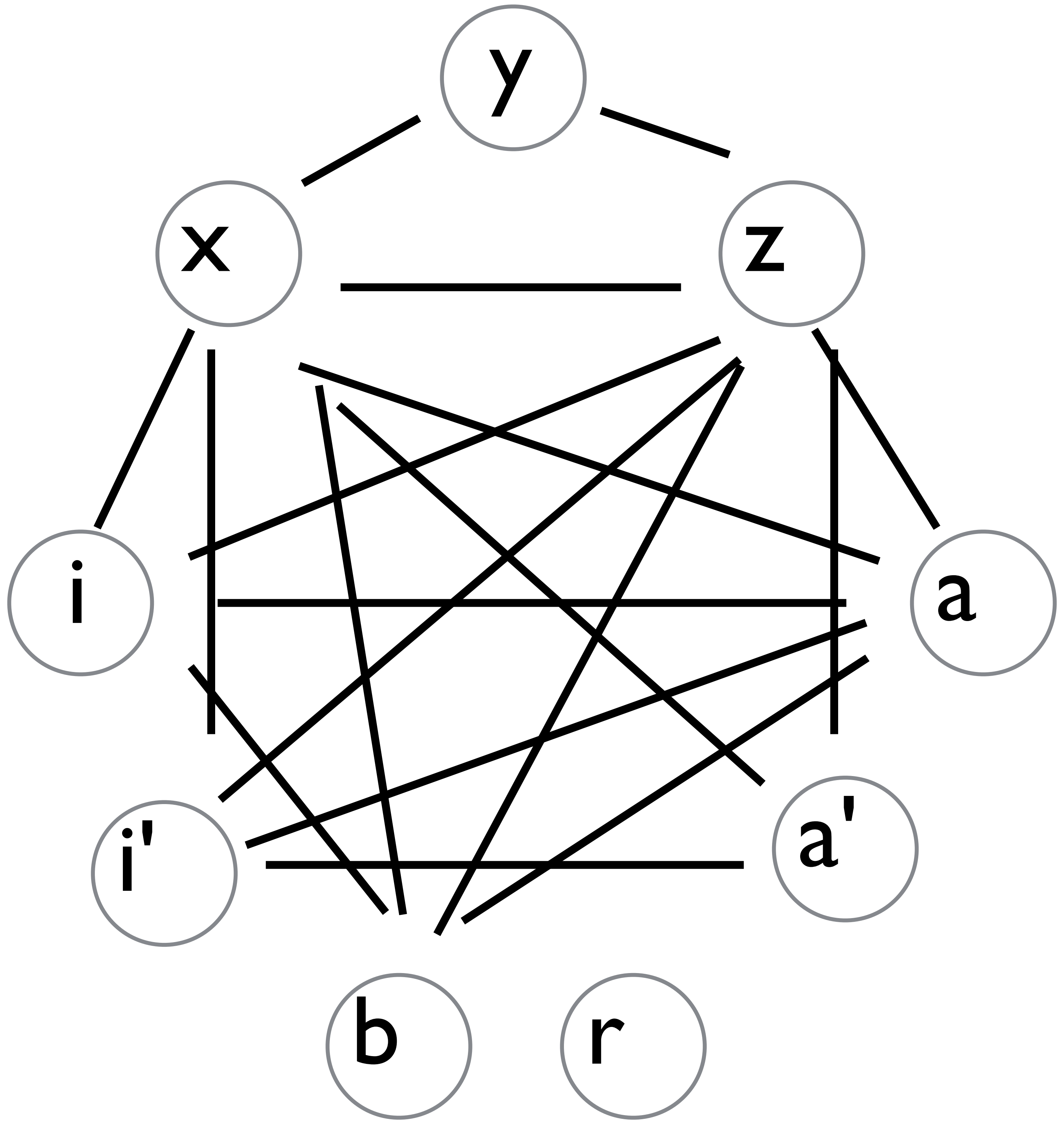
SSA Program

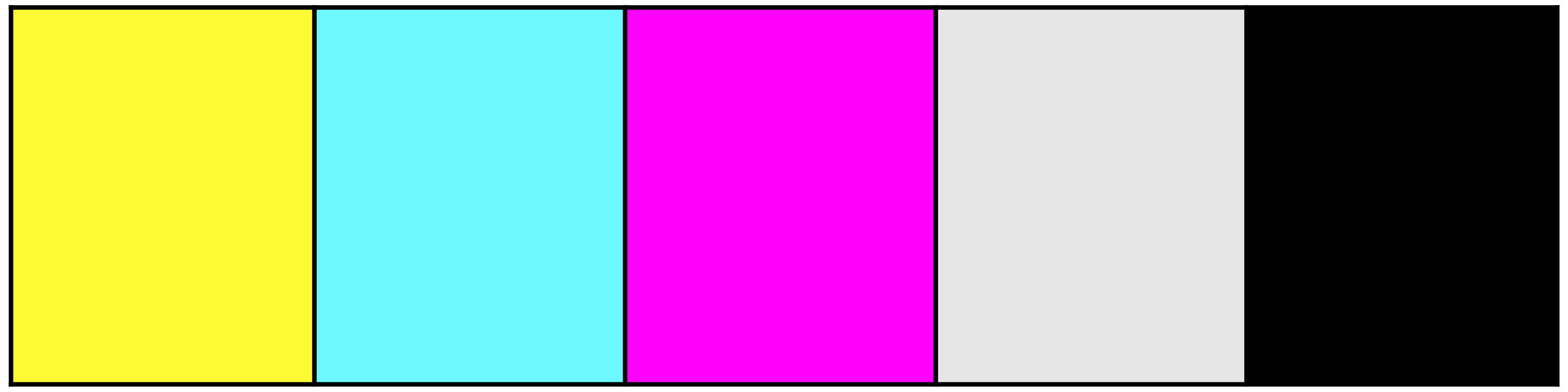
```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Pick a PEO:

variables should be colored after anything that was already in scope

Interference Graph

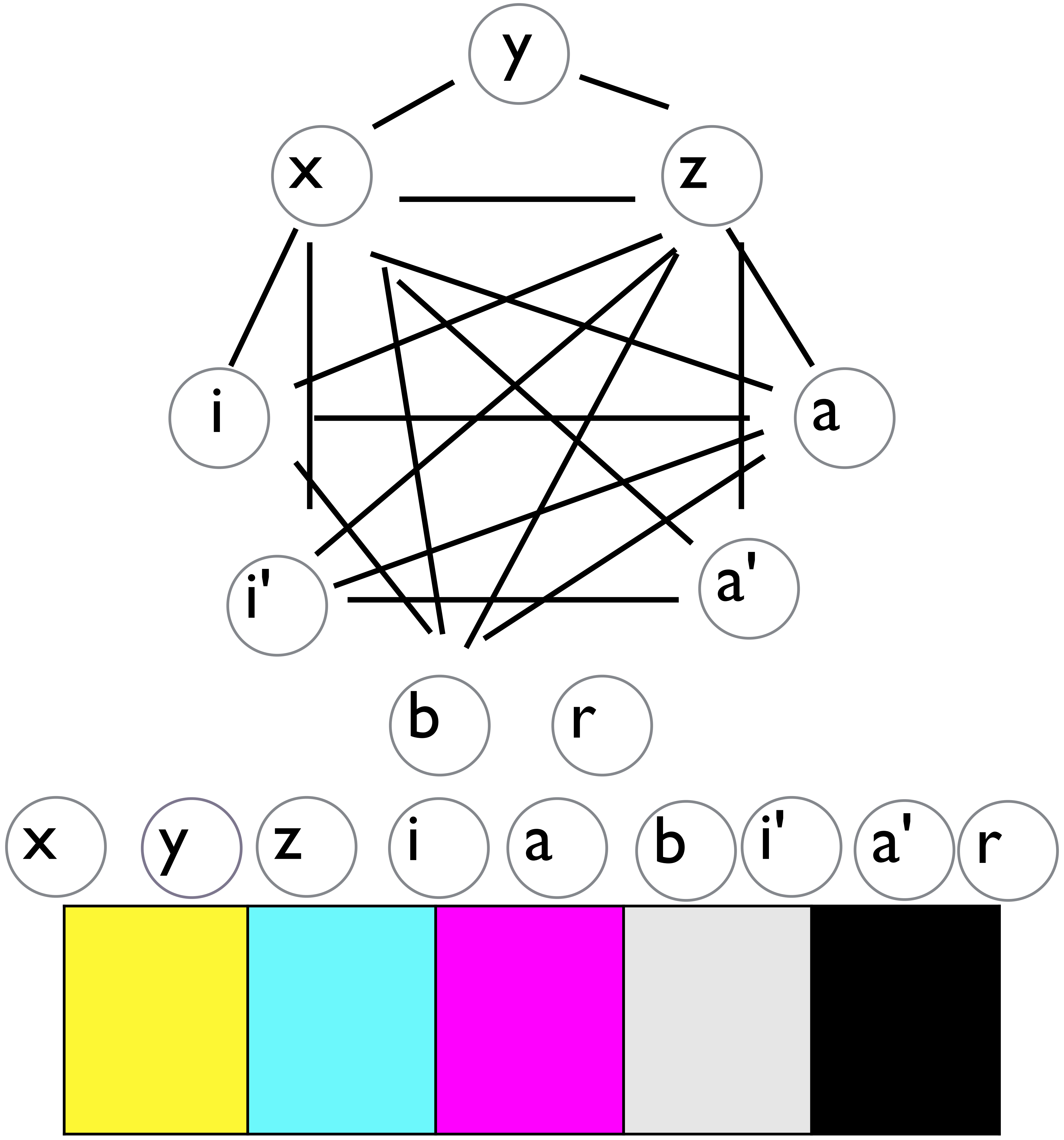




SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

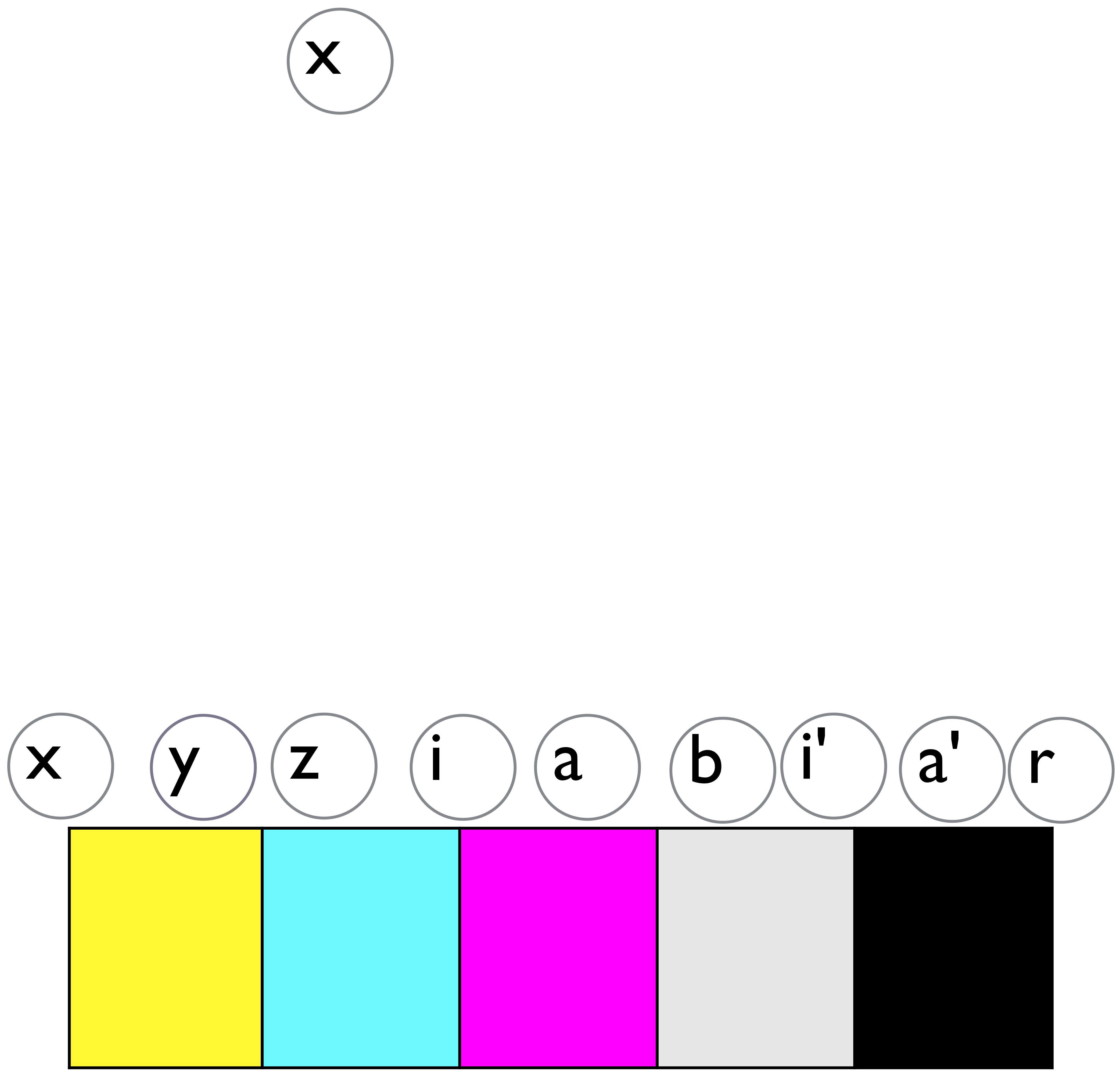
Interference Graph



SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

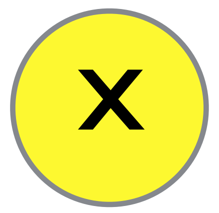
Interference Graph

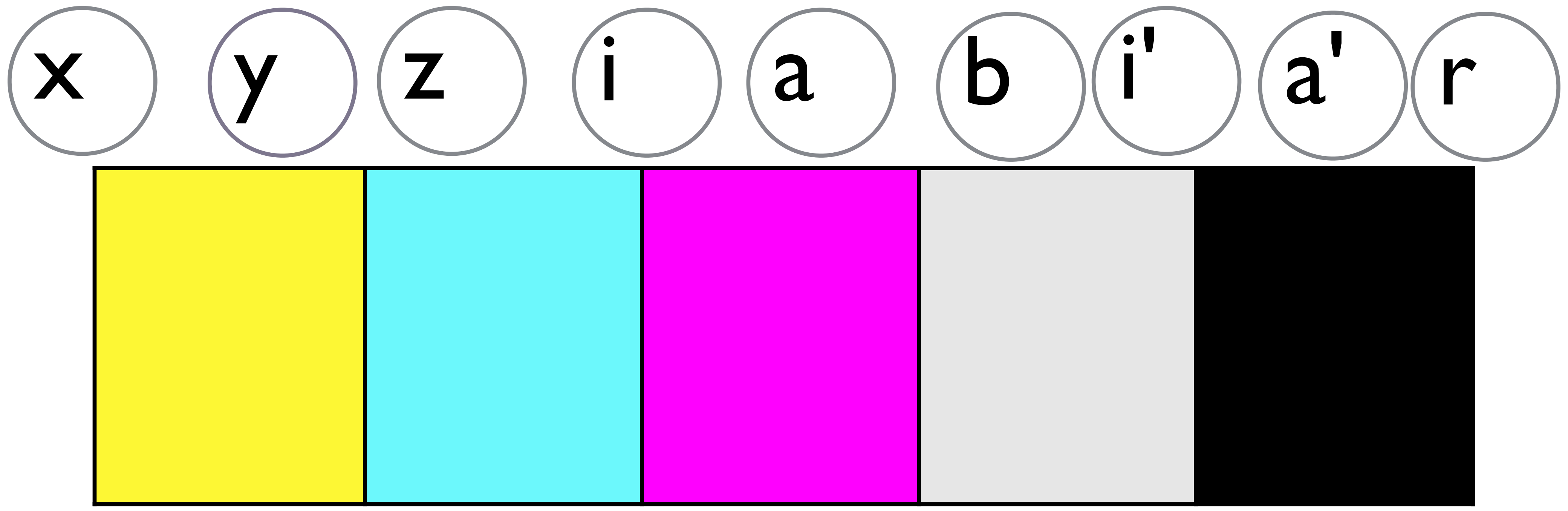


SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Interference Graph

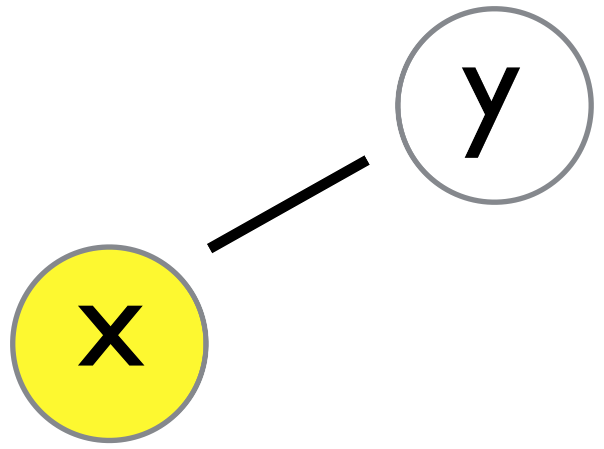


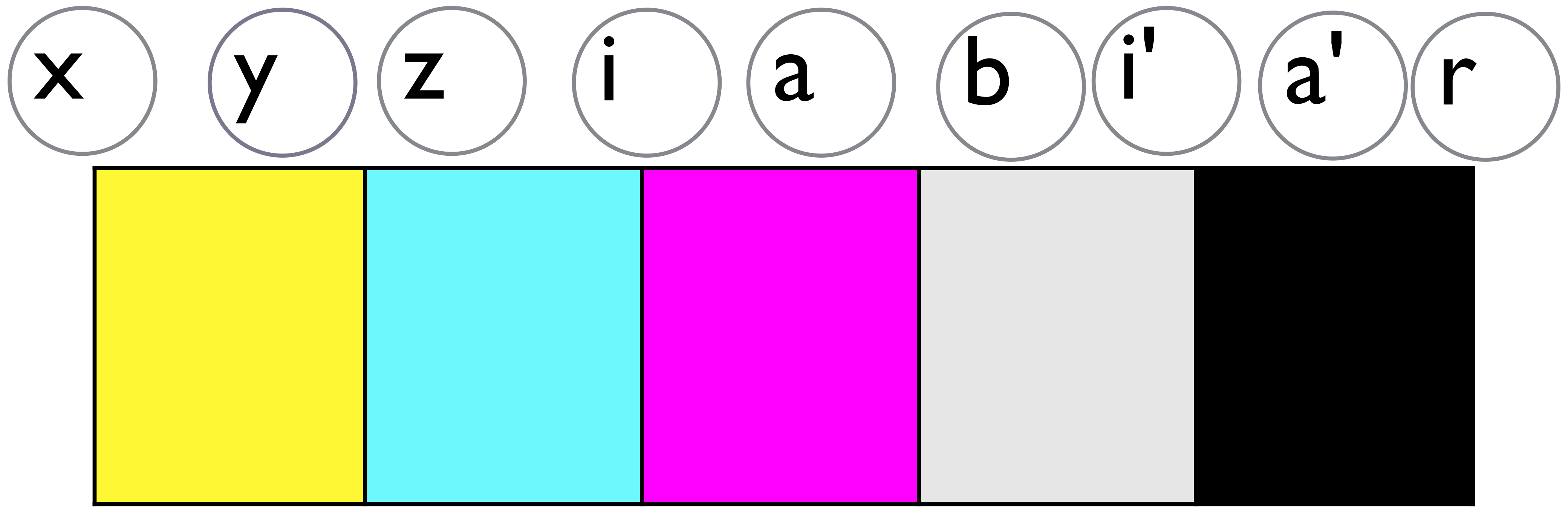


SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Interference Graph

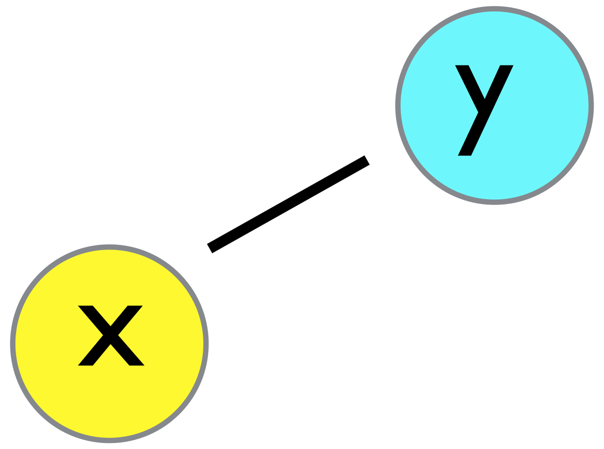


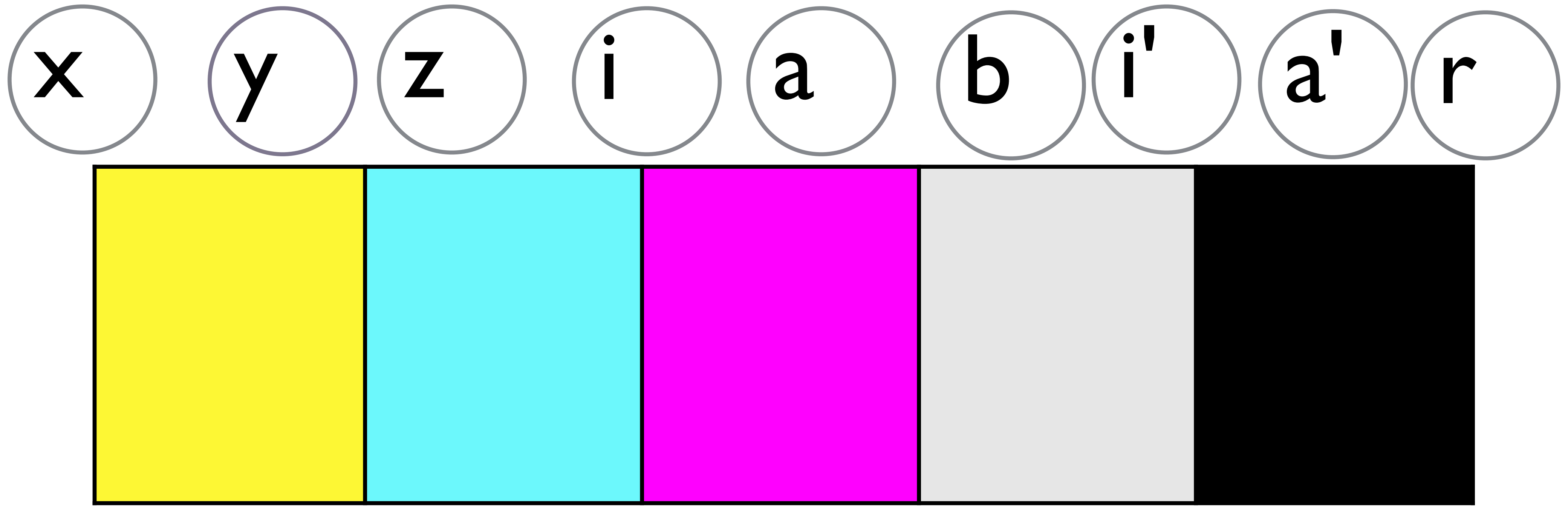


SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Interference Graph

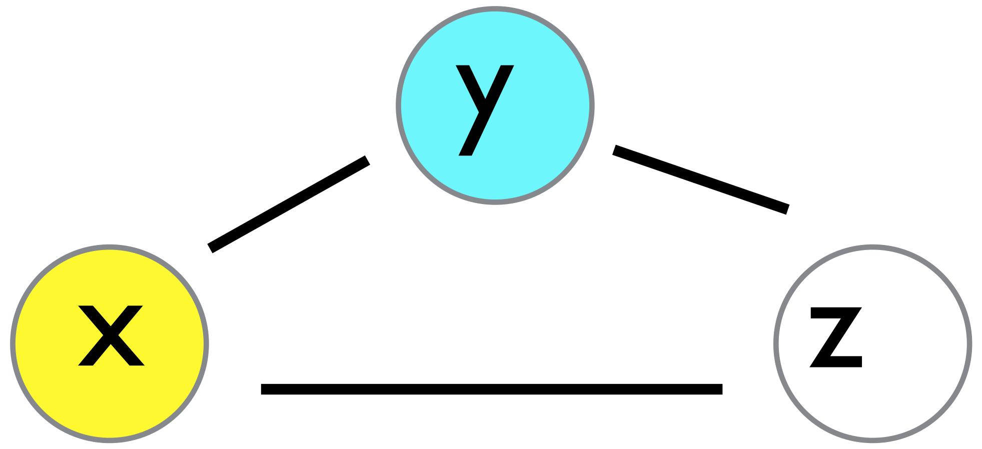


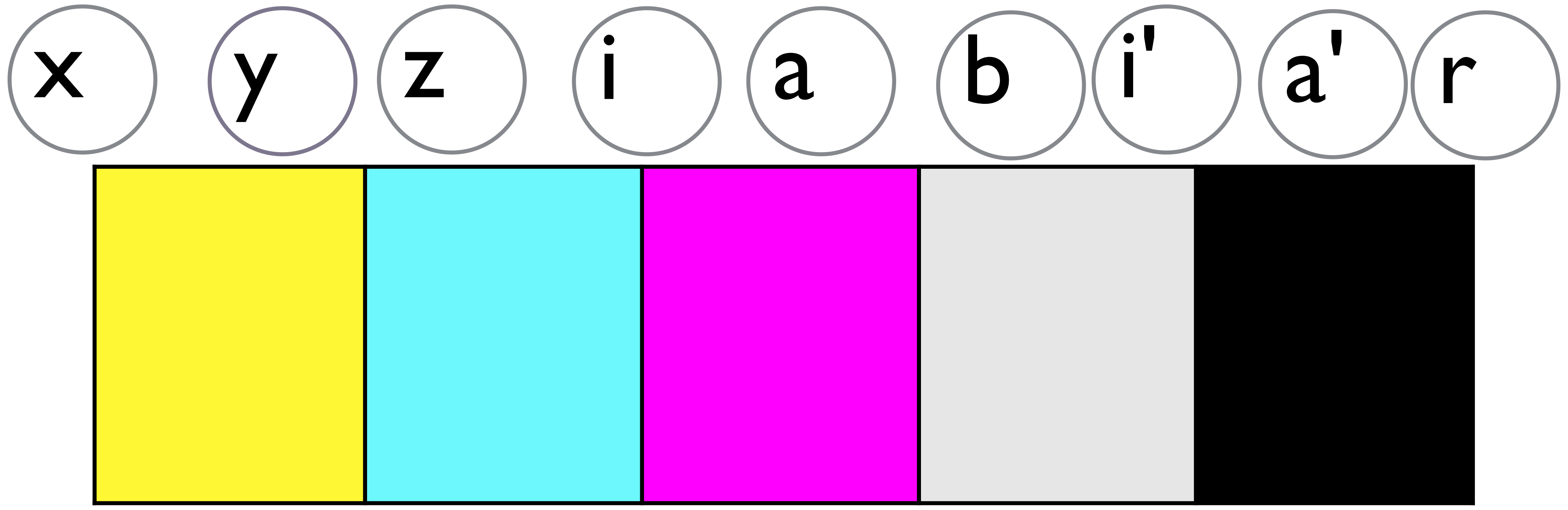


SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Interference Graph

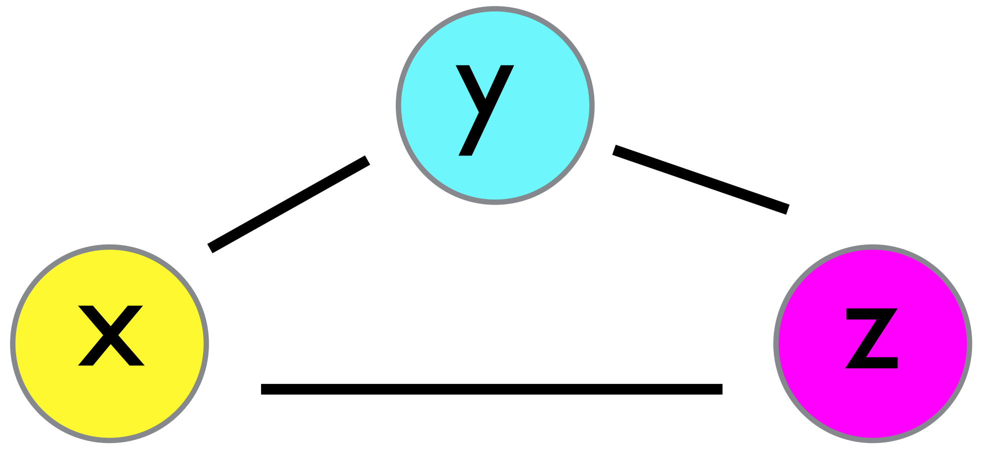


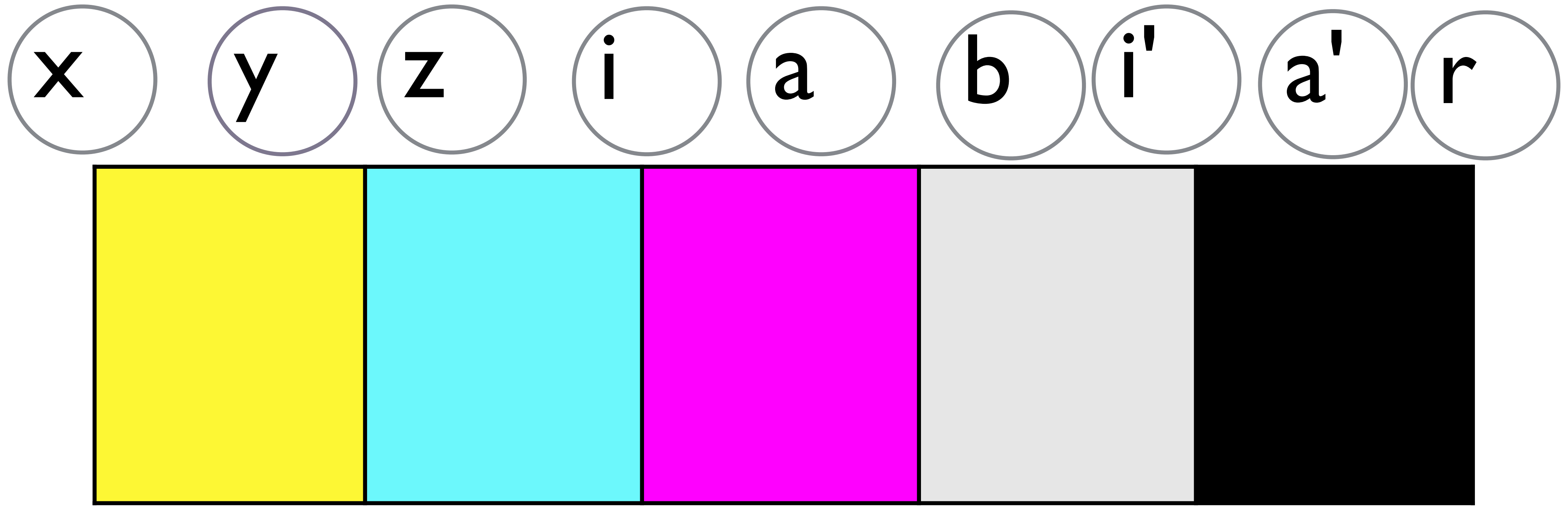


SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Interference Graph

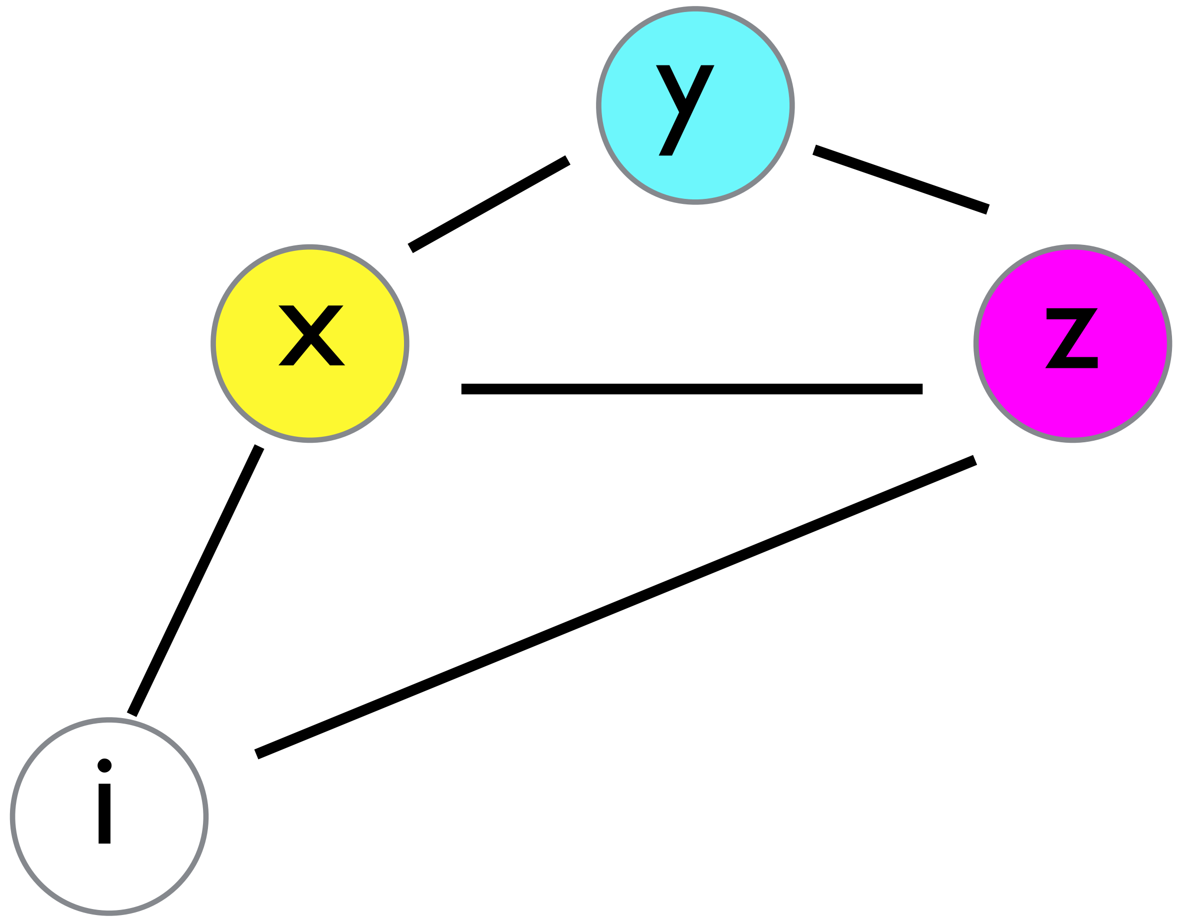


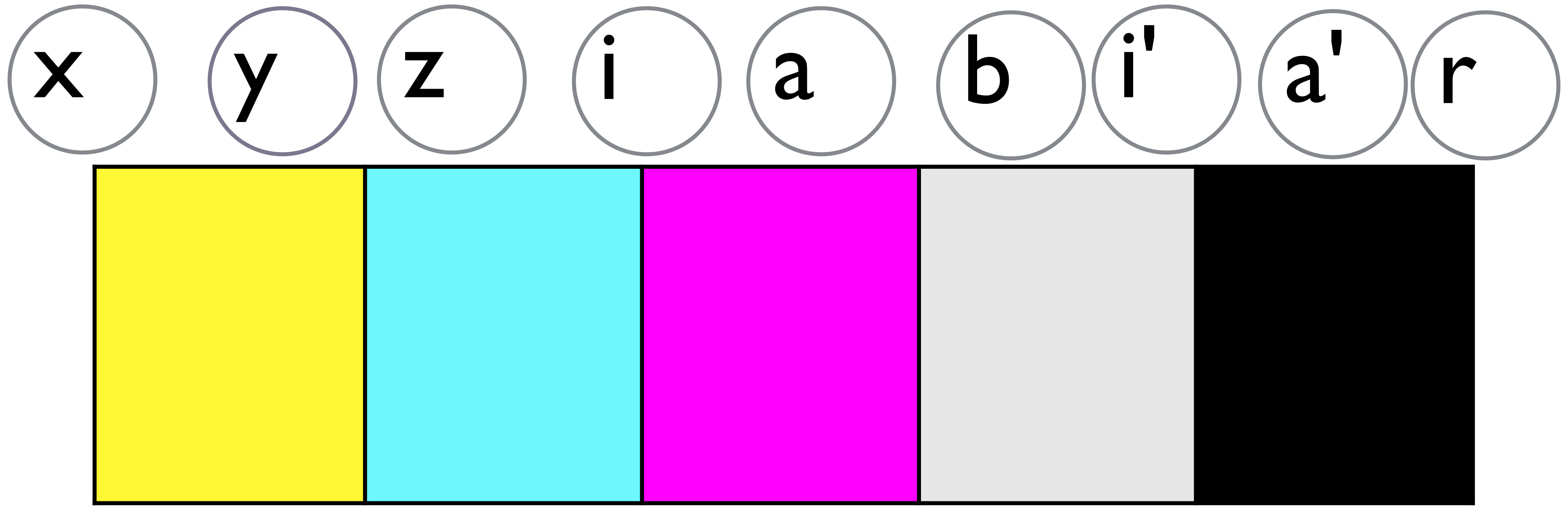


SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Interference Graph

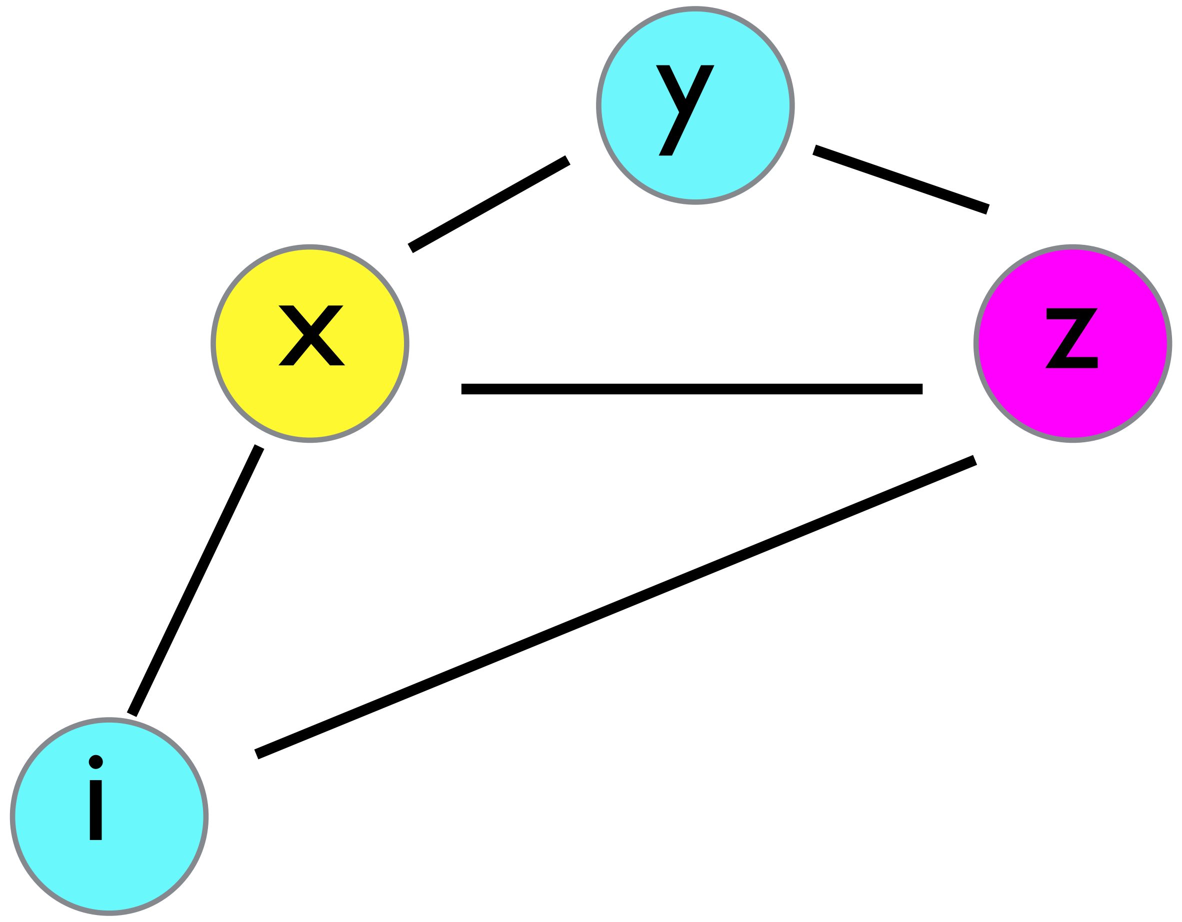


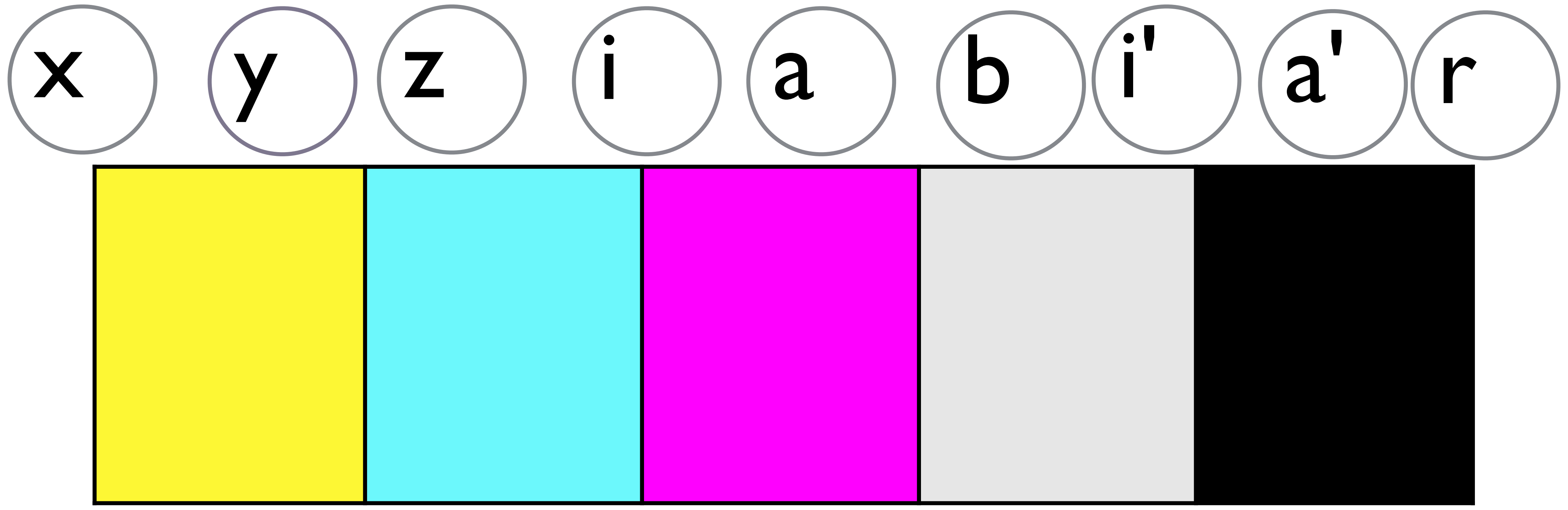


SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Interference Graph

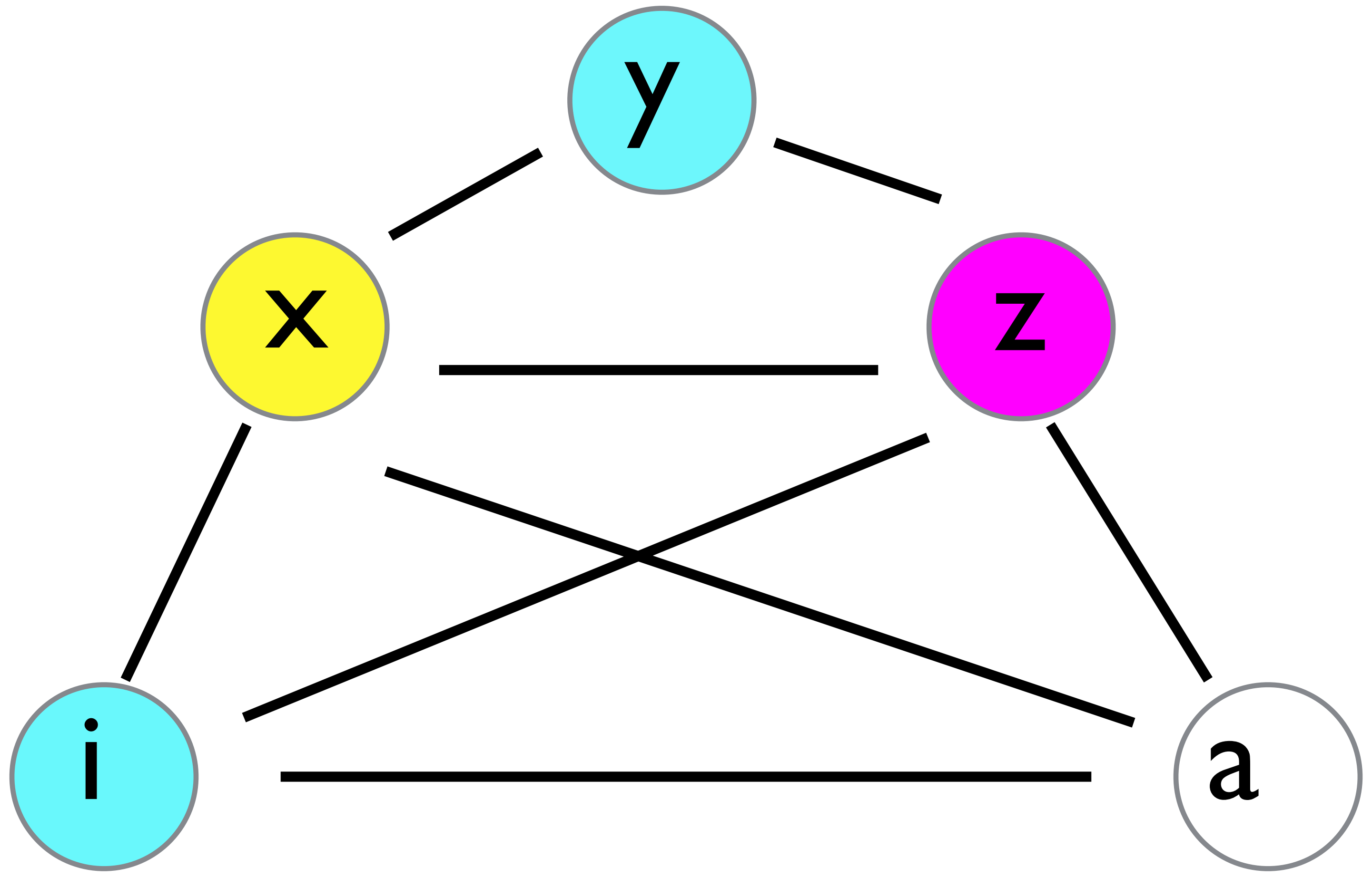


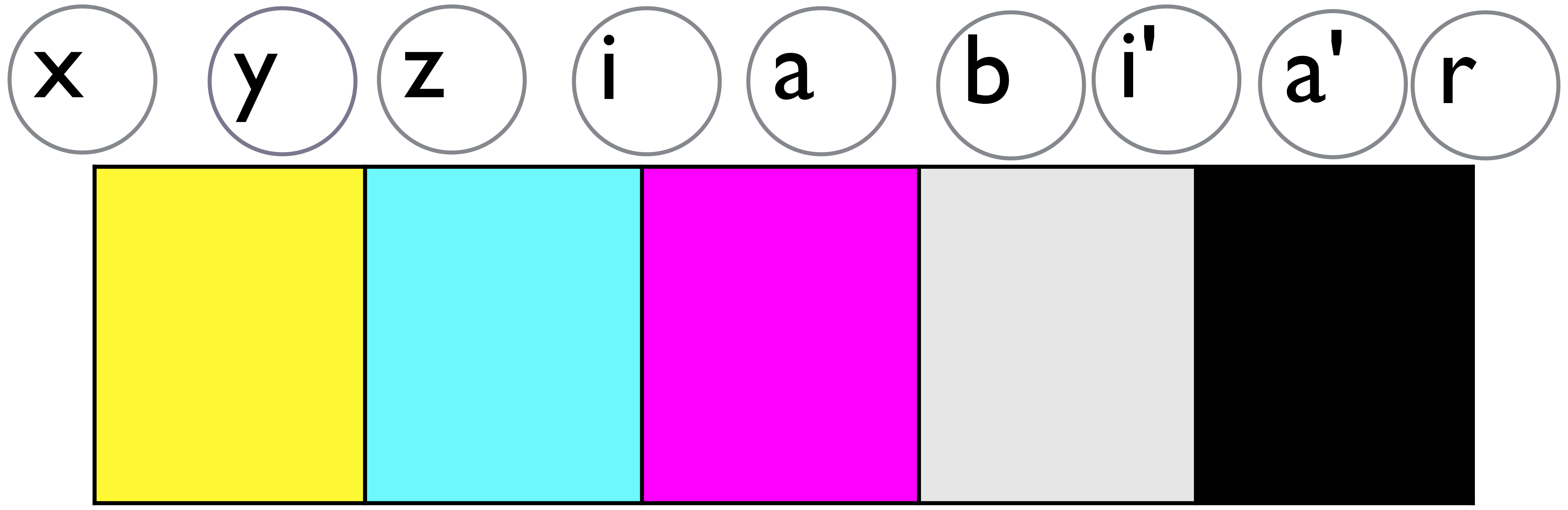


SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Interference Graph

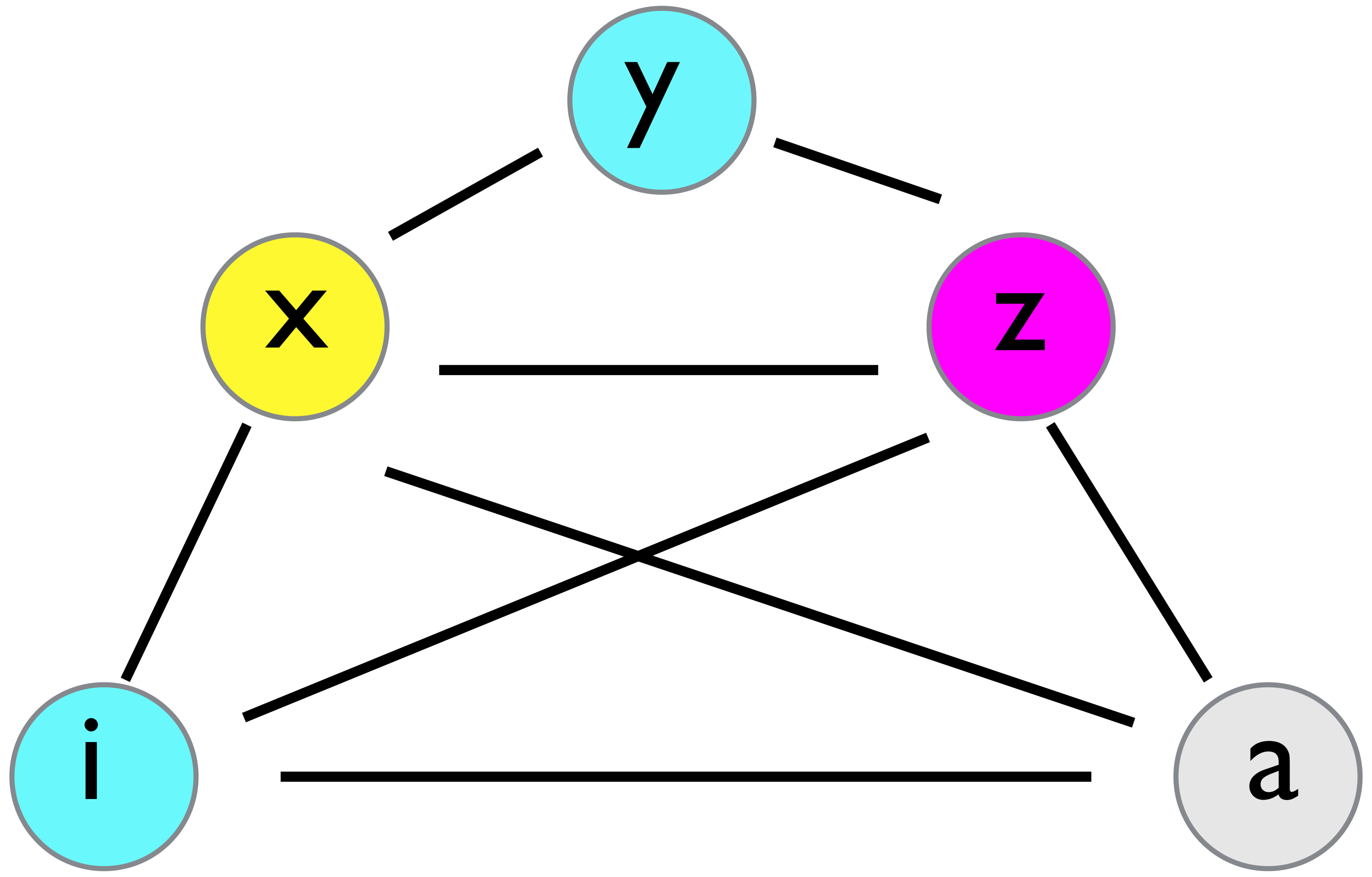


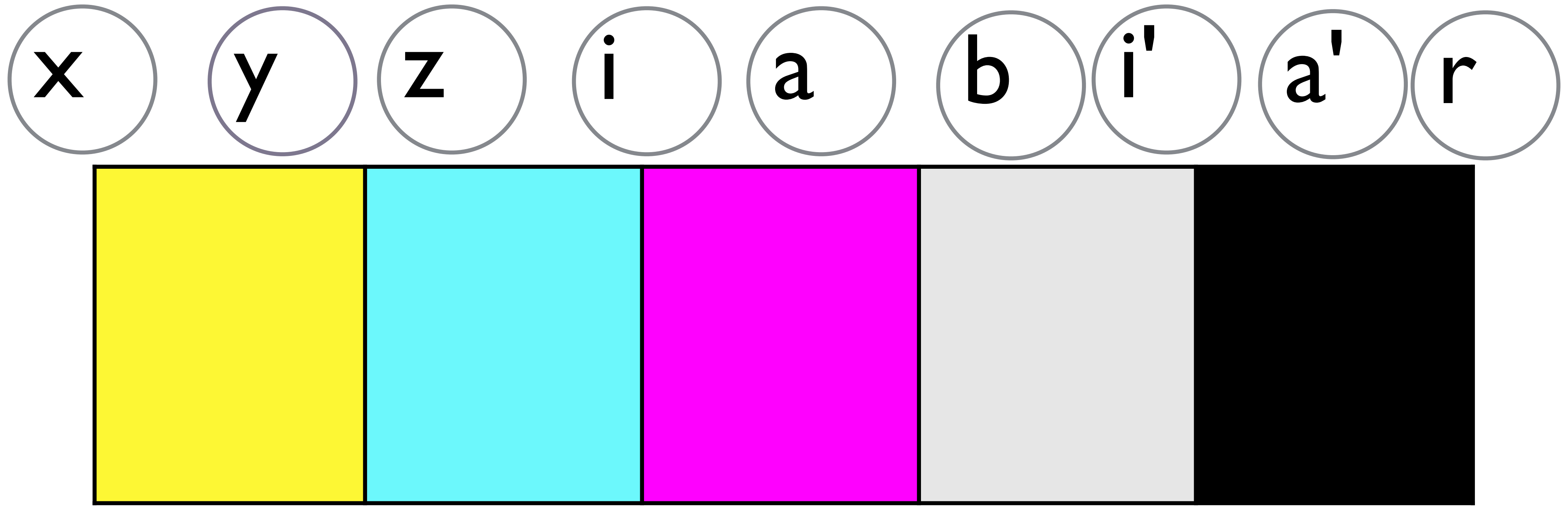


SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Interference Graph

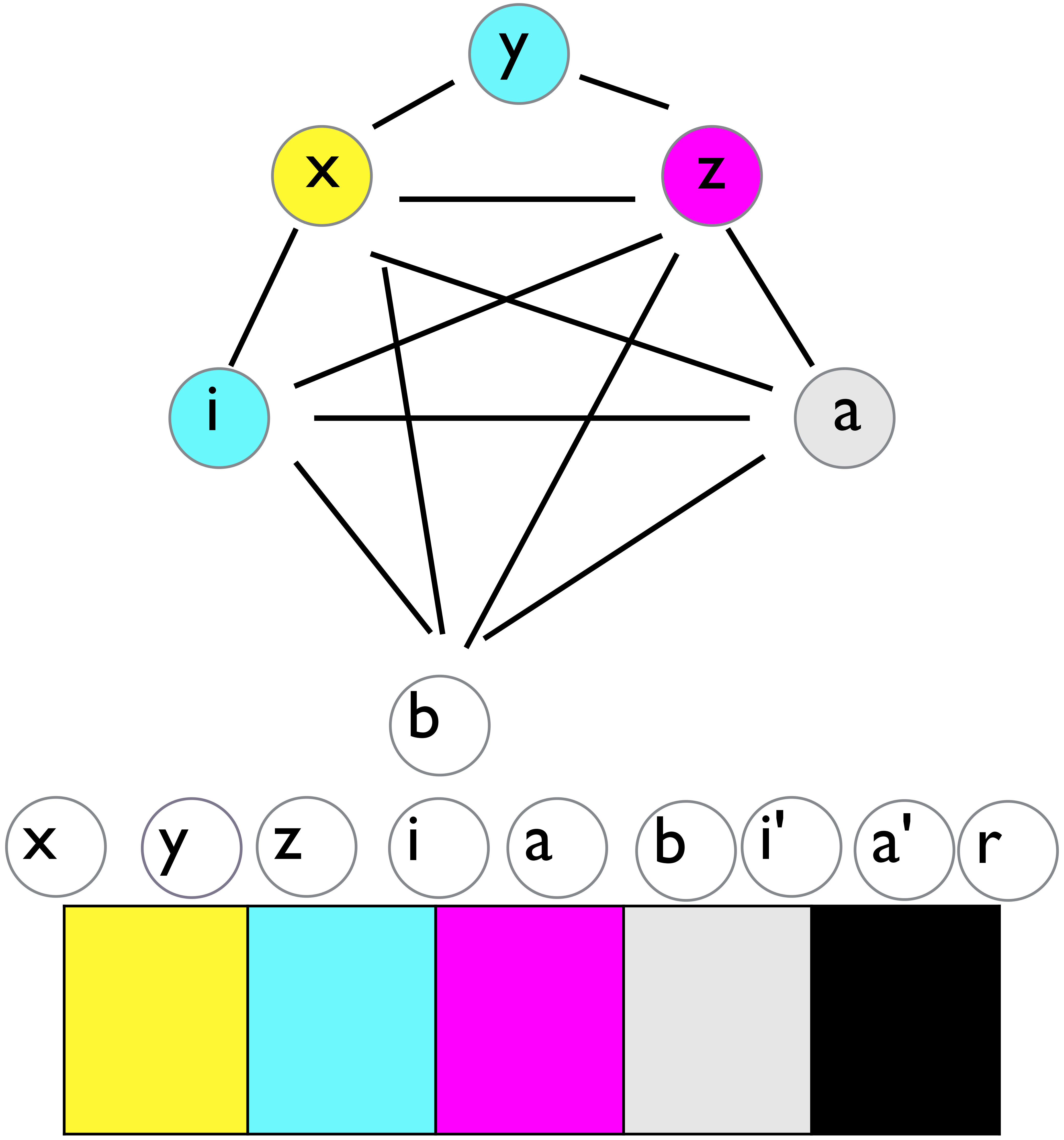




SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

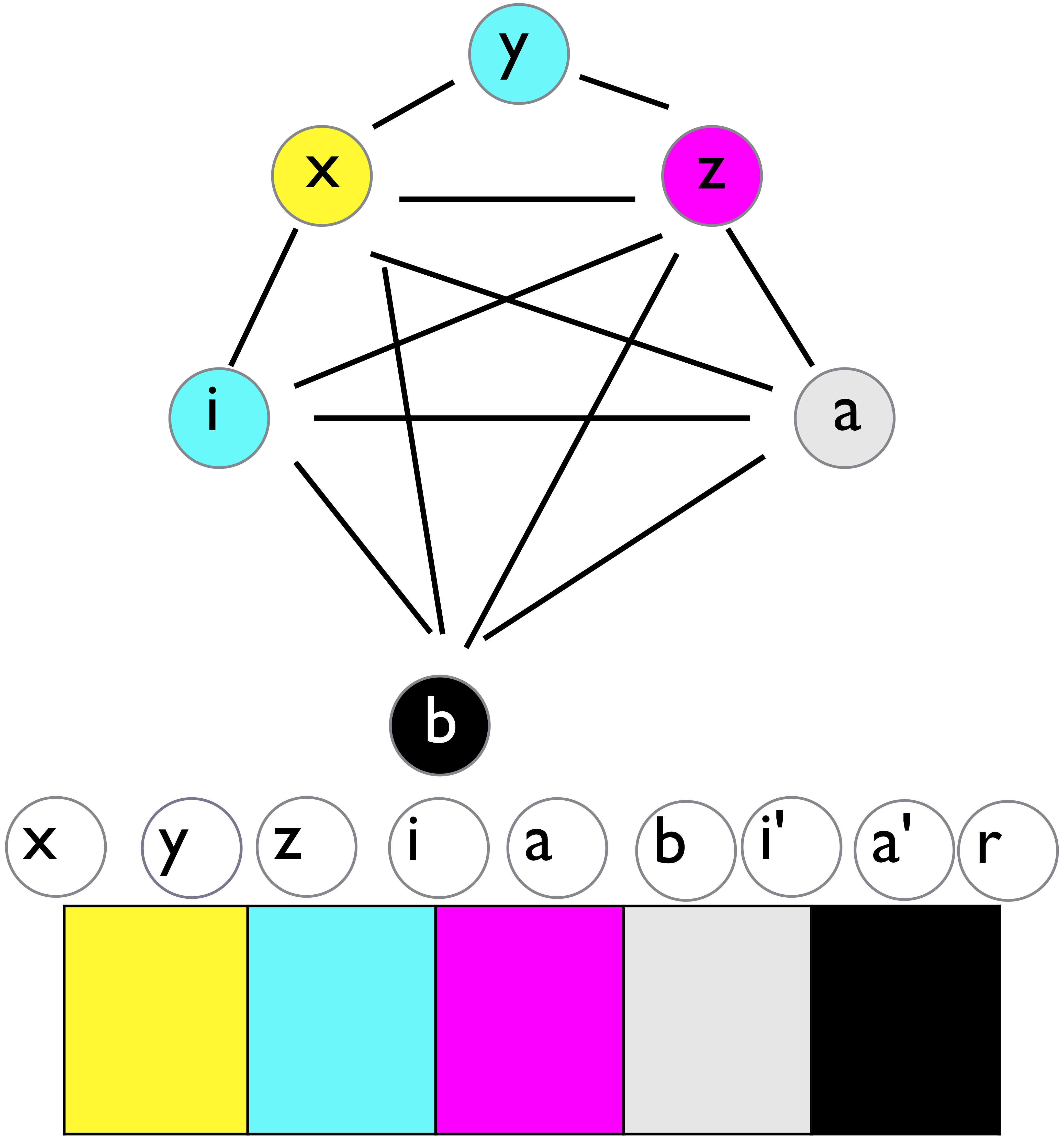
Interference Graph



SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

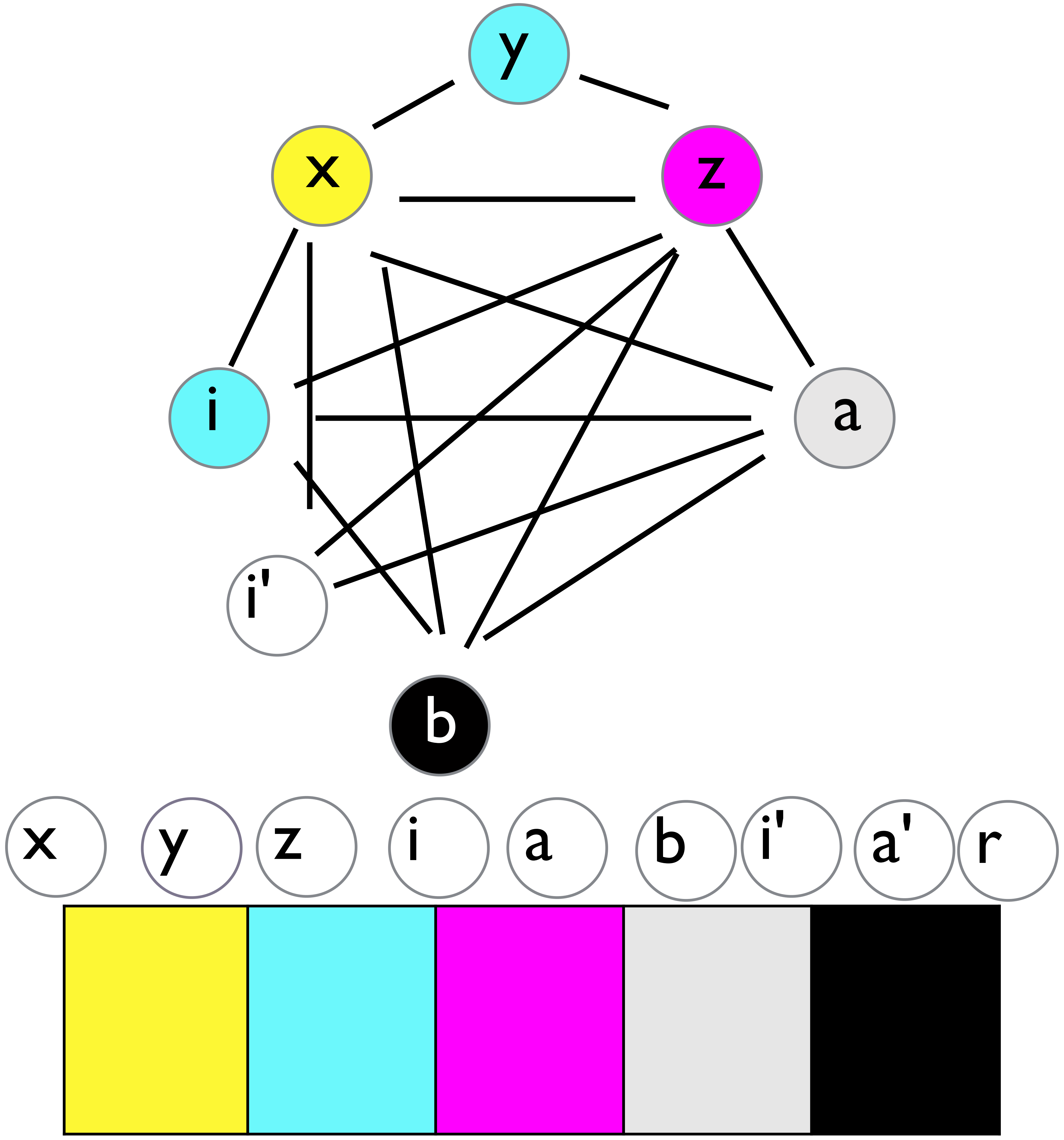
Interference Graph



SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

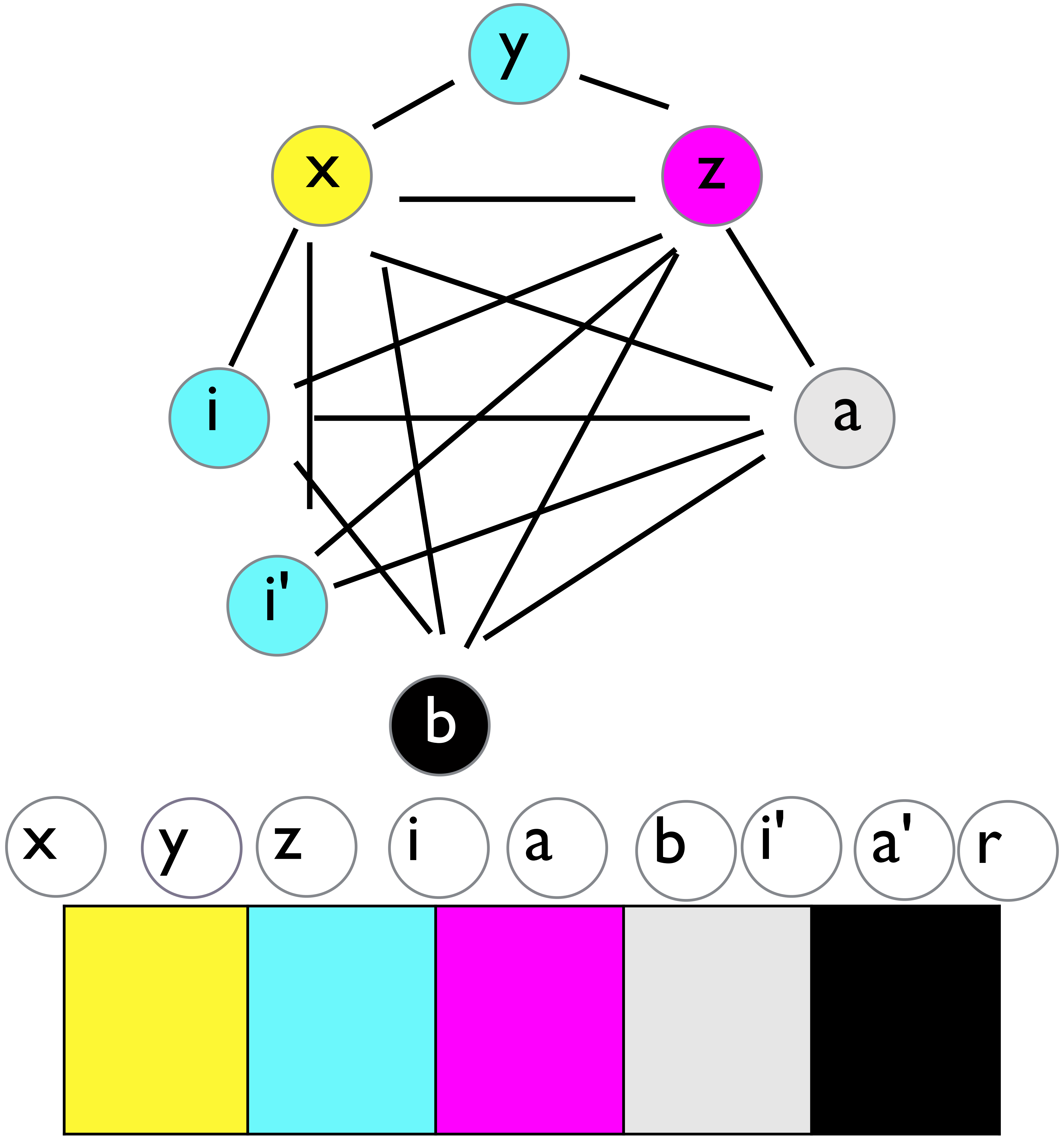
Interference Graph



SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

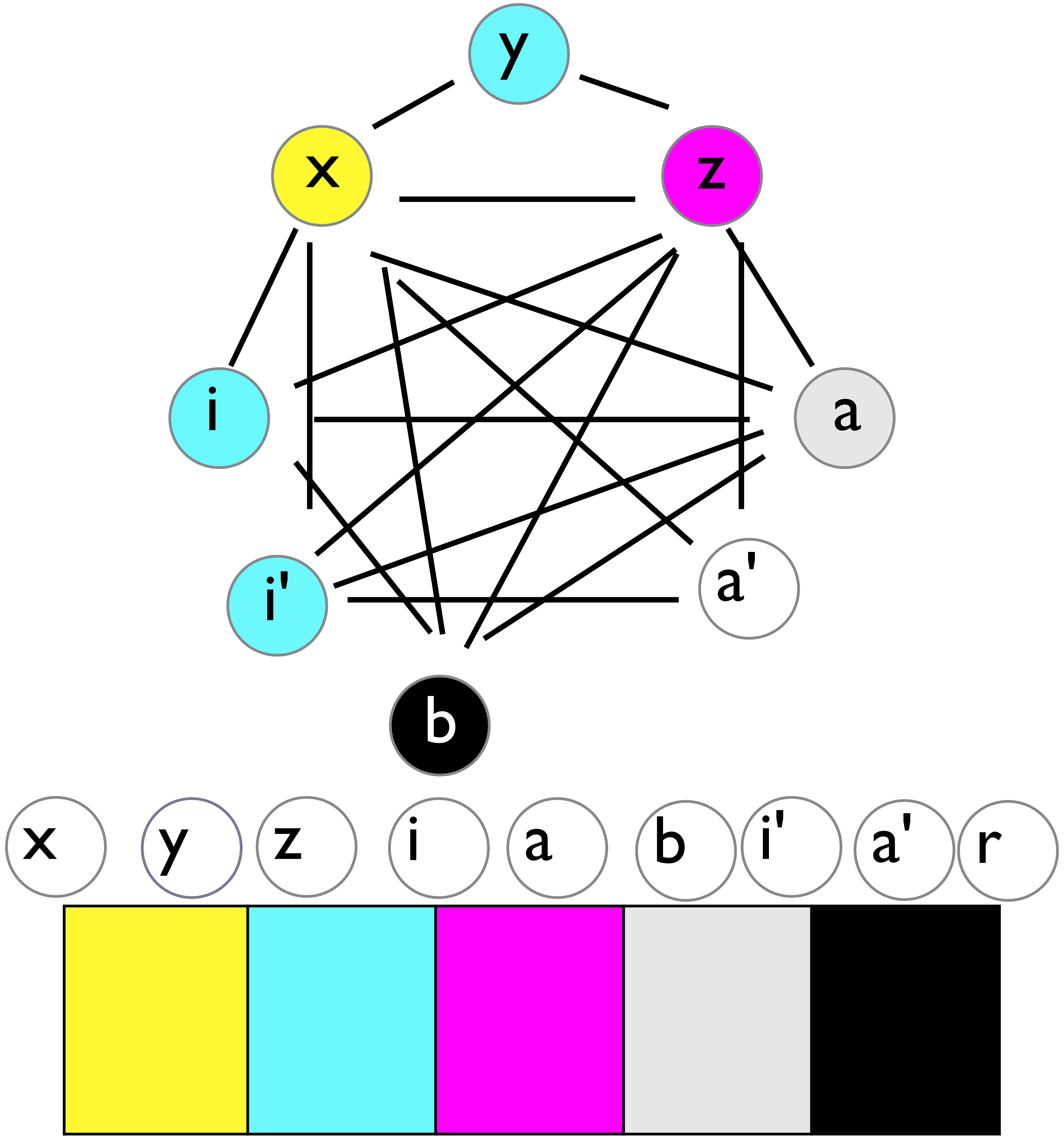
Interference Graph



SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

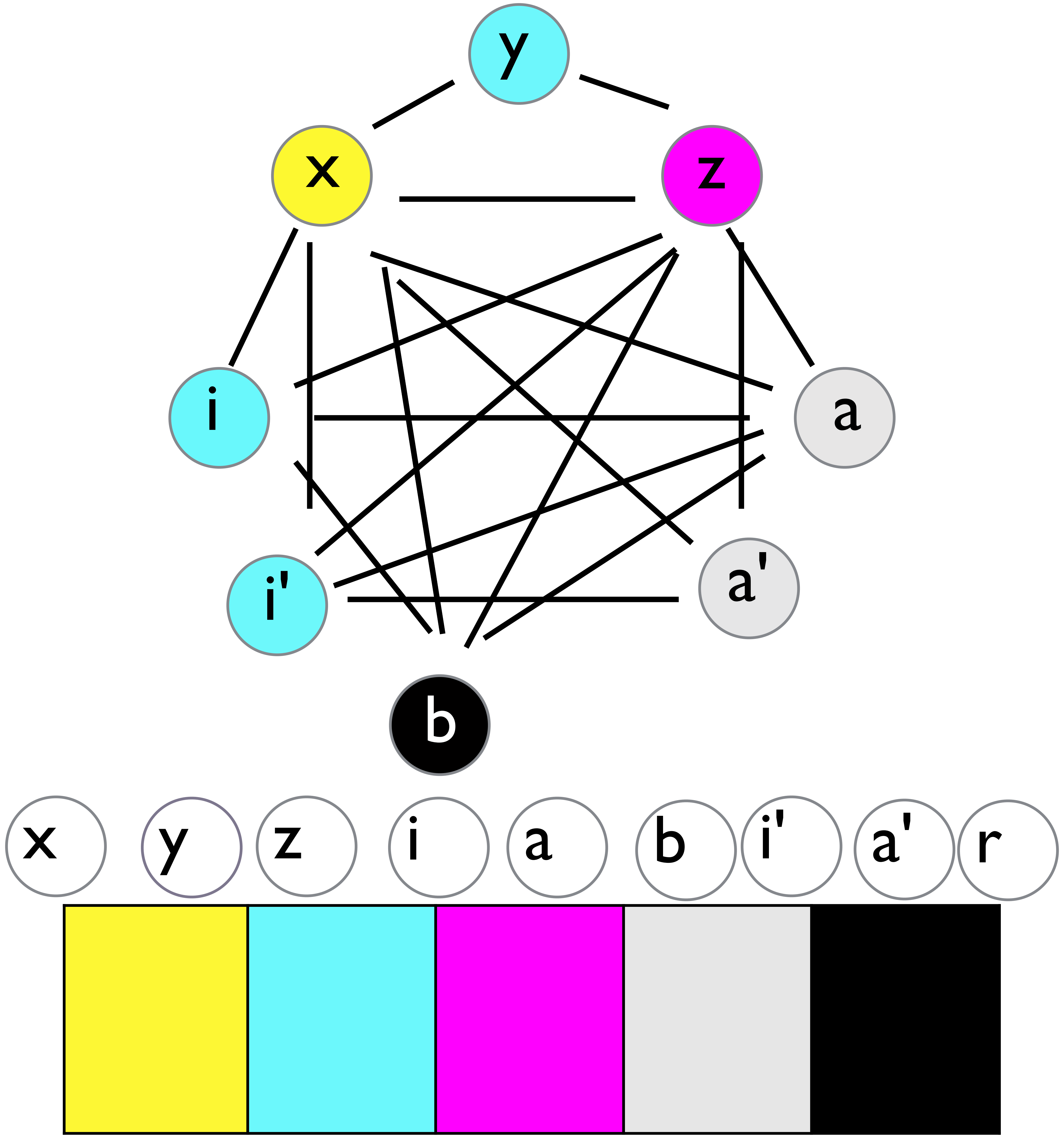
Interference Graph



SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

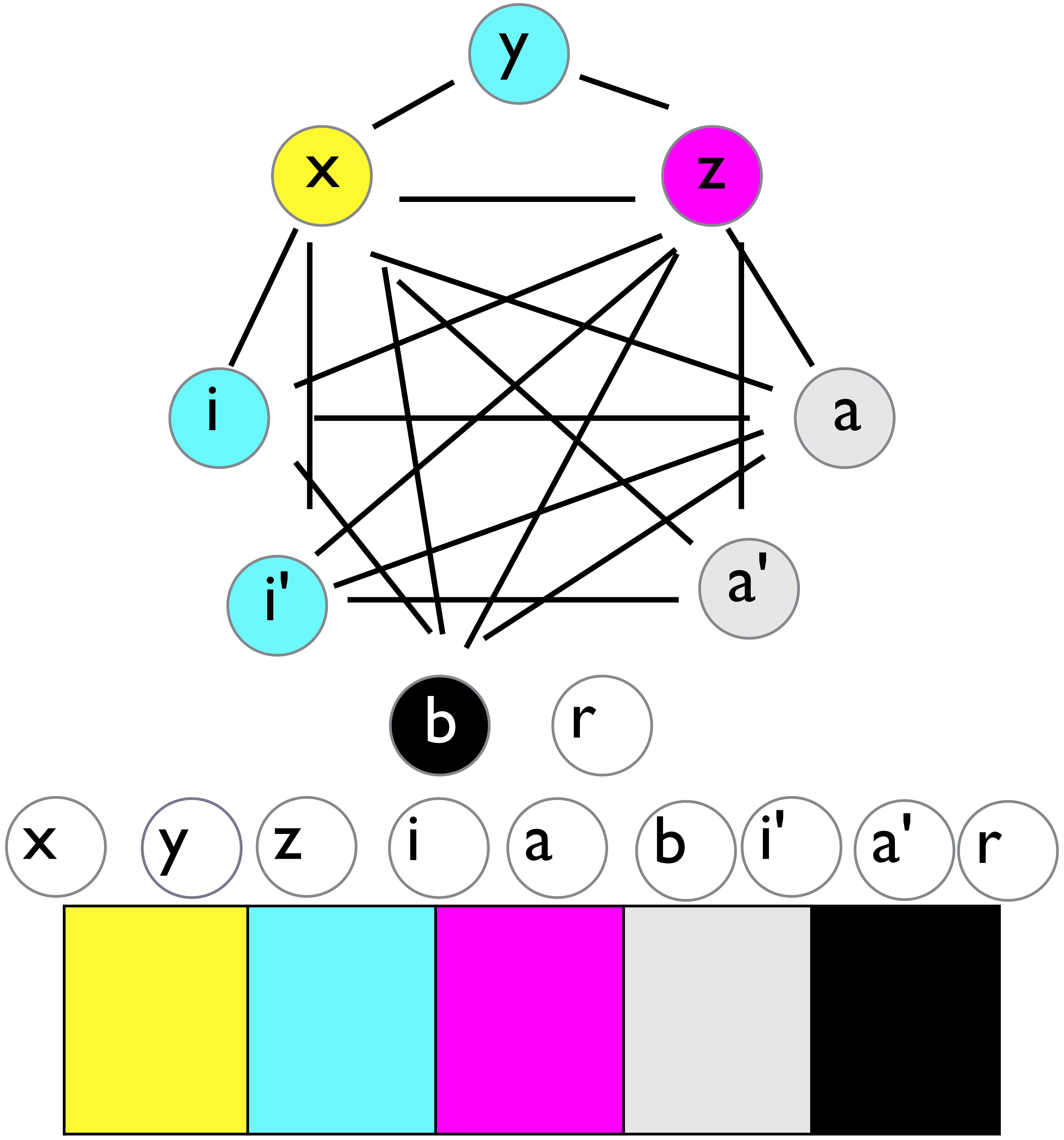
Interference Graph



SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

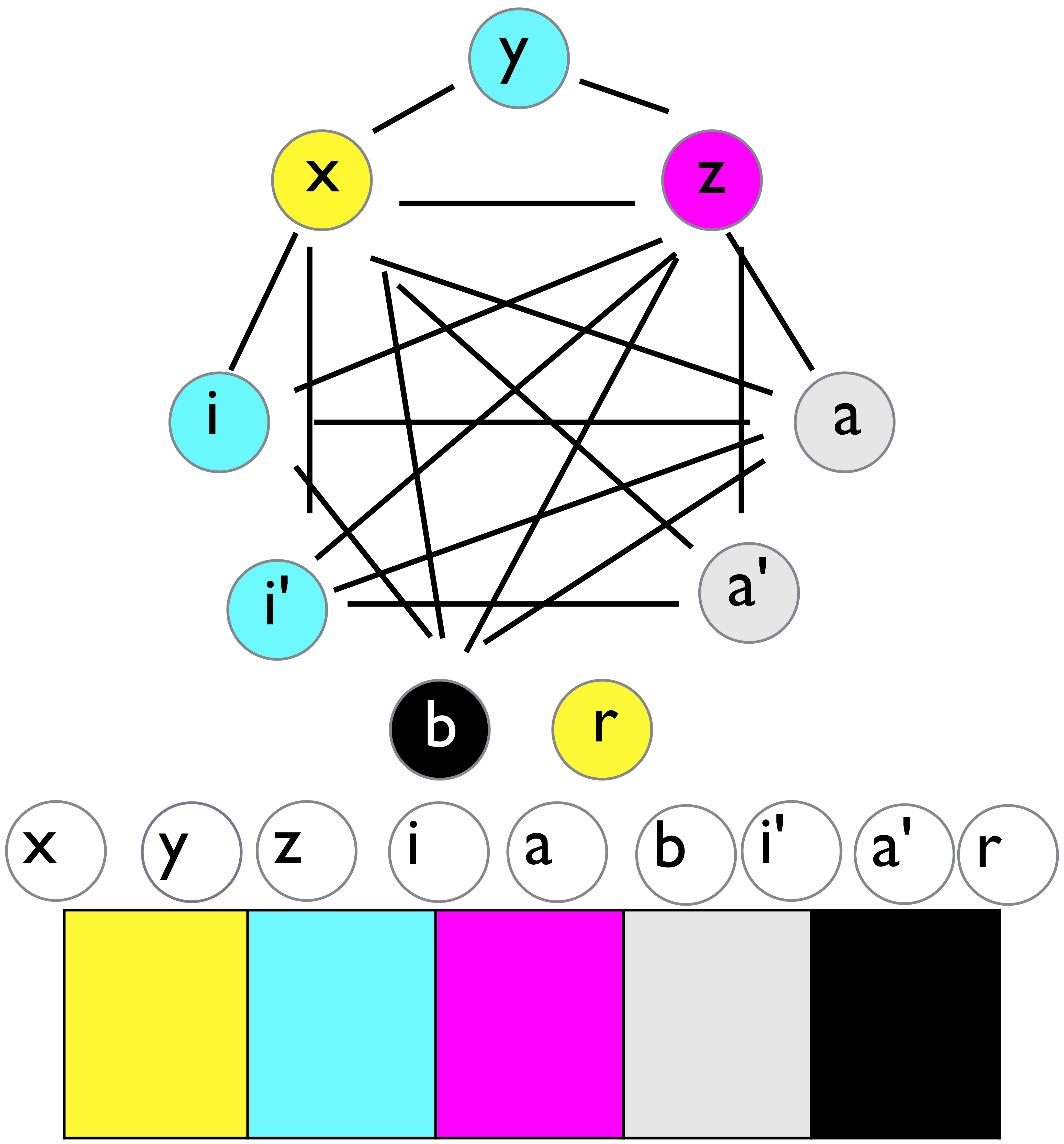
Interference Graph



SSA Program

```
f(x,y,z):
  loop(i,a):
    thn():
      r = a * z
      ret r
    els():
      i' = i - 1
      a' = a + x
      br loop(i', a')
    b = i == 0
    cbr b thn() els()
  br loop(y, 0)
```

Interference Graph



Incorporating Register Allocation

Perform register allocation on all the **blocks** of our SSA program. Treat function blocks specially.

Effects on Codegen

Store results directly in the output register

x = y - z

```
mov rx, ry
sub rx, rz
```

Does this always work? Need to be careful if output register is the same as one of the input registers (e.g., rx = rz)

Can either use a scratch register, or op-specific tricks:

```
sub rx, ry
imul rx, -1
```

Spilling

If a variable is picked to be spilled:

- When it is assigned to, store the result in memory
- When it is used, access memory

One issue:

x = y + z

If all x,y,z are spilled, cannot implement this without a register.

Easy solution: reserve one scratch register for this purpose:

```
mov r, [rsp - off(y)]
add r, [rsp - off(z)]
mov [rsp - off(x)], r
```

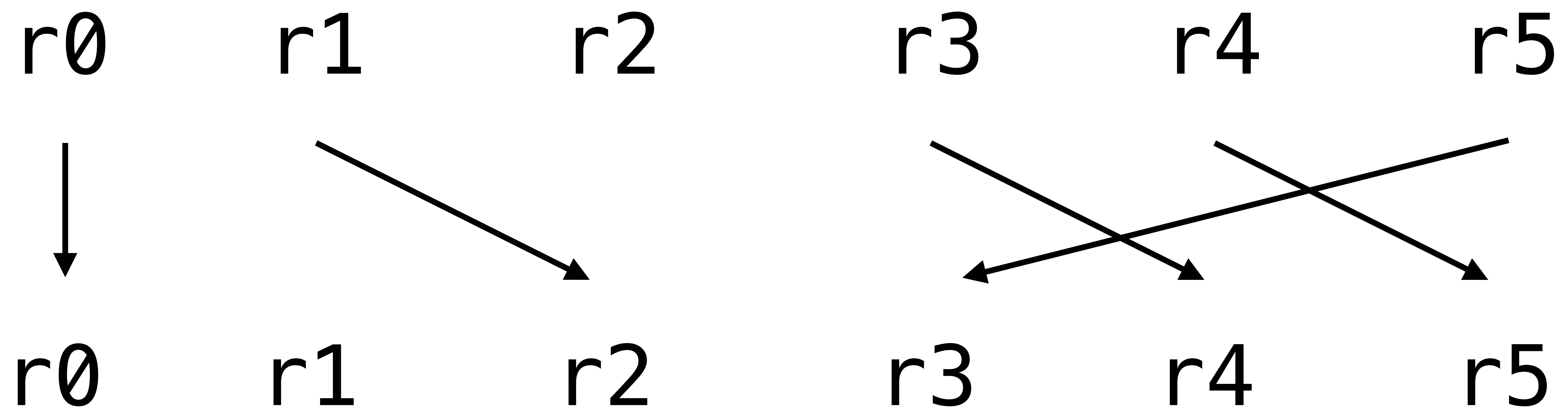
Implementing Branch with Arguments

```
f(a,b,c): ...
...
br f(x,y,z)
```

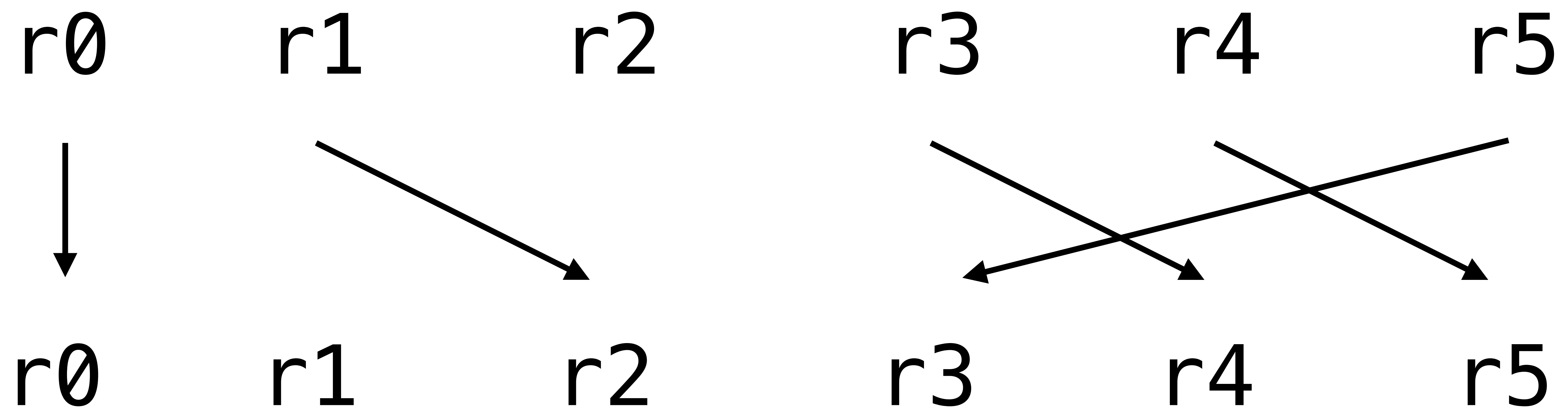
```
mov r_x, r_a
mov r_y, r_b
mov r_z, r_z
jmp f
```

```
what if a,b,c registers and
x,y,z registers overlap?
```

Implementing Branch with Arguments

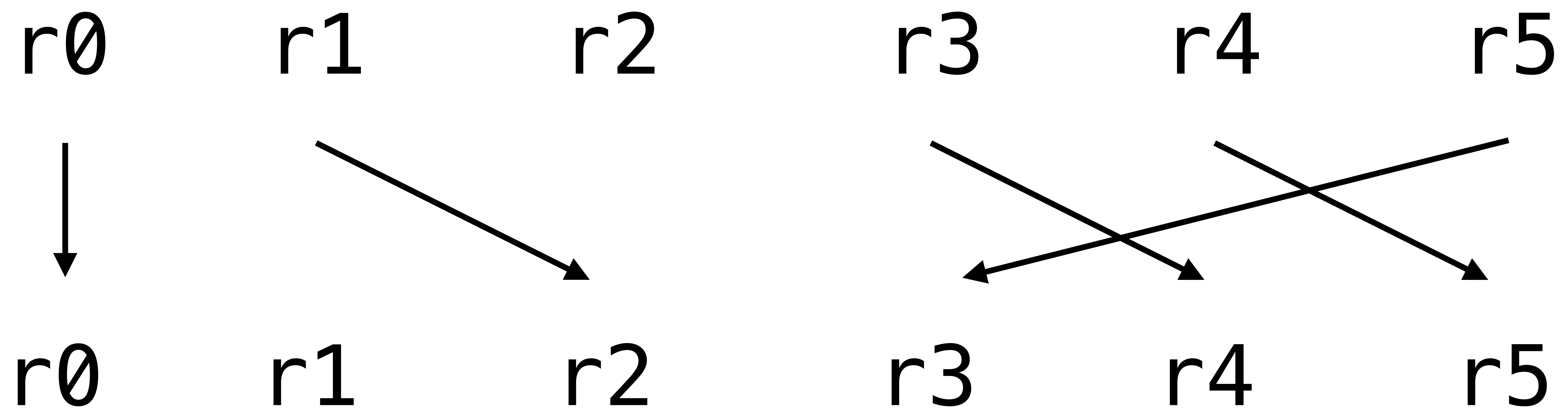


Implementing Branch with Arguments



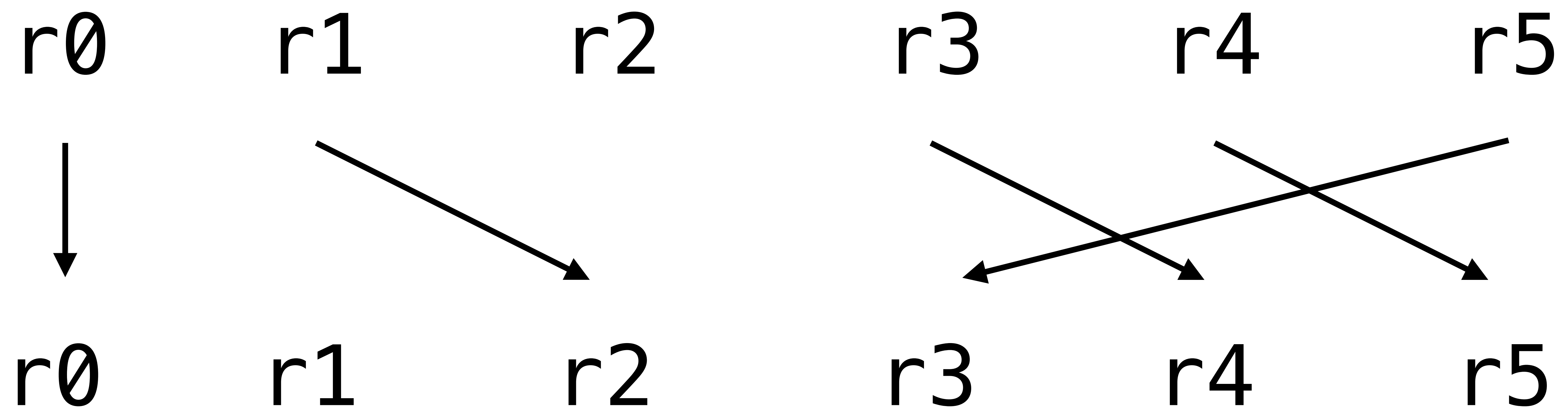

```
nop
```

Implementing Branch with Arguments



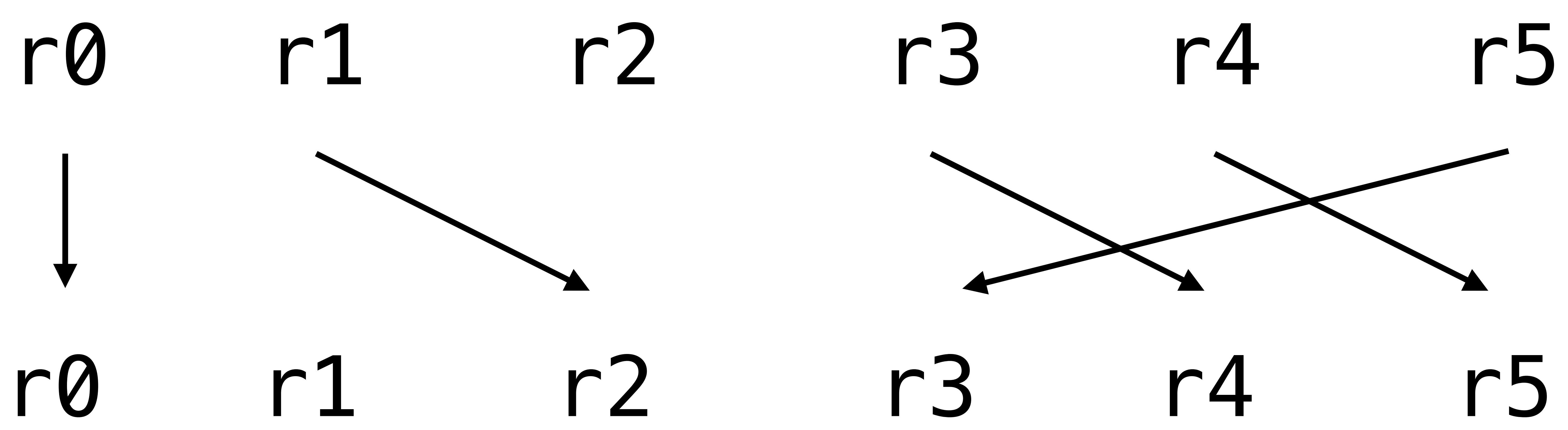

```
nop  mov r2, r1
```

Implementing Branch with Arguments




```
nop  mov r2, r1
```

Implementing Branch with Arguments




```
nop  mov r2, r1     xchg r3, r4
                    xchg r3, r5
```

xchg is like mov, but **exchanges** the values without need for an extra register.
Faster than xor swapping

SSA reg allocation is polytime, but minimizing the resulting number of movs/xchg is NP hard

Register Allocation vs Calling Conventions

Now that we are using registers we need to take care to respect treatment of registers in the calling conventions we use.

Register Allocation vs Calling Conventions

Now that we are using registers we need to take care to respect treatment of registers in the calling conventions we use.

In System V AMD 64 Calling convention, registers are divided into two classes:

- **volatile** aka **caller-save**: when you make a call, the value of these registers may change when the callee returns
- **non-volatile** aka **callee-save**: when you make a call, the value of these registers will be the same when the callee returns

Volatile/Caller Save registers

volatile aka **caller-save**

x = ...

y = f(z)

z = x + y

if **x** is stored in a volatile register, its value may be overwritten by the function **f**.

- Simple solution: save all live volatiles to the stack before a call, restore after the call
- Better solution: add nodes to interference graph for volatile registers, add conflicts at every non-tail call

Non-volatile/Callee Save registers

```
y = ...
z = x + y
ret z
```

if **y** is stored in a **non-volatile** register, the value of the register must be **saved** on entry and **restored** when we return

- Solution: save all used non-volatiles to the stack at the beginning of every global function def, restore them before every return/external tail call
- Start spilled variables **after** the saved non-volatile registers

Implementing function blocks

```
fun f_fun(a,b,c,...):
  br f_tail(a,b,c,...)
```

1. Save all used non-volatiles/callee-save registers

2. Treat f_fun's args are pre-determined by the calling convention, otherwise similar to any branch with args

mov [rsp - 8], rbx

mov [rsp - 16], rbp

...

; br f_tail(...)

Implementing ret

```
ret x
```

- Move x into rax
- Restore non-volatile/callee-saves

mov rax, loc(x)

mov rbx, [rsp - 8]

mov rbp, [rsp - 16]

...