Graph Coloring
Register Allocation

October 20, 2021

Register Allocation

The compiler needs to decide which variables to store in which registers, which registers to “spill” onto the stack

Register Allocation

4 Steps

Register Allocation

4 Steps

1. Liveness analysis: identify which variables are needed at every point in the program

Register Allocation

4 Steps

1. Liveness analysis: identify which variables are needed at every point in the program
2. Conflict analysis: based on liveness info, identify which variables *cannot* be assigned the same register

Register Allocation

4 Steps

1. Liveness analysis: identify which variables are needed at every point in the program
2. Conflict analysis: based on liveness info, identify which variables *cannot* be assigned the same register
3. Graph Coloring: based on conflict information, assign registers to variables so that conflicting vars get different registers

Register Allocation

4 Steps

1. Liveness analysis: identify which variables are needed at every point in the program
2. Conflict analysis: based on liveness info, identify which variables *cannot* be assigned the same register
3. Graph Coloring: based on conflict information, assign registers to variables so that conflicting vars get different registers
4. Spilling: if graph coloring fails, pick a variable to put on the stack and retry

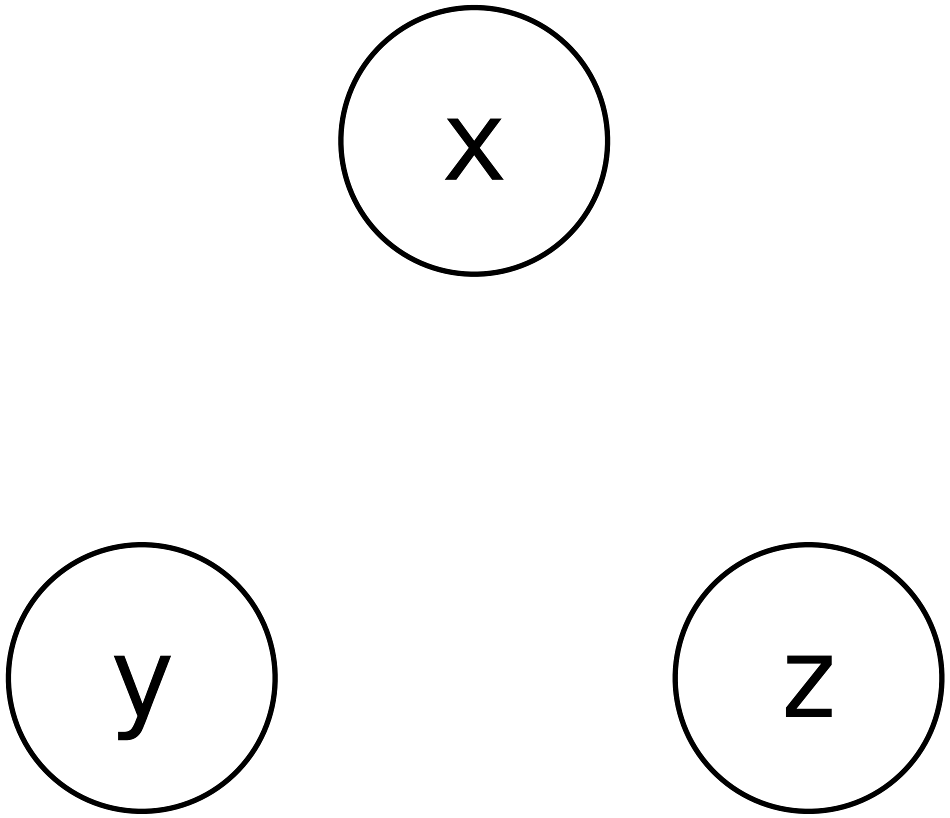
Example

```
def f(a, b):
  let x = a + b in
  let y = g(x)  in
  let z = x * y in
  h(x, z)
end
```

x, y conflict
x, z conflict

Example

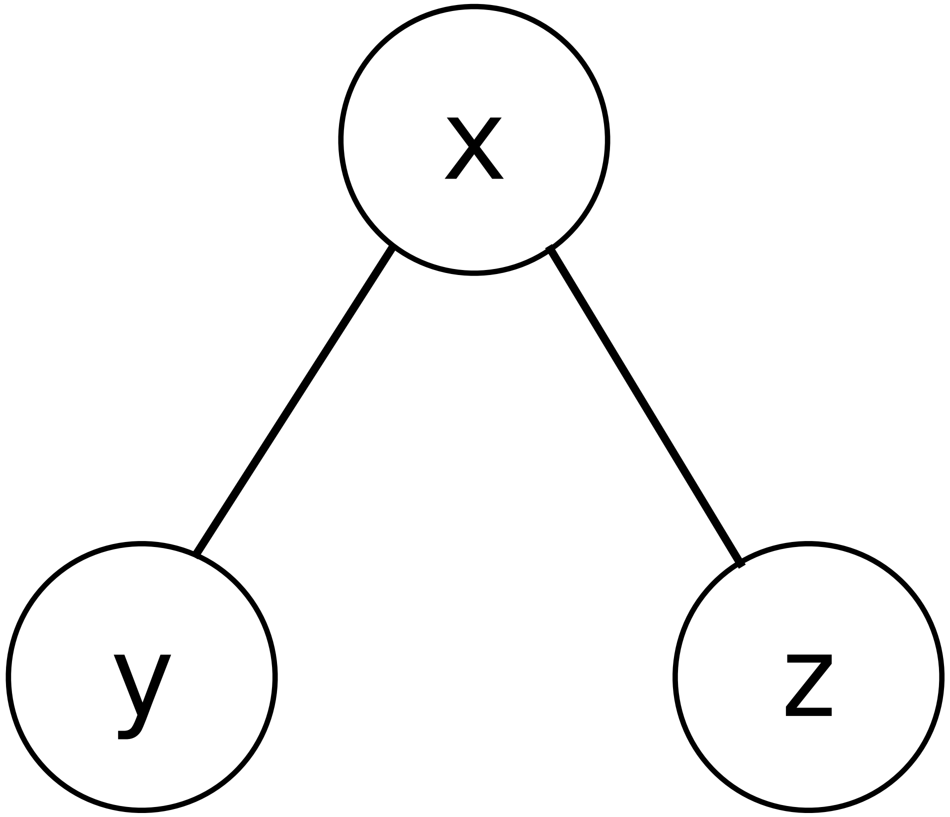
```
def f(a, b):
  let x = a + b in
  let y = g(x)  in
  let z = x * y in
  h(x, z)
end
```



x, y conflict
x, z conflict

Example

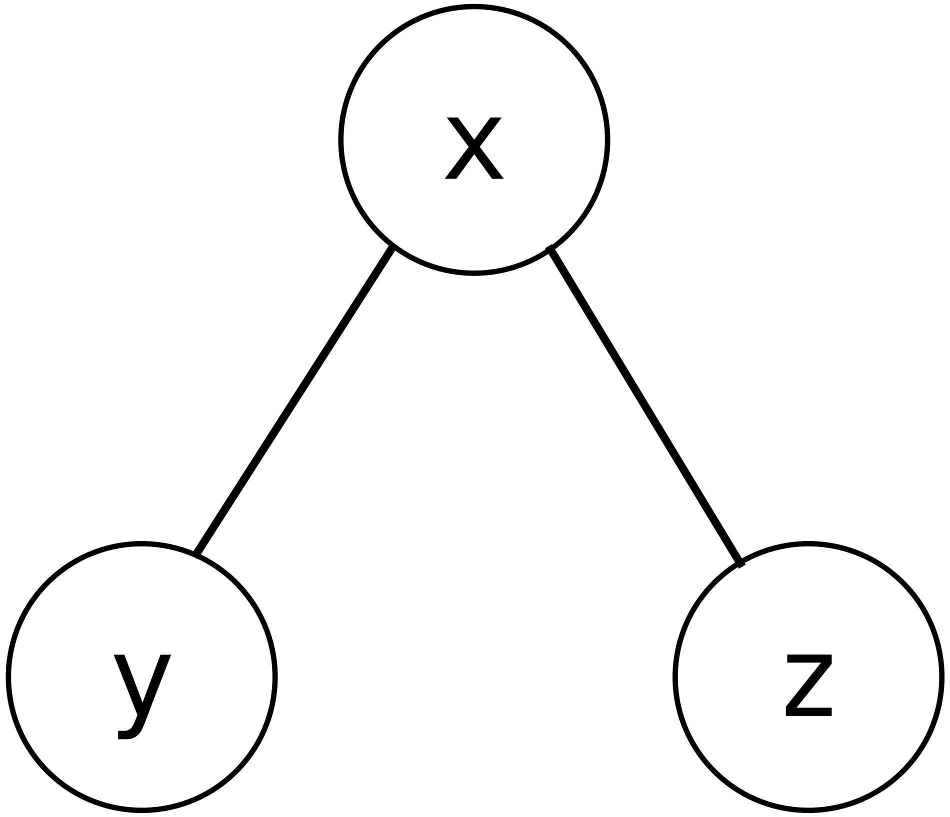
```
def f(a, b):
  let x = a + b in
  let y = g(x)  in
  let z = x * y in
  h(x, z)
end
```



x, y conflict
x, z conflict

Example

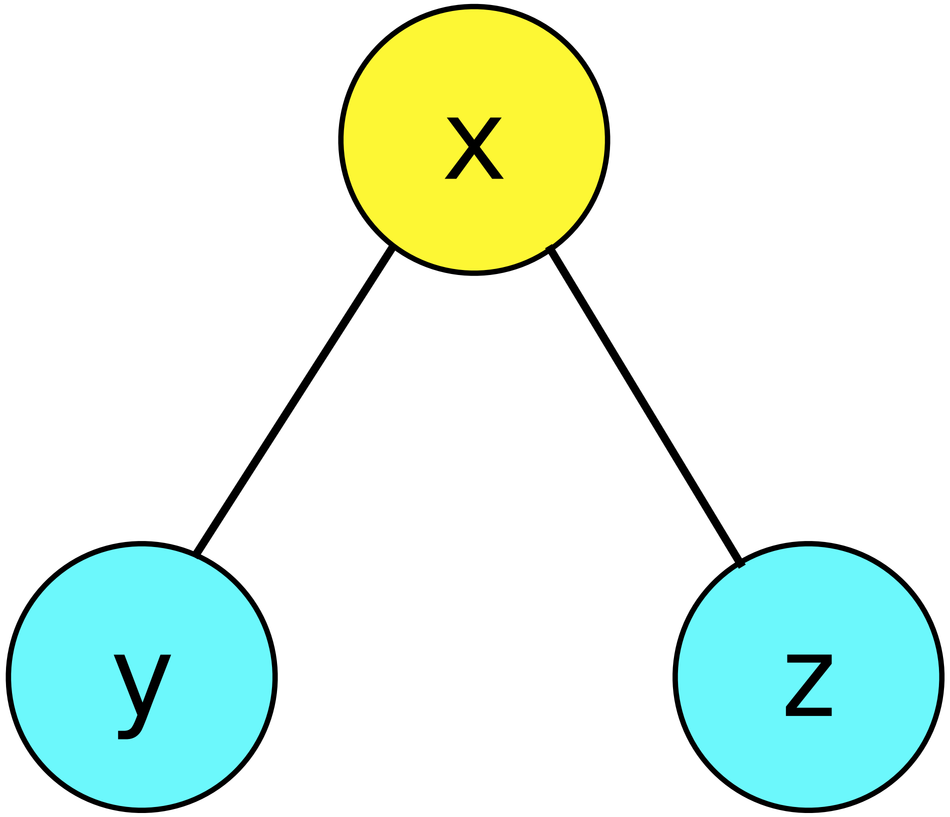
```
def f(a, b):
  let x = a + b in
  let y = g(x)  in
  let z = x * y in
  h(x, z)
end
```

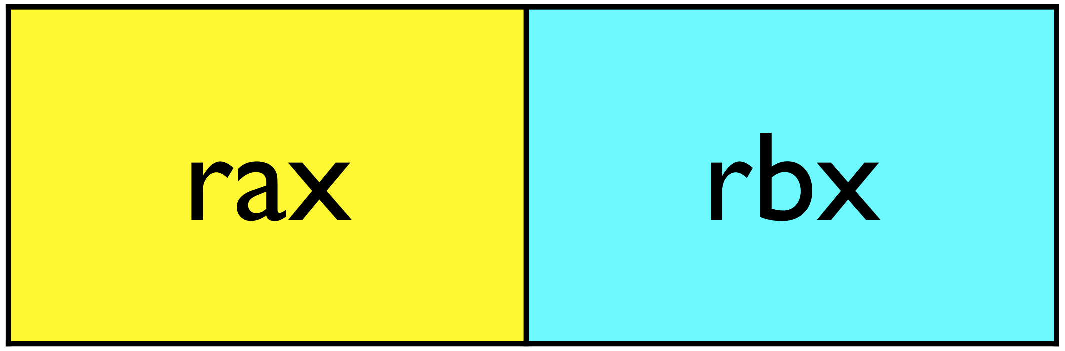


rax	rbx

Example

```
def f(a, b):
  let x = a + b in
  let y = g(x)  in
  let z = x * y in
  h(x, z)
end
```





Graph Coloring Register Allocation

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

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

Limitation: Computational Complexity

Graph coloring is an NP-hard problem.

Limitation: Computational Complexity

Graph coloring is an NP-hard problem.

So no polynomial-time algorithm is known.

Limitation: Computational Complexity

Graph coloring is an NP-hard problem.

So no polynomial-time algorithm is known.

Concession: we'll use heuristics and accept that we won't always get an optimal solution

Limitation: Computational Complexity

Graph coloring is an NP-hard problem.

So no polynomial-time algorithm is known.

Concession: we'll use heuristics and accept that we won't always get an optimal solution

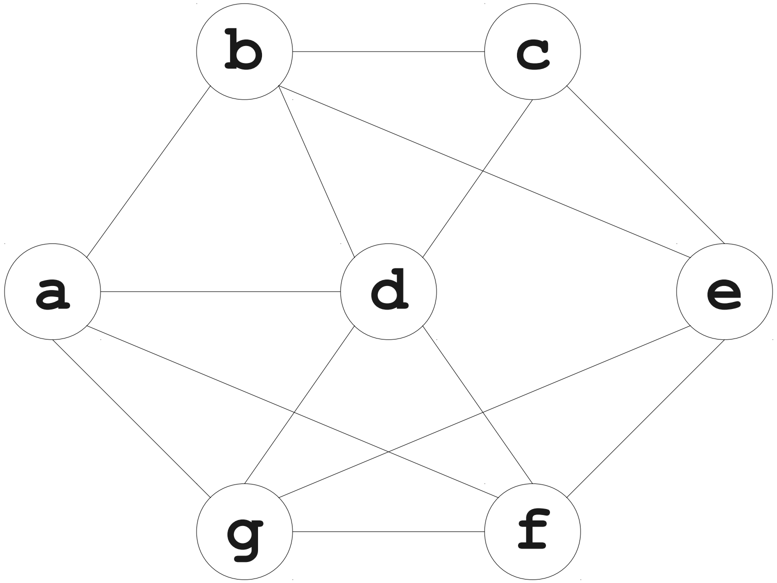
Even with heuristics, the algorithm is still n^2, and the slowest part of the compiler.

Chaitin's Algorithm

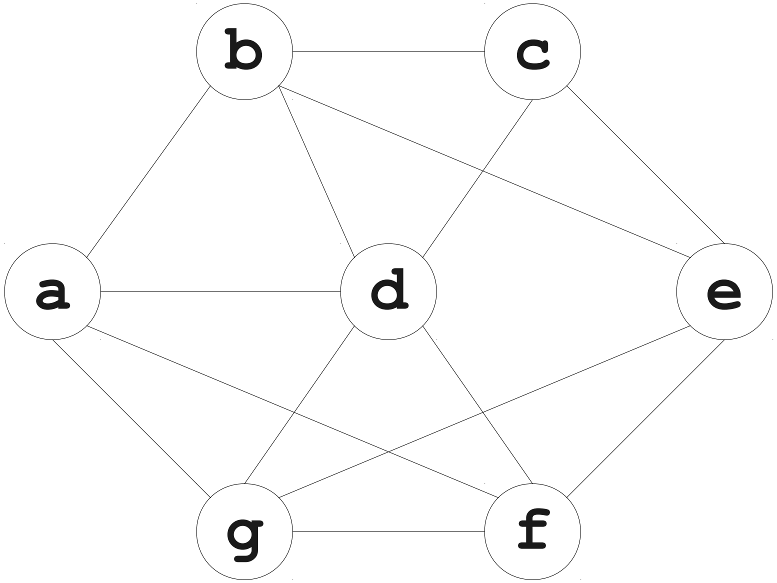
- Intuition:
  - Suppose we are trying to *k*-color a graph and find a node with fewer than *k* edges.
  - If we delete this node from the graph and color what remains, we can find a color for this node if we add it back in.
  - Reason: With fewer than *k* neighbors, some color must be left over.
- Algorithm: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



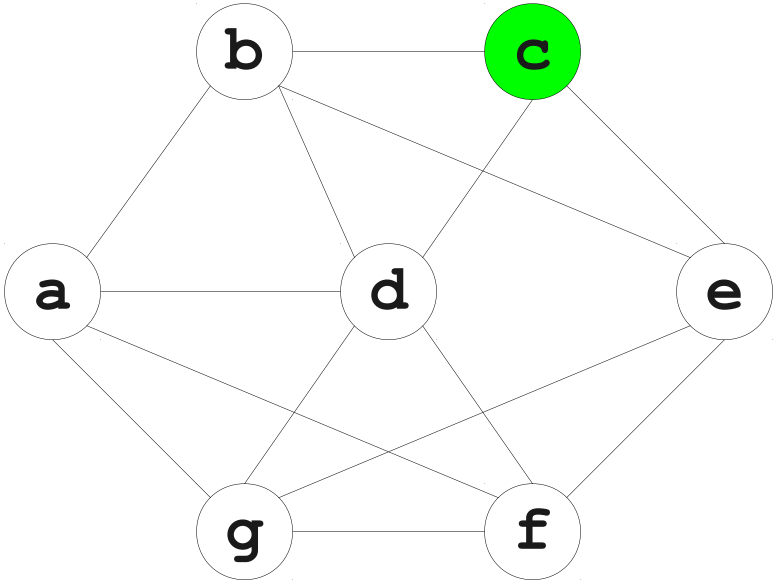
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

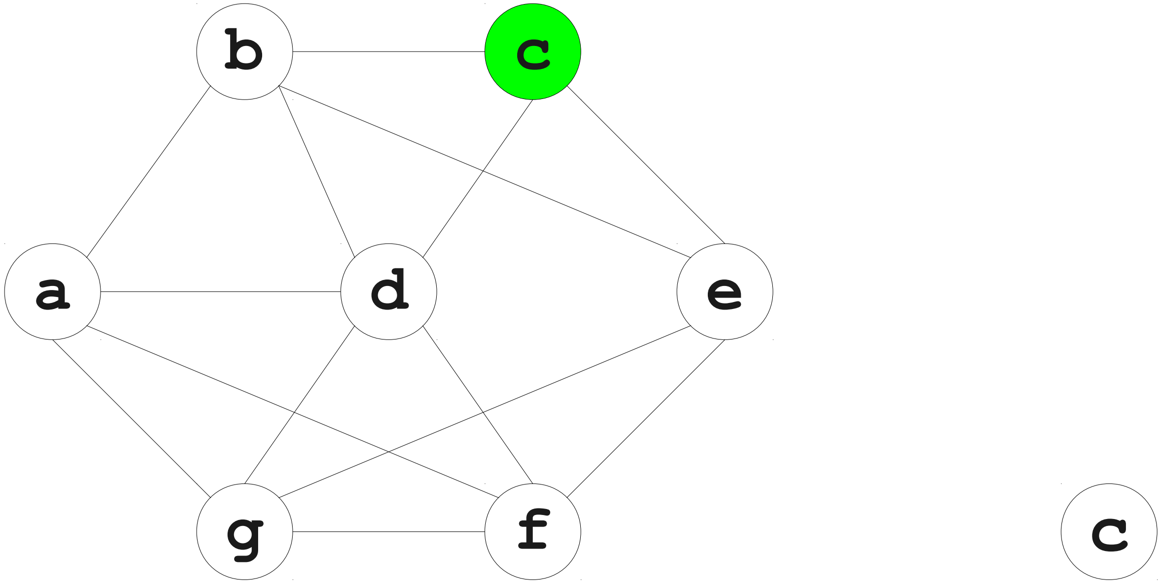
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

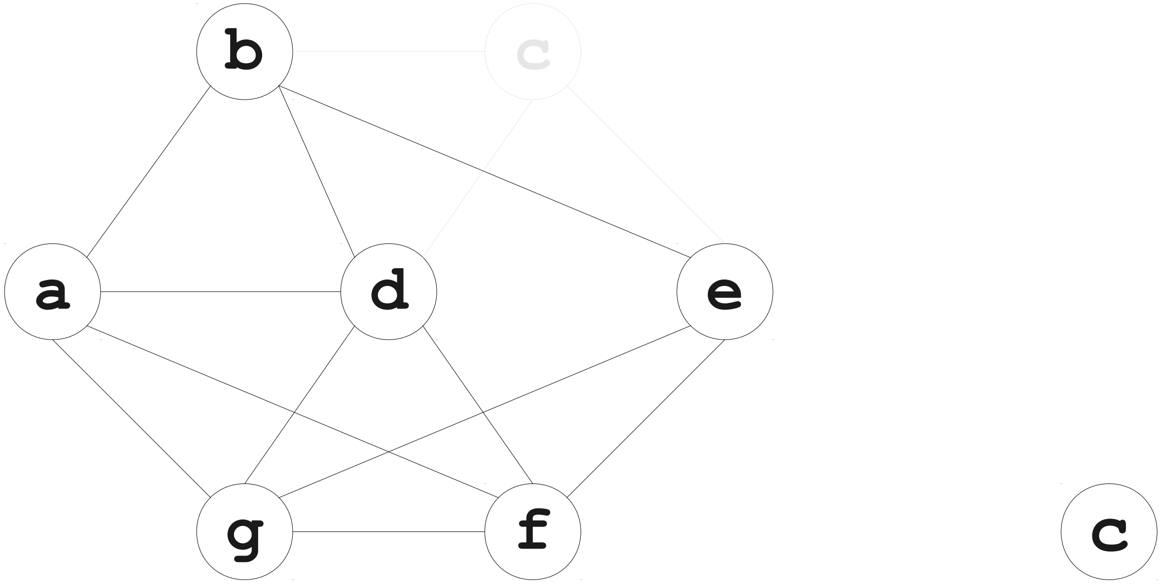
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

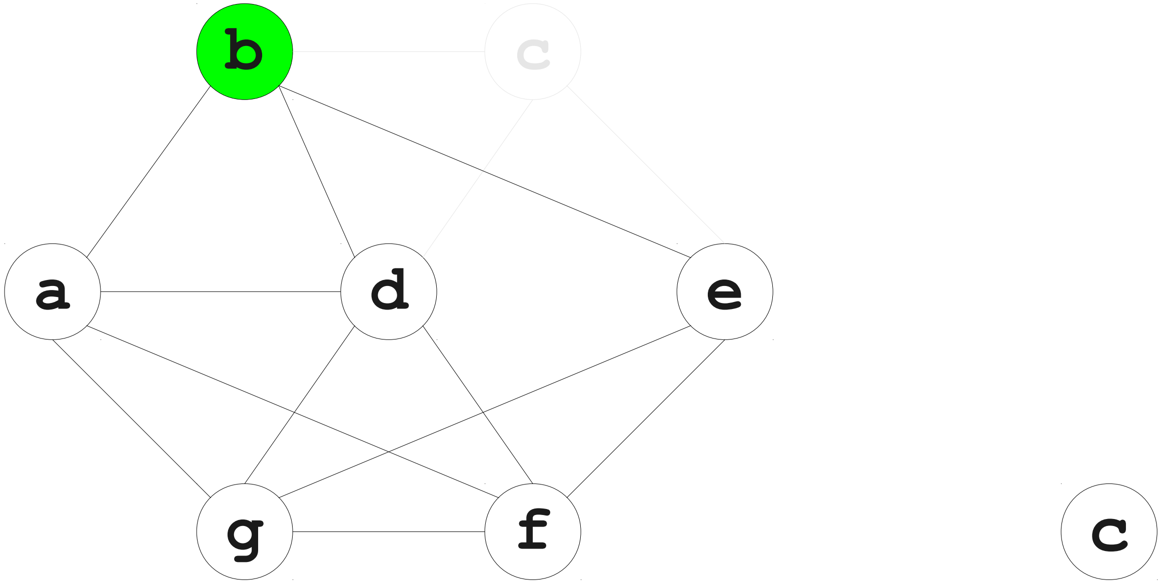
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

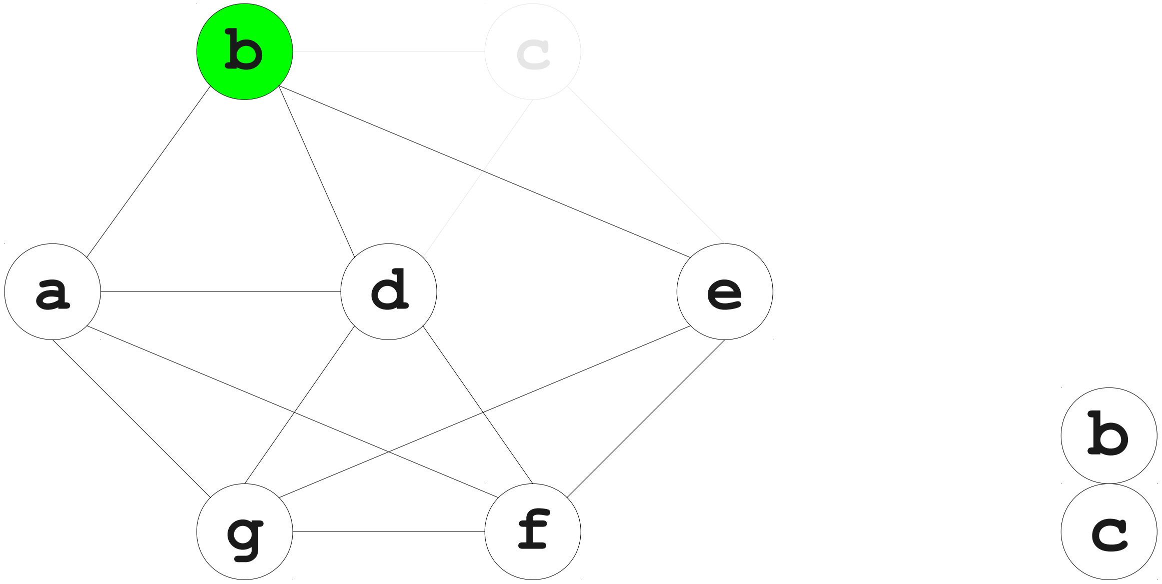
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

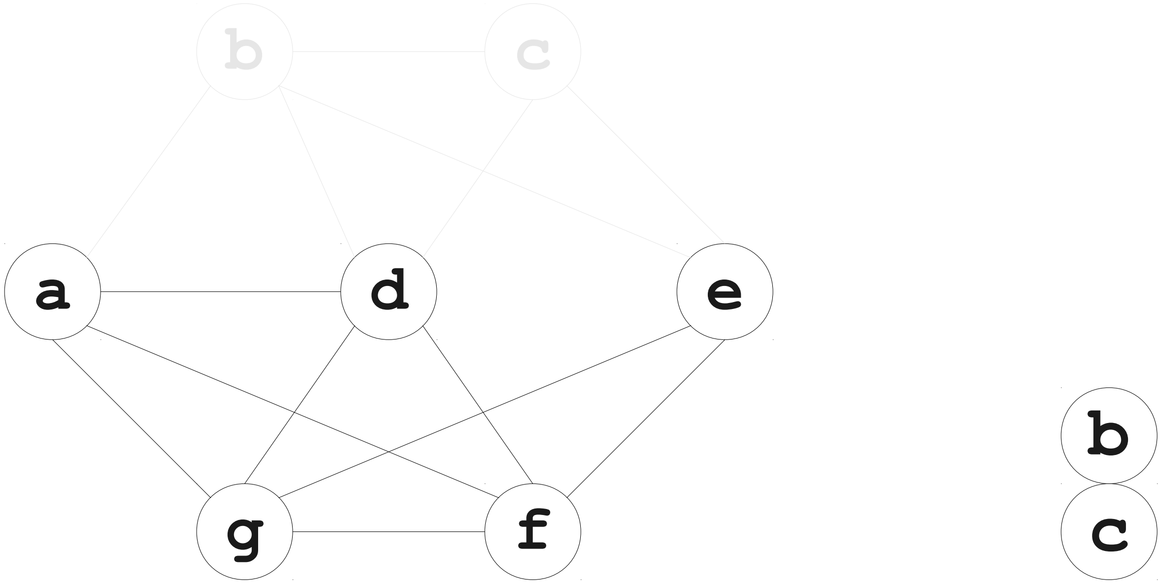
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

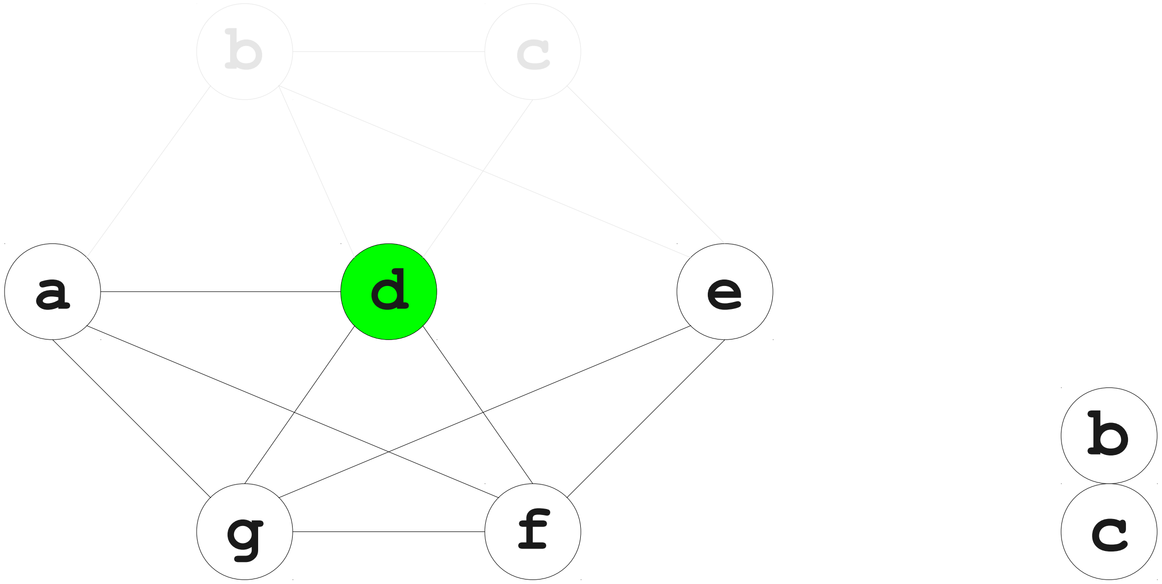
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

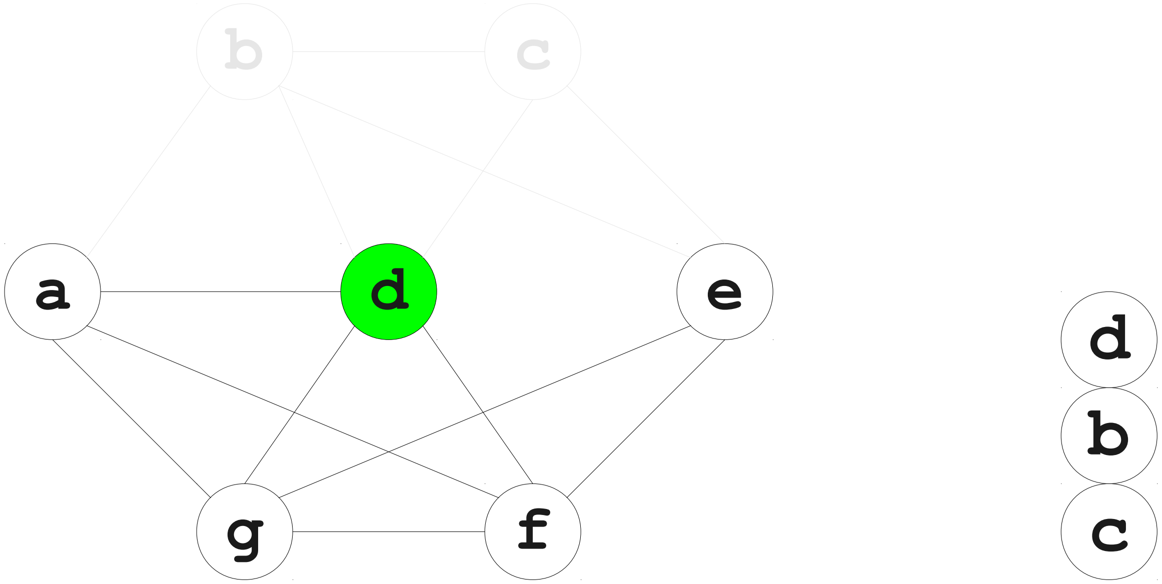
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

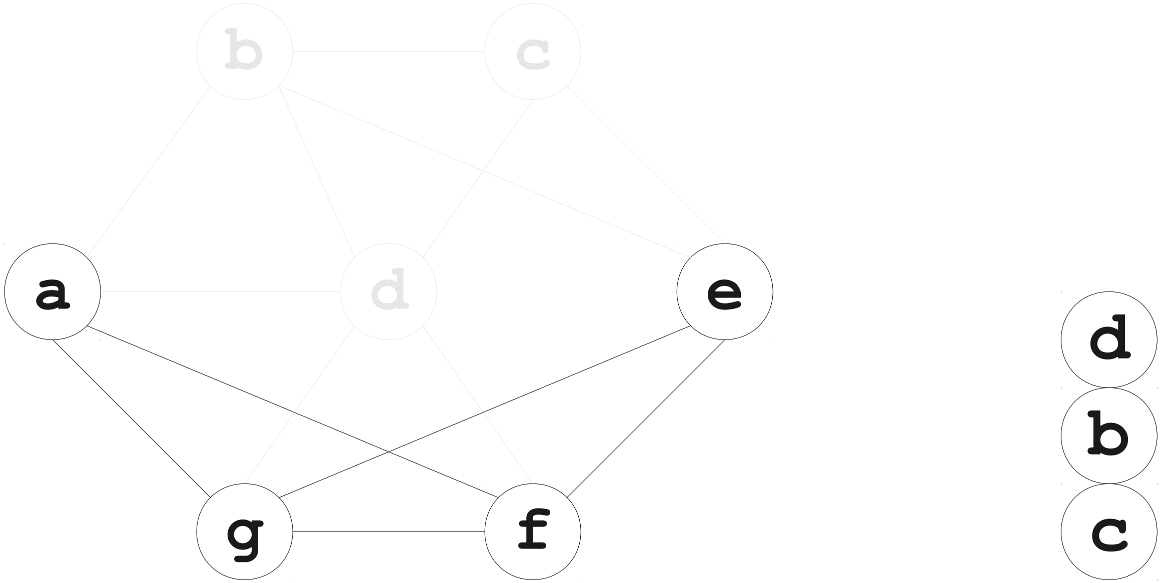
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

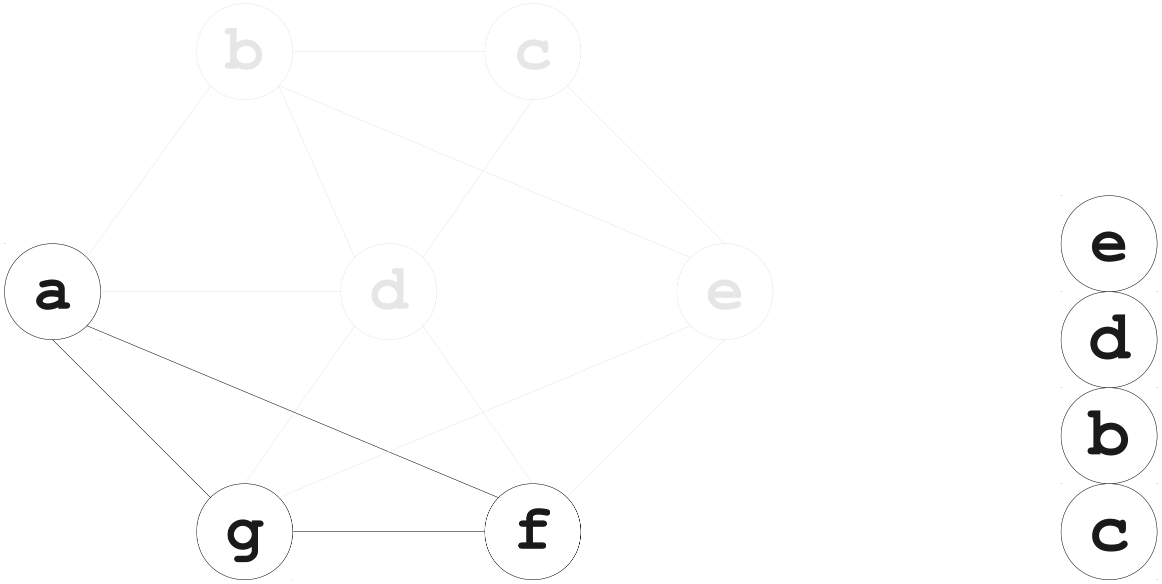
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

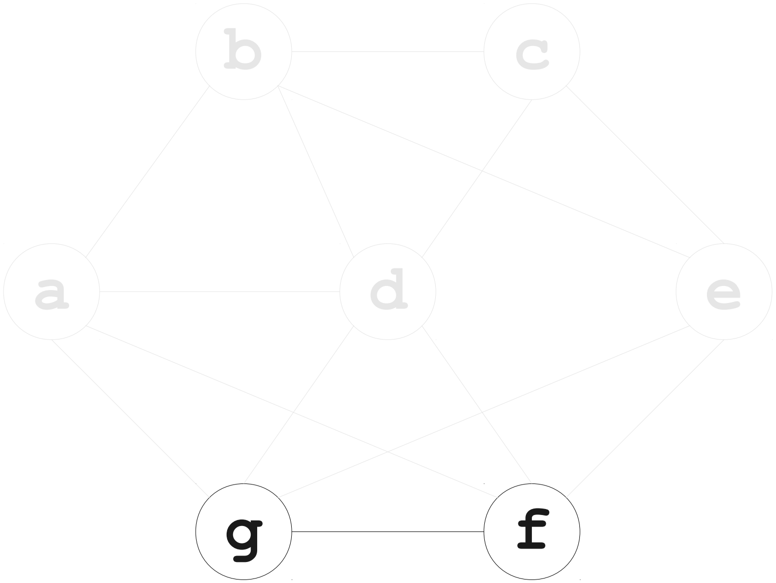
Chaitin's Algorithm

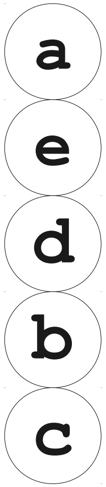


Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm

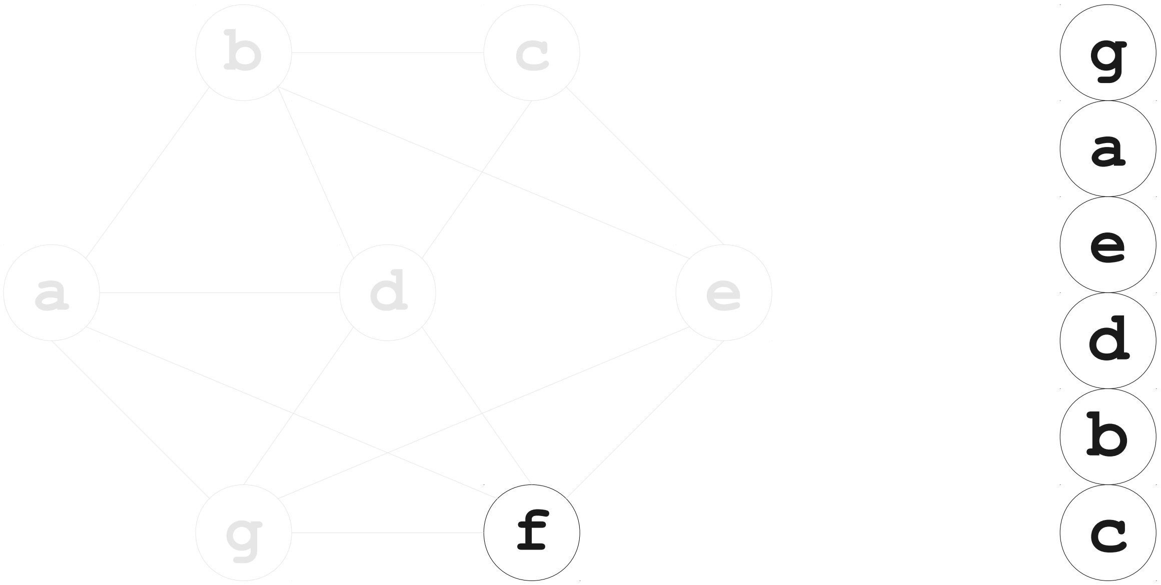




Registers

R_0	R_1	R_2	R_3

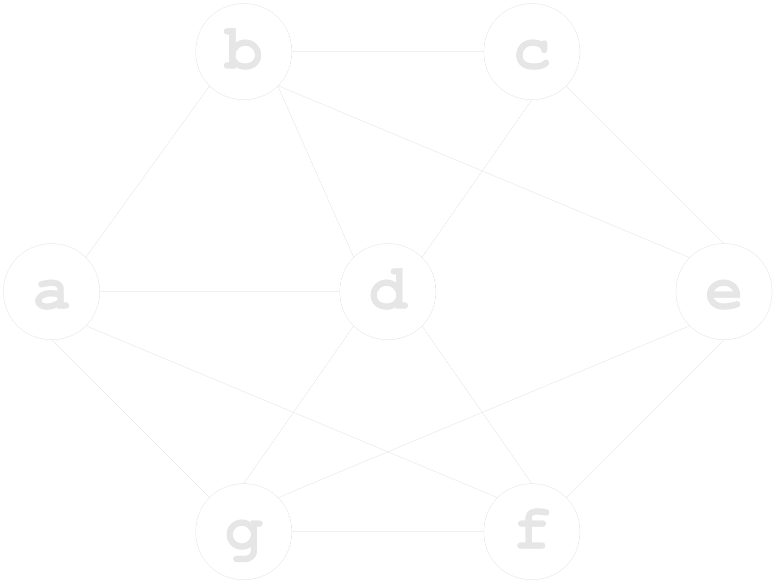
Chaitin's Algorithm

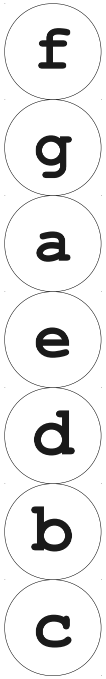


Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm

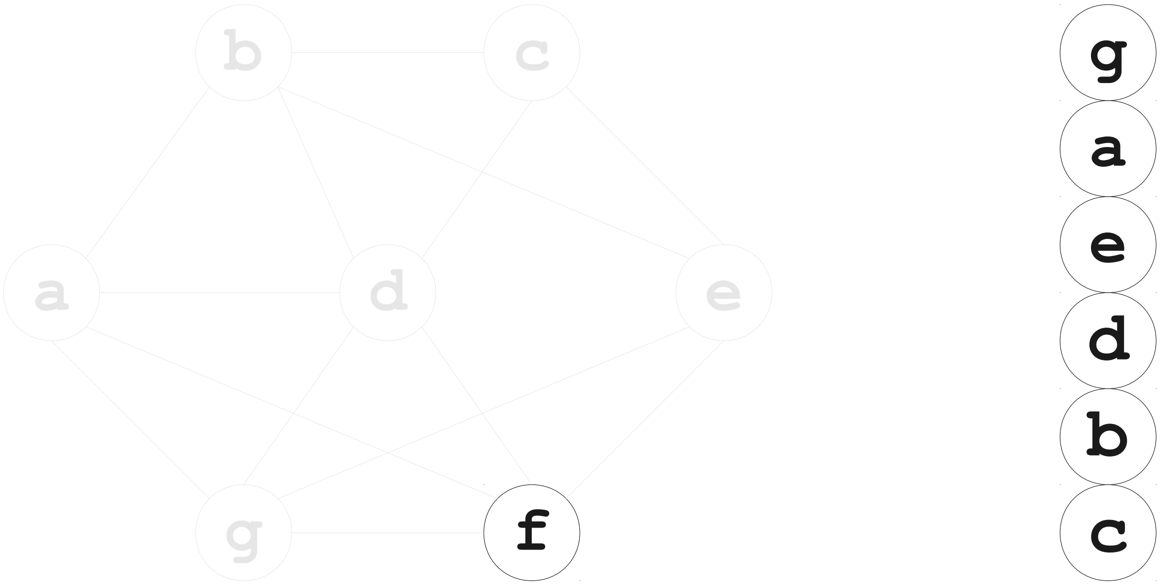




Registers

R_0	R_1	R_2	R_3

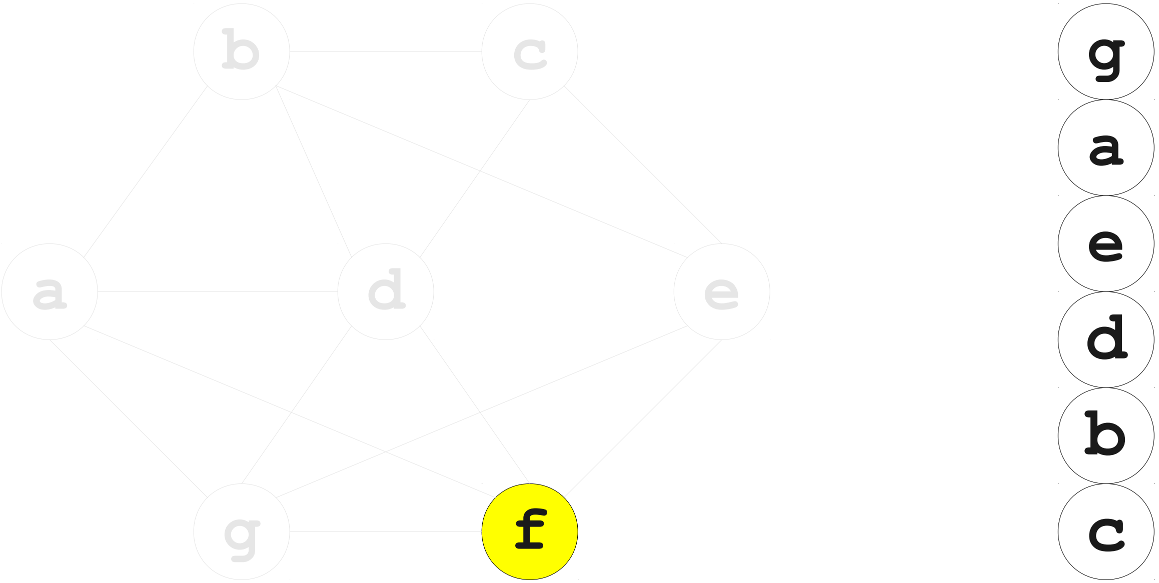
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

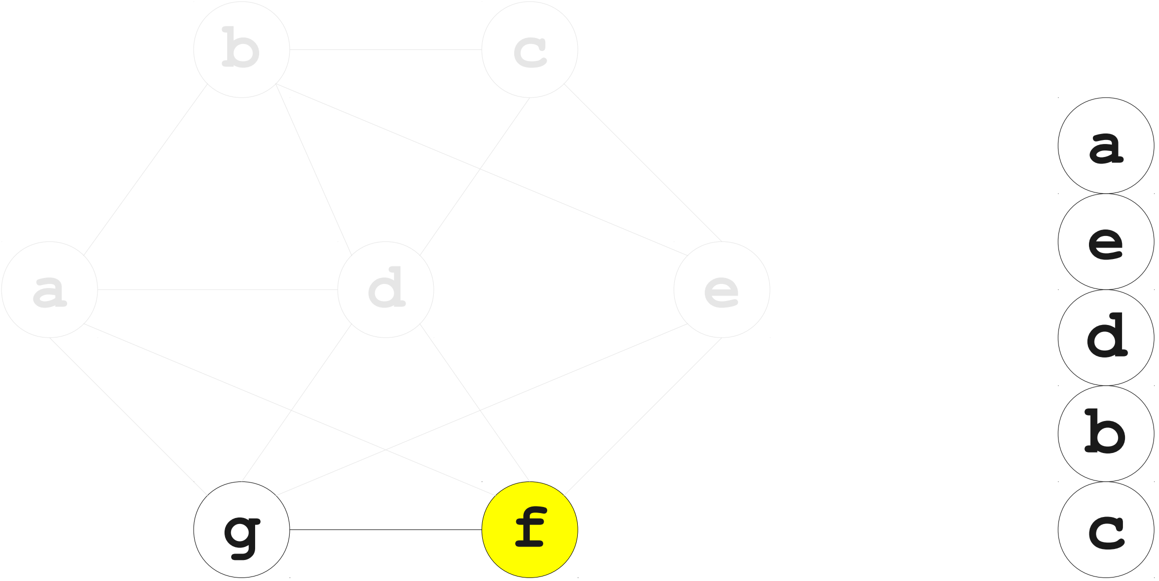
Chaitin's Algorithm



Registers

R_0	**R_1**	**R_2**	**R_3**

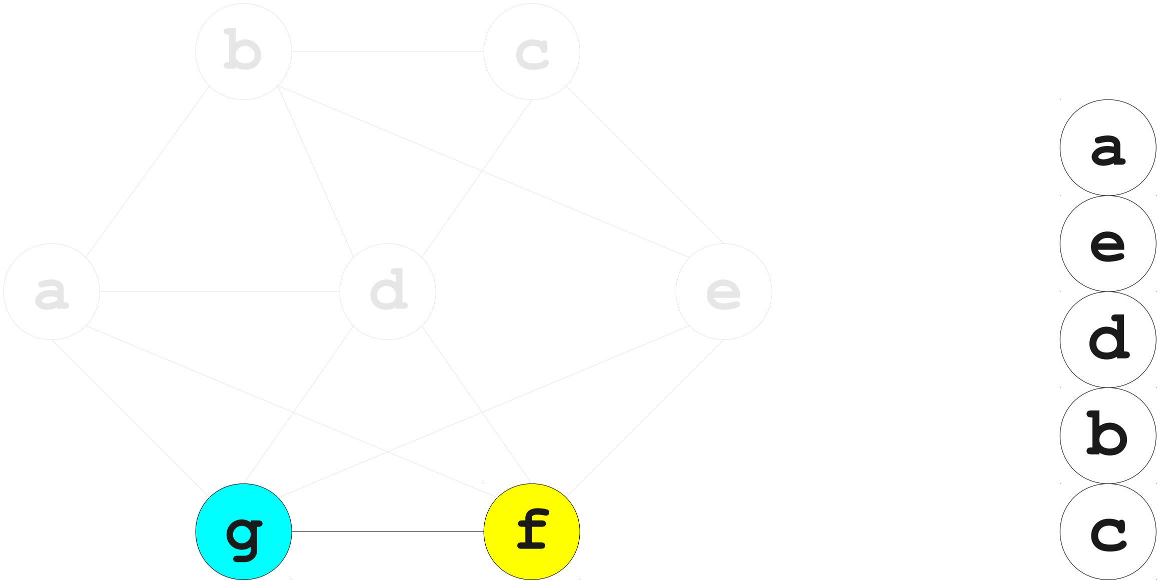
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

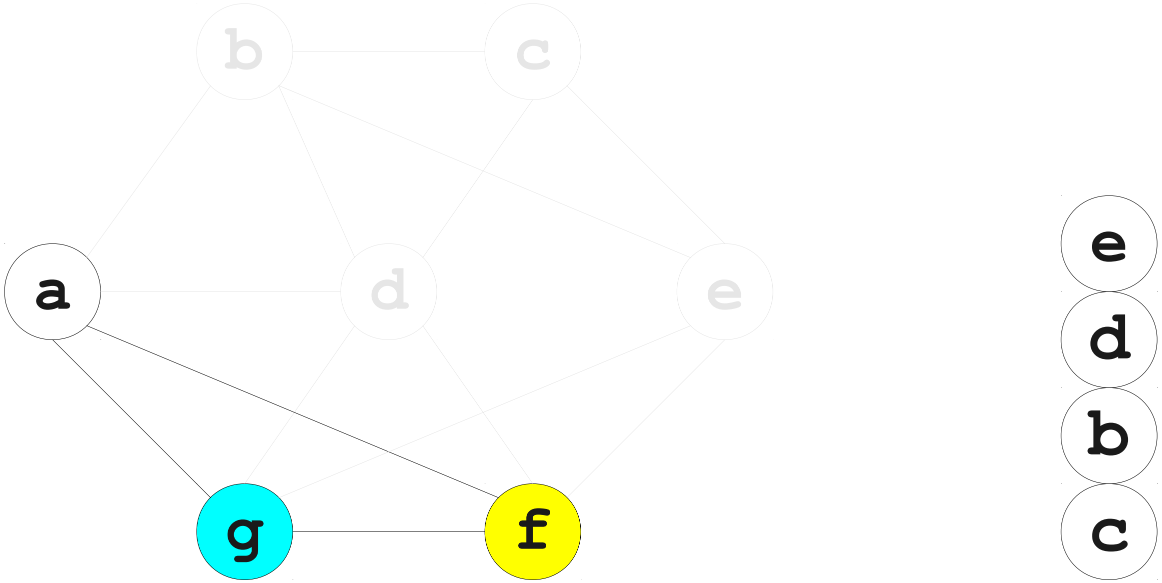
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

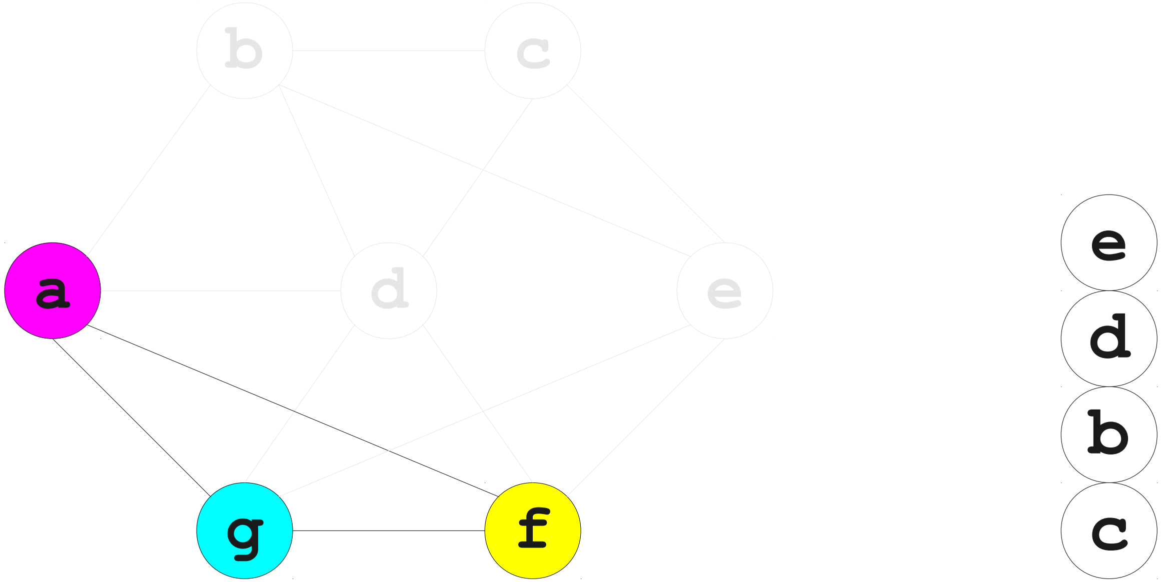
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

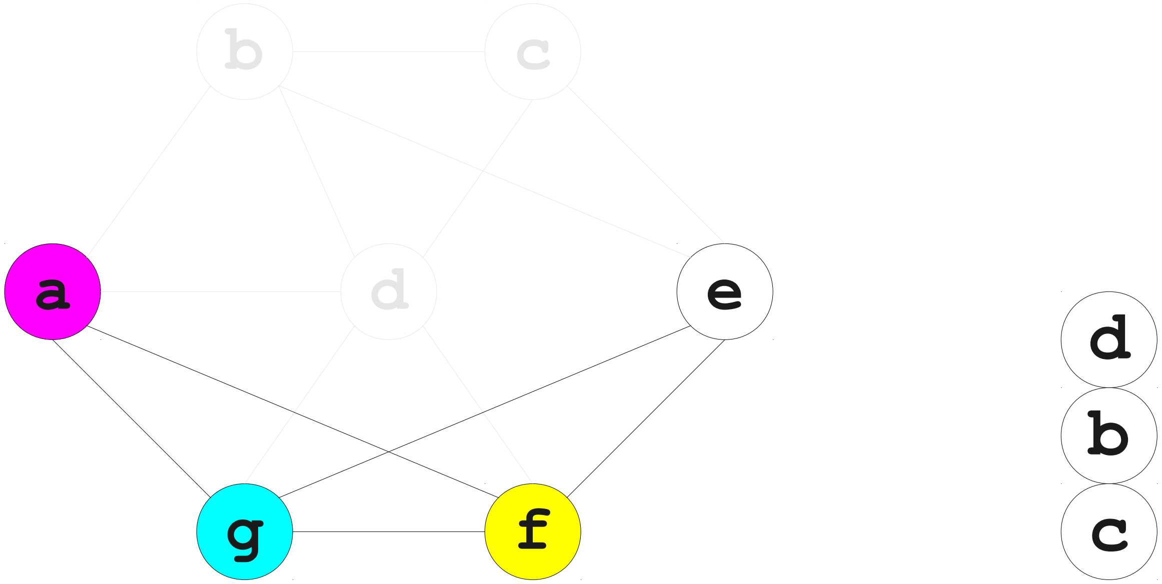
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

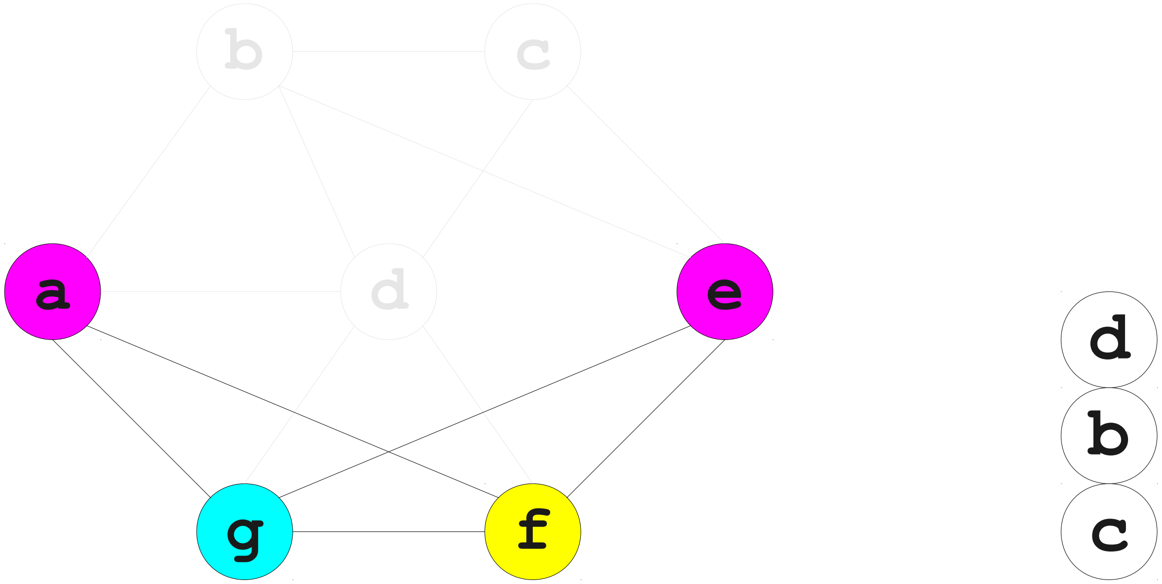
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

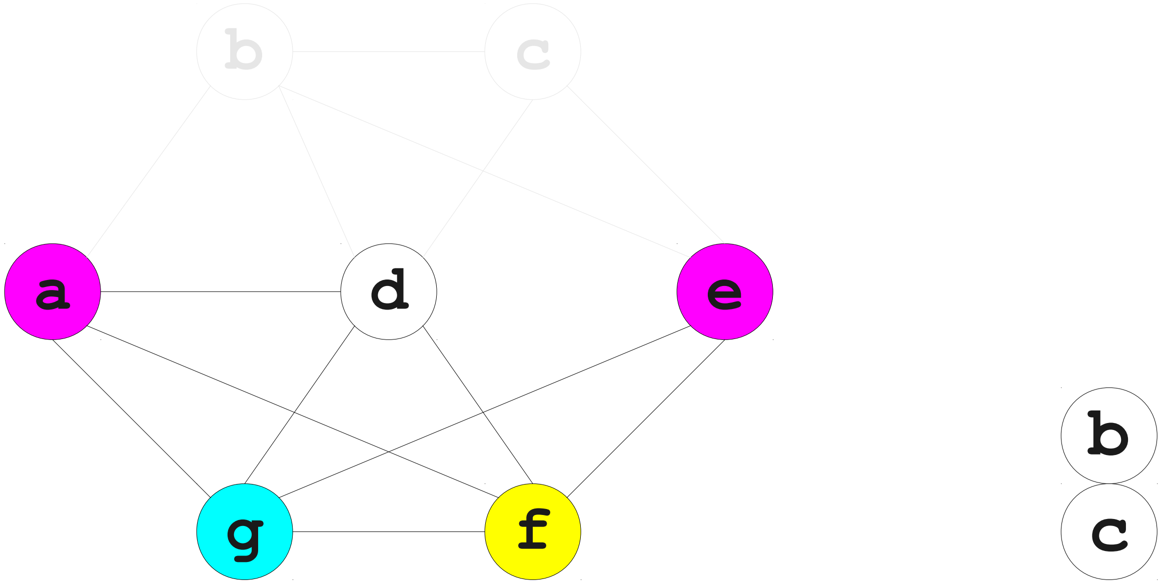
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

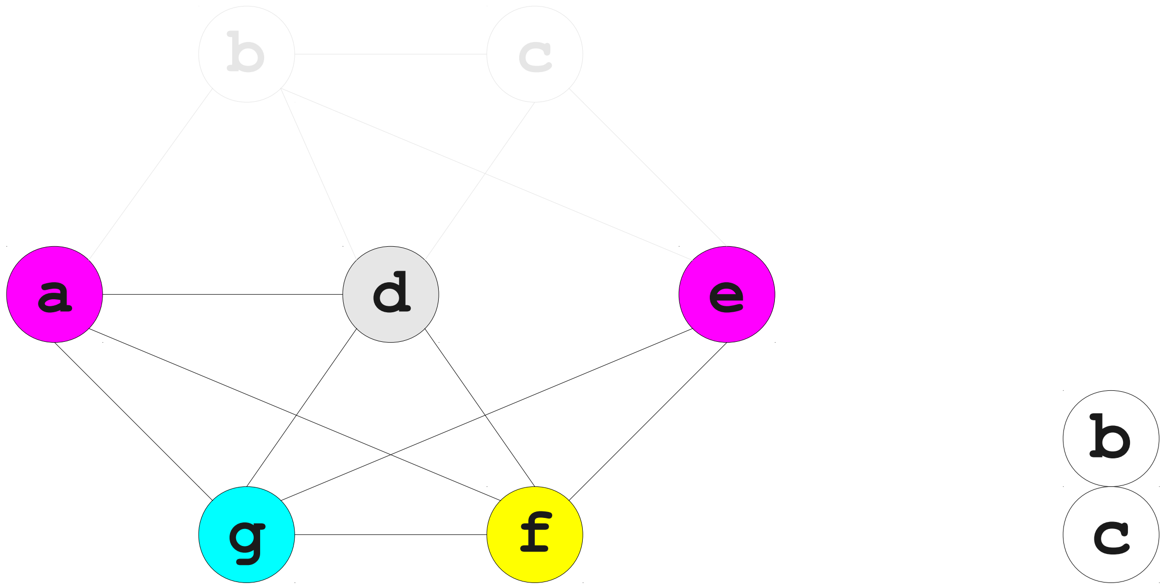
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

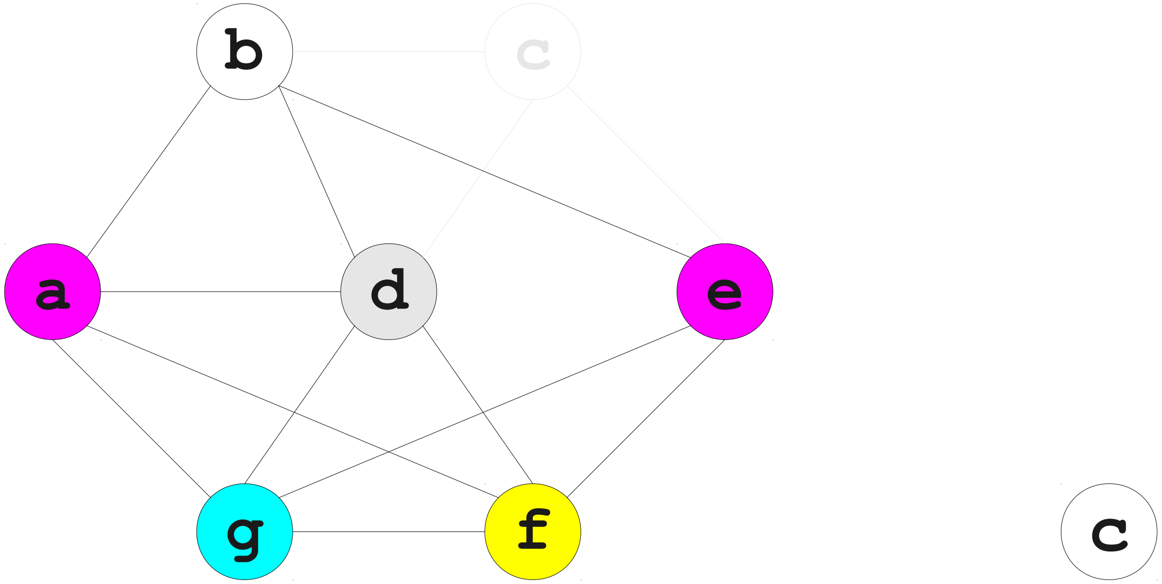
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

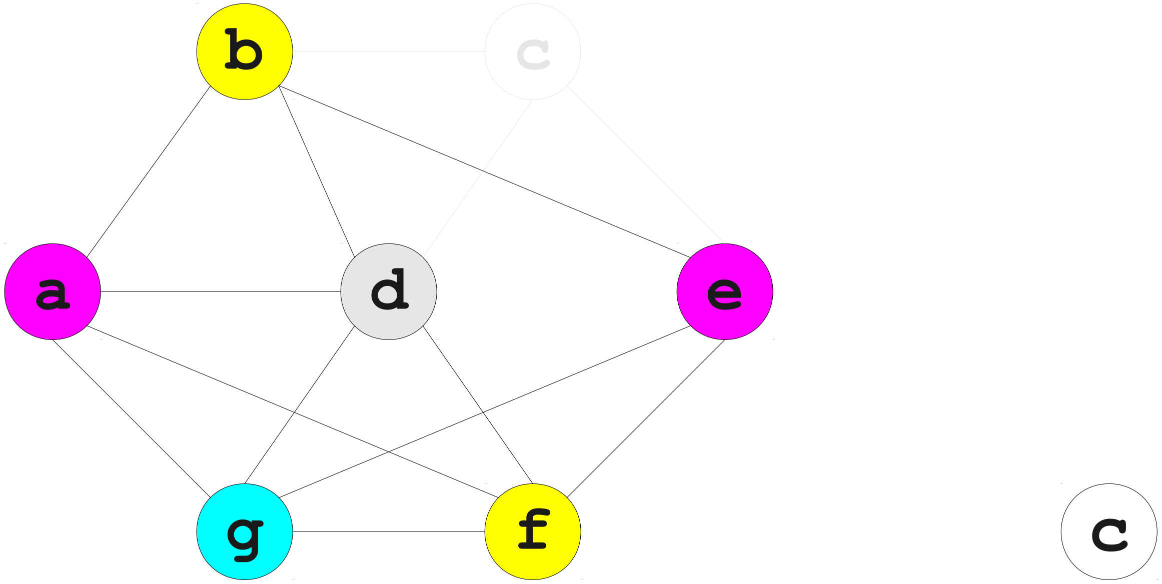
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

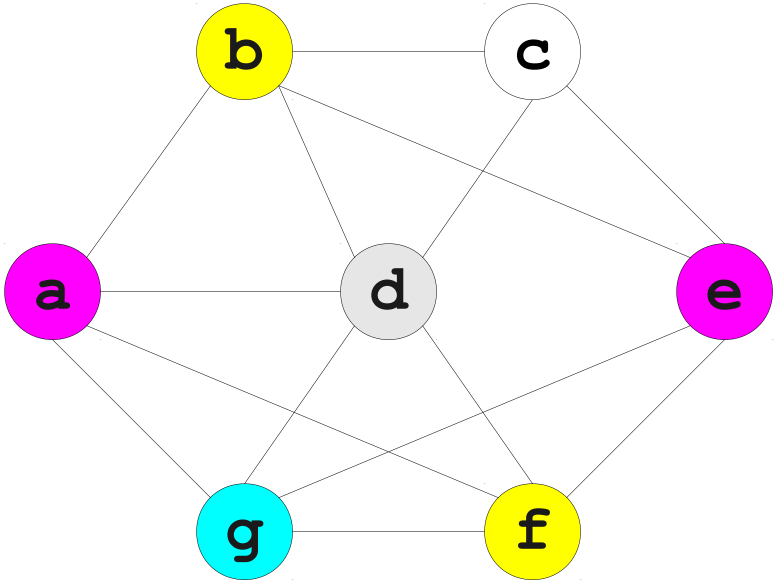
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

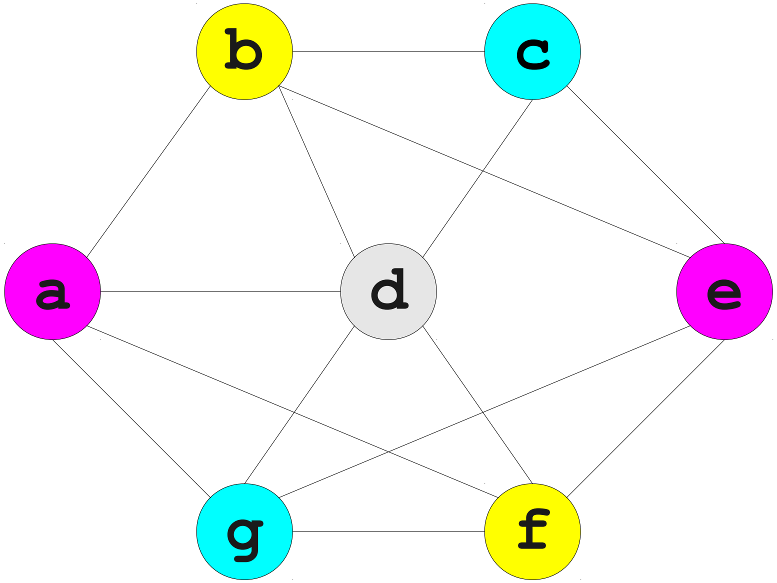
Chaitin's Algorithm



Registers

R_0	R_1	R_2	R_3

Chaitin's Algorithm



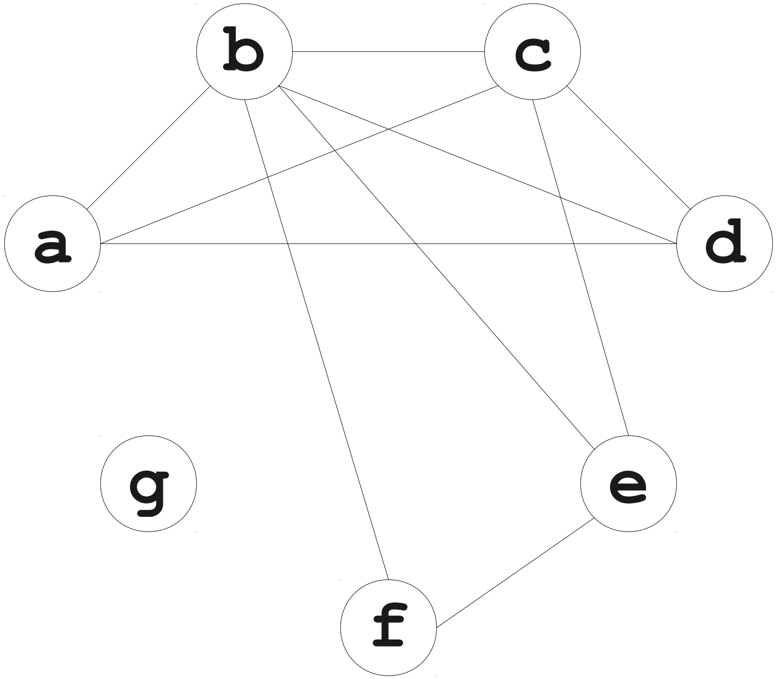
Registers

R_0	R_1	R_2	R_3

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.

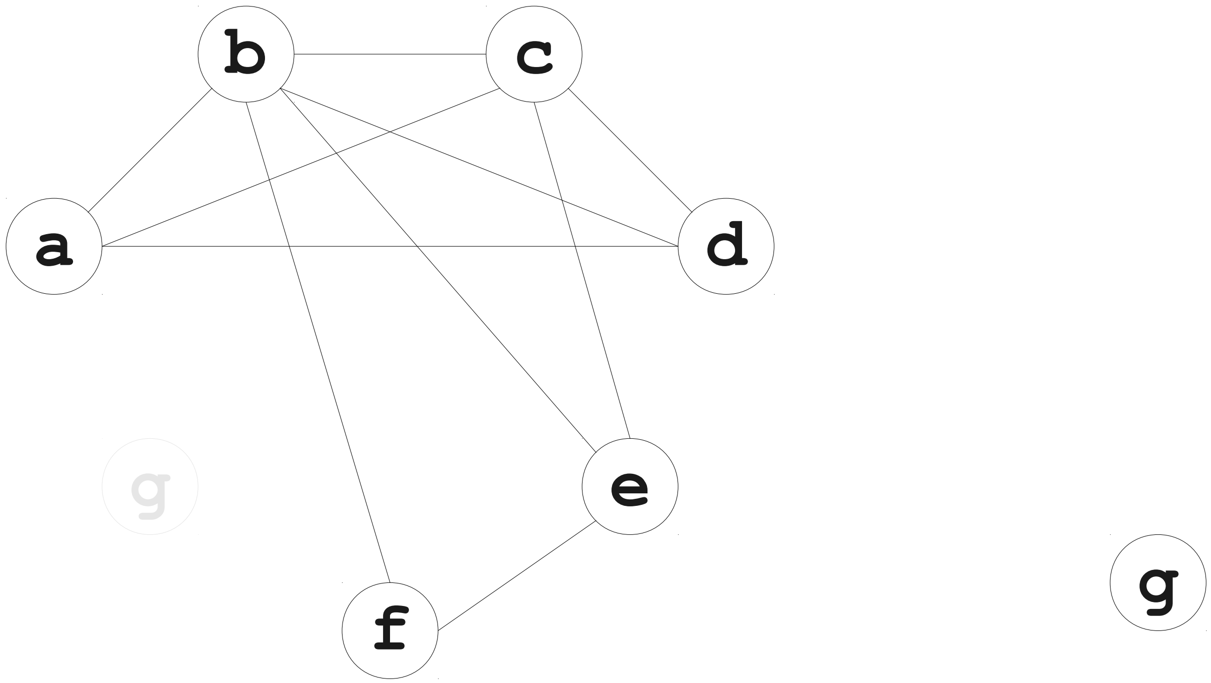
Chaitin's Algorithm Reloaded



Registers

R_0 R_1 R_2

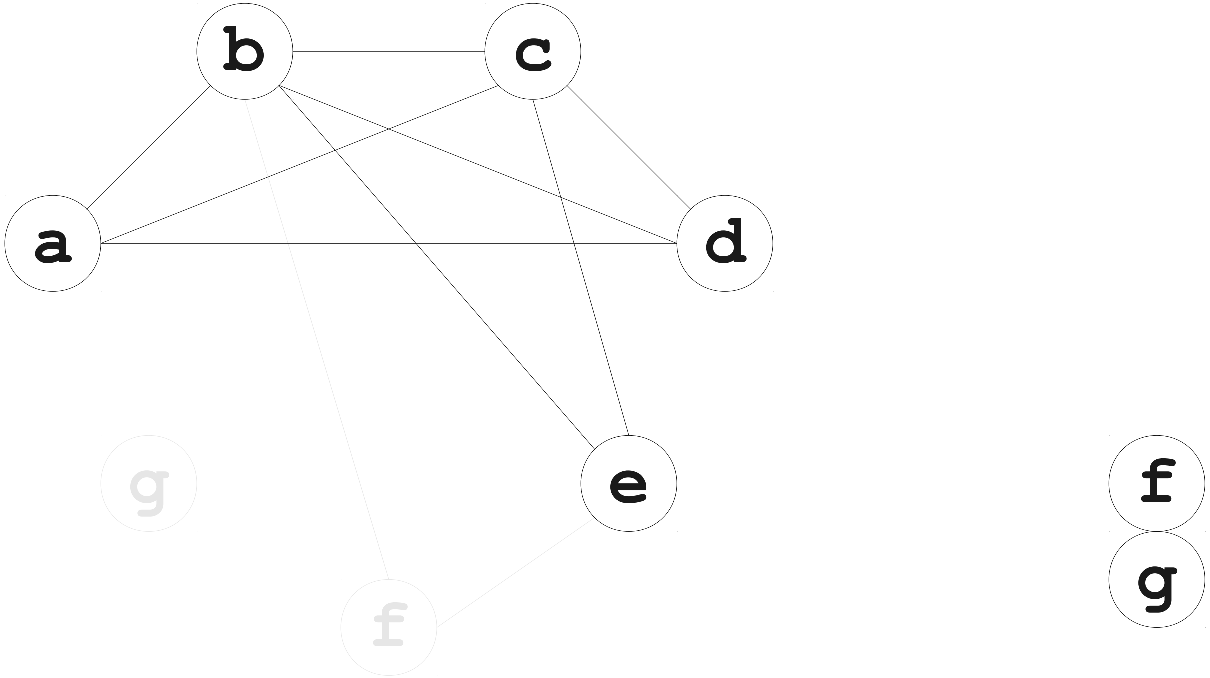
Chaitin's Algorithm Reloaded



Registers

R_0 **R_1** **R_2**

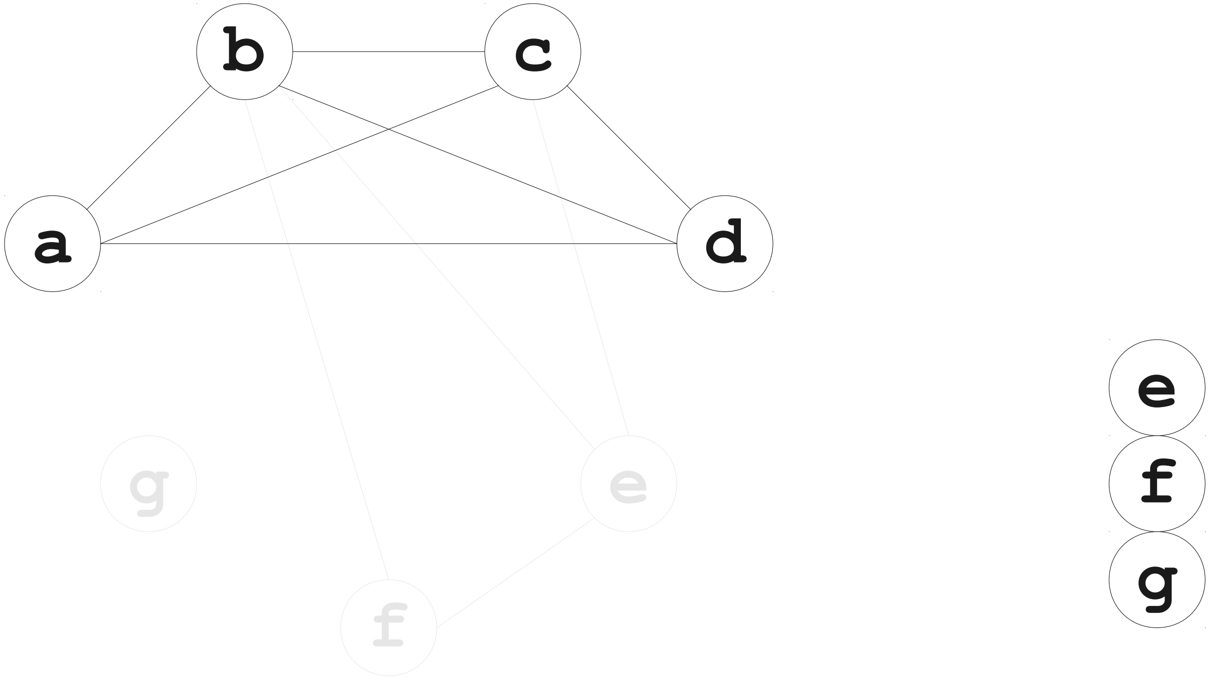
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

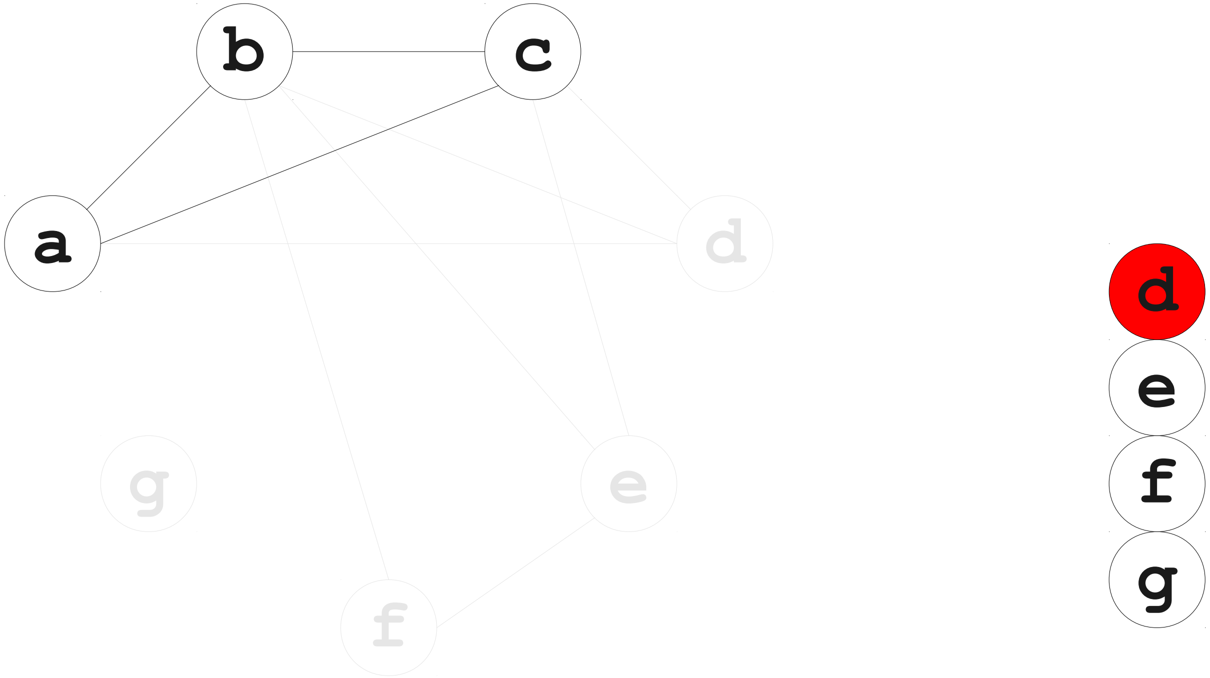
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

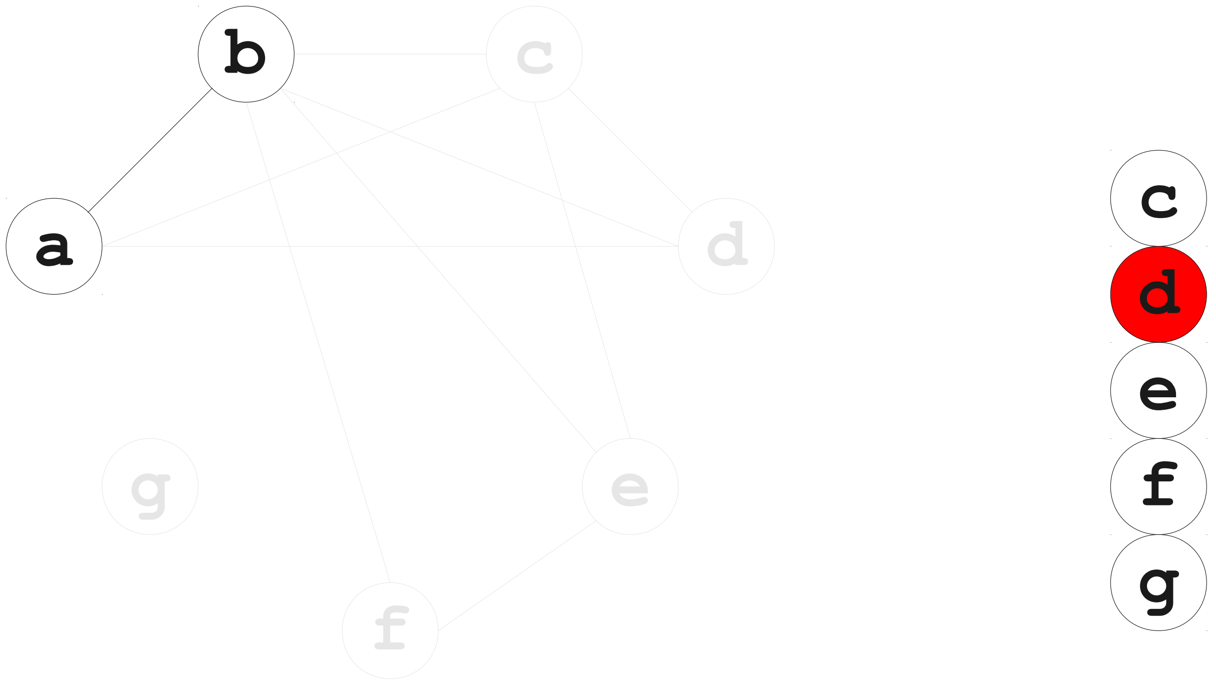
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

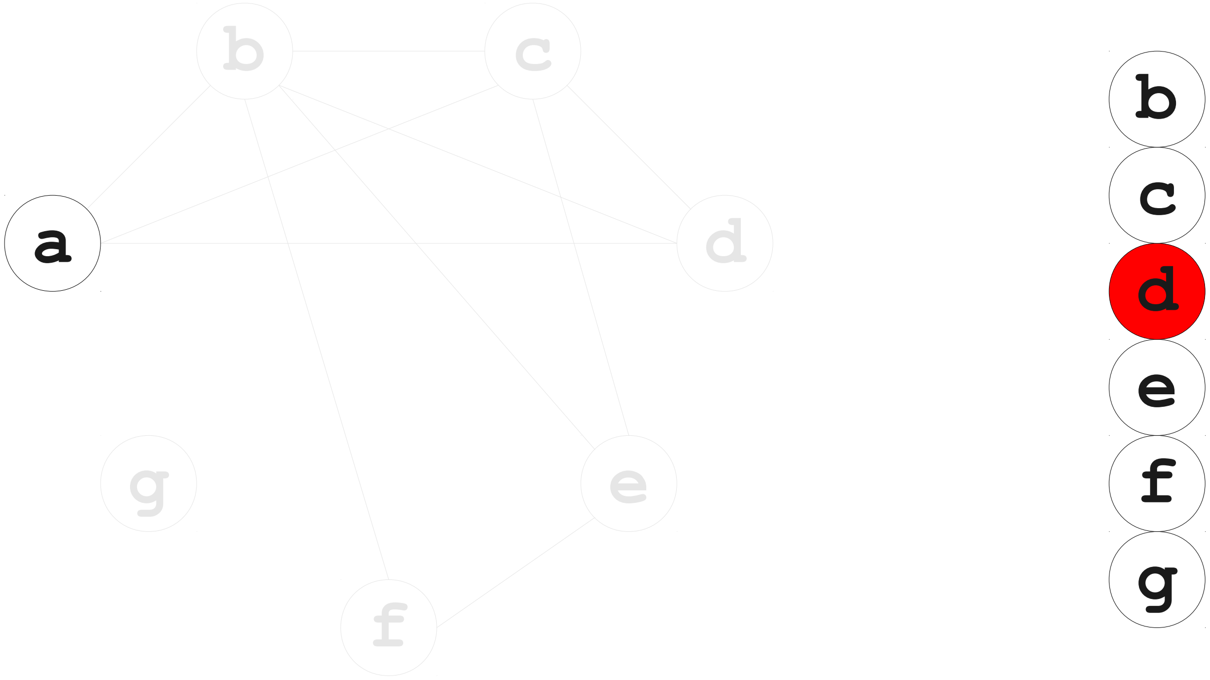
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

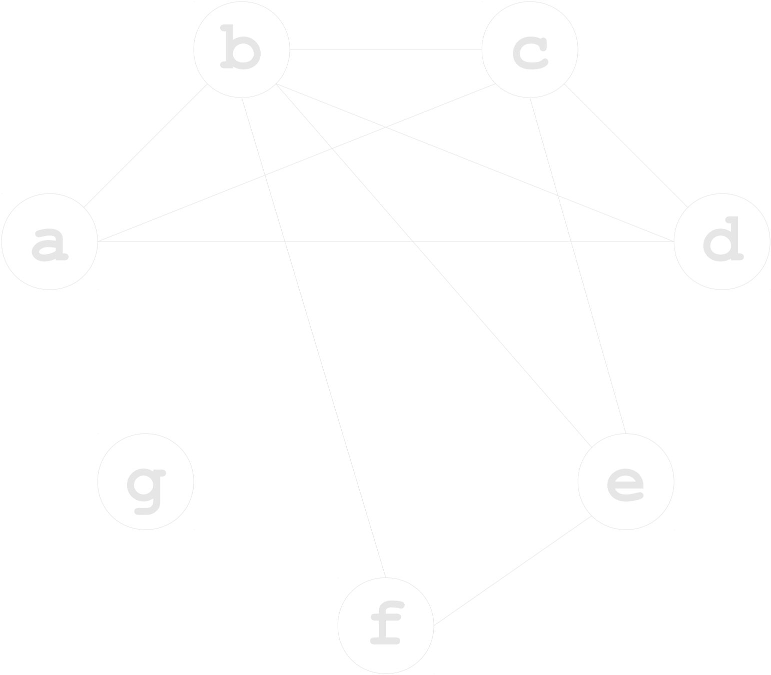
Chaitin's Algorithm Reloaded

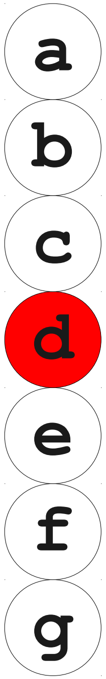


Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded

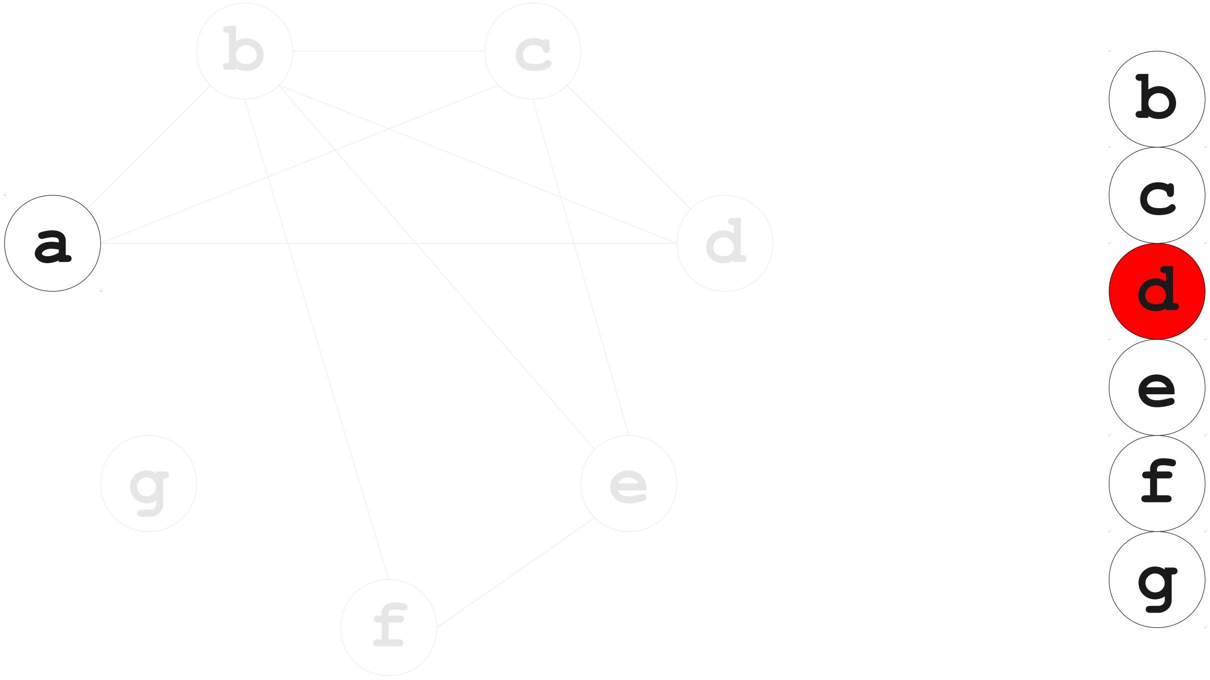




Registers

R_0	R_1	R_2

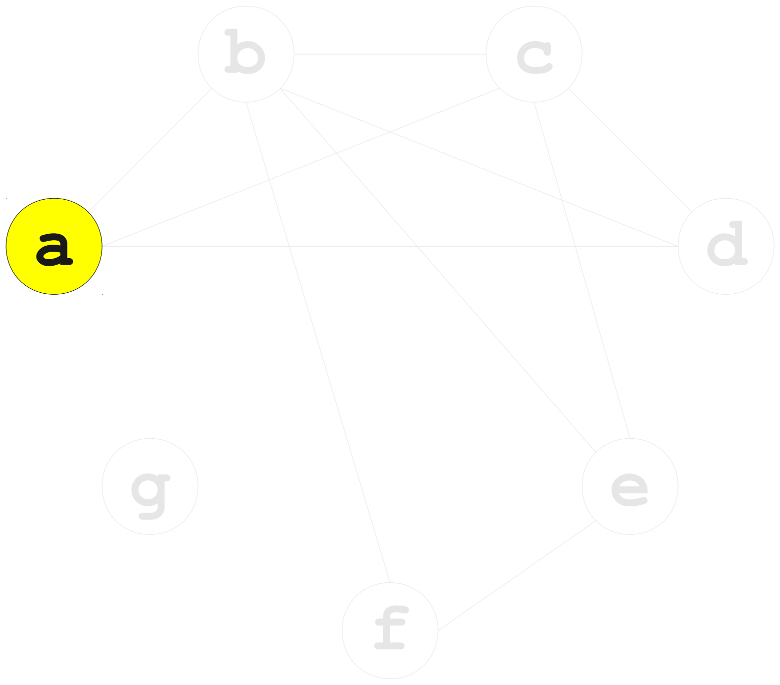
Chaitin's Algorithm Reloaded

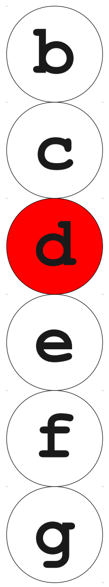


Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded

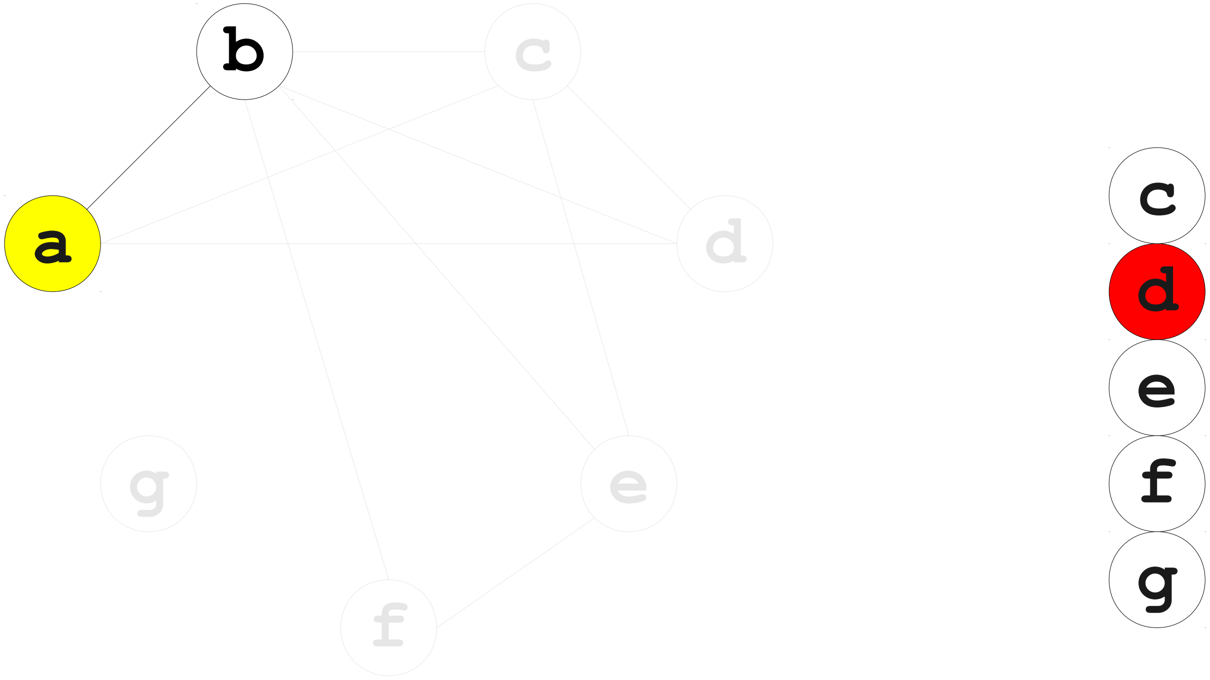




Registers

R_0	R_1	R_2

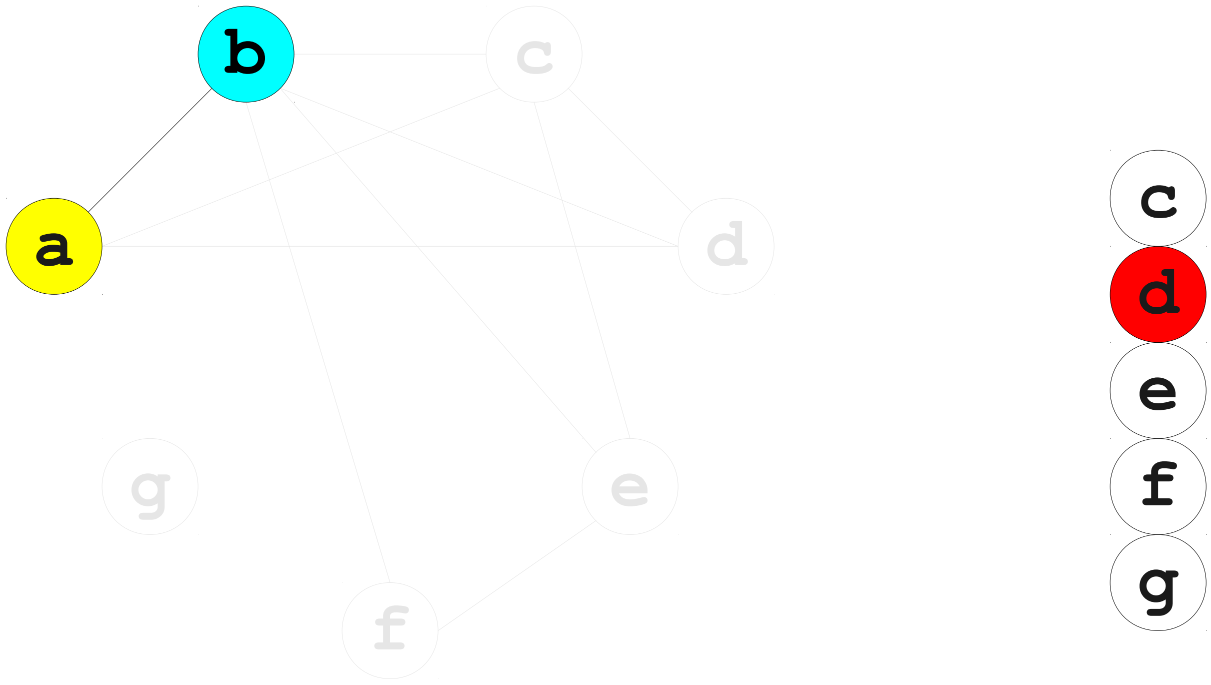
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

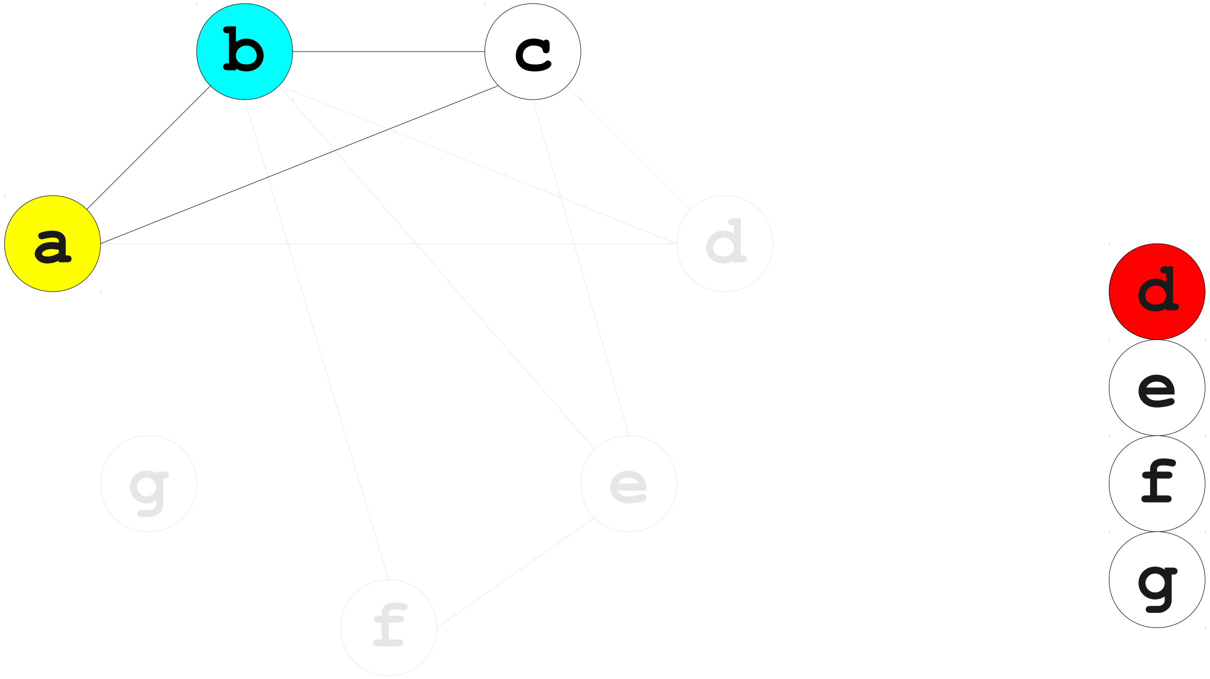
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

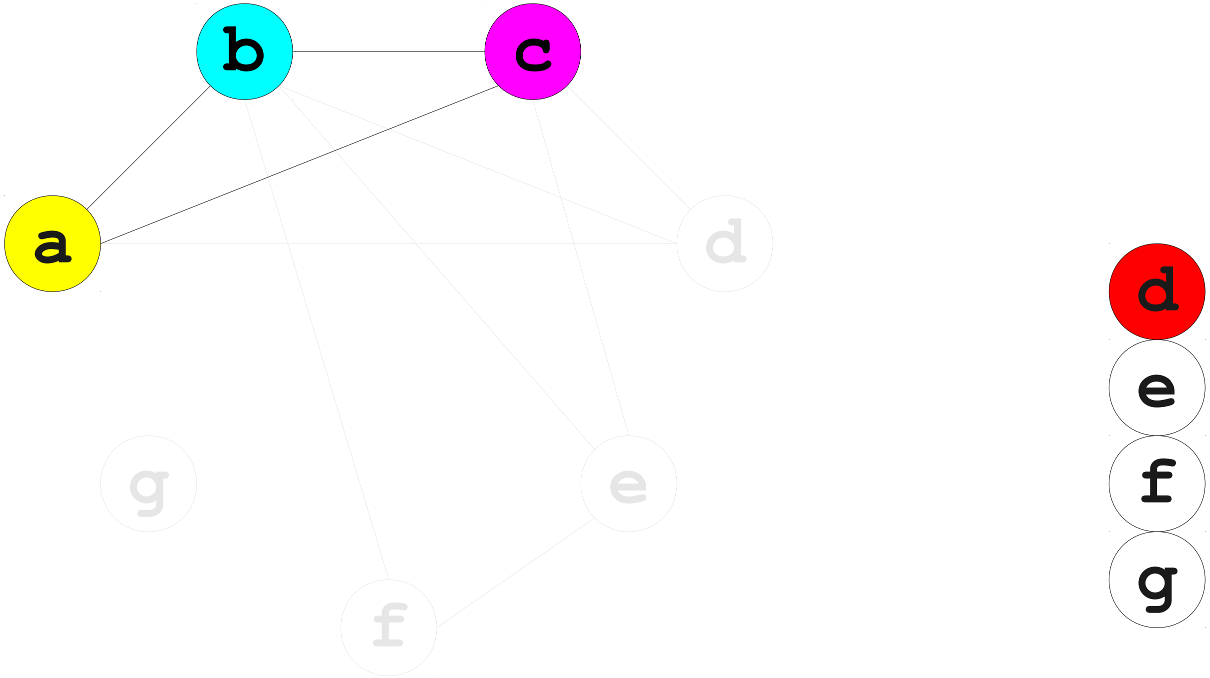
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

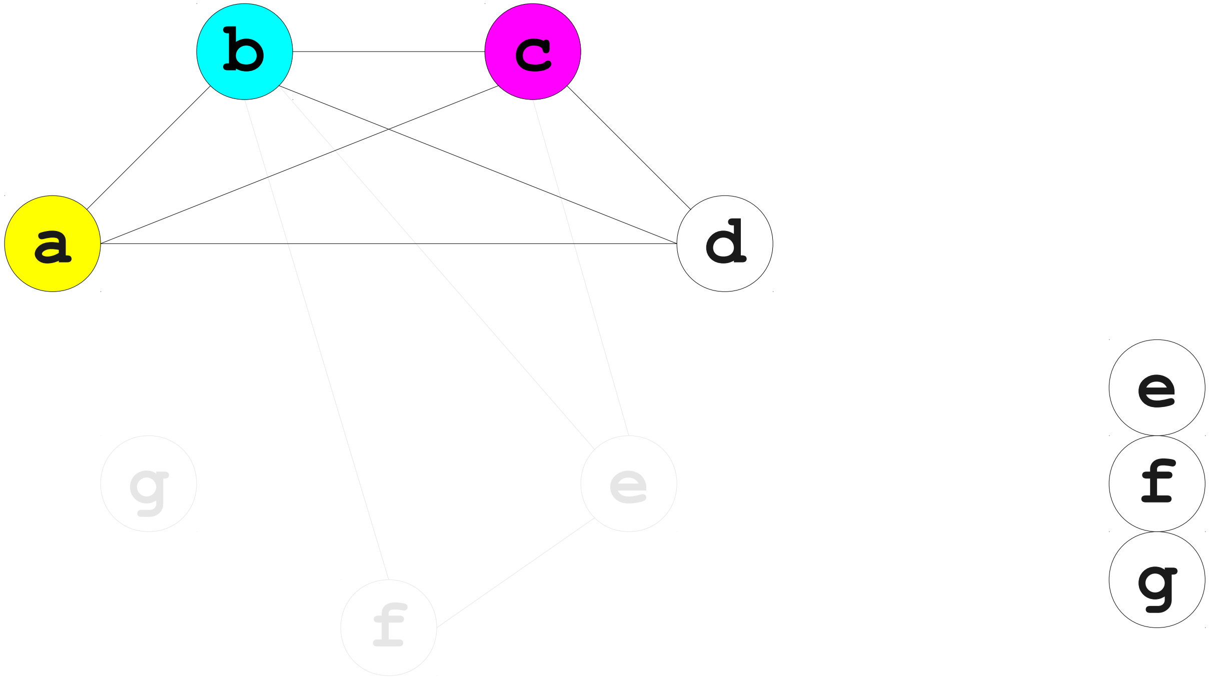
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

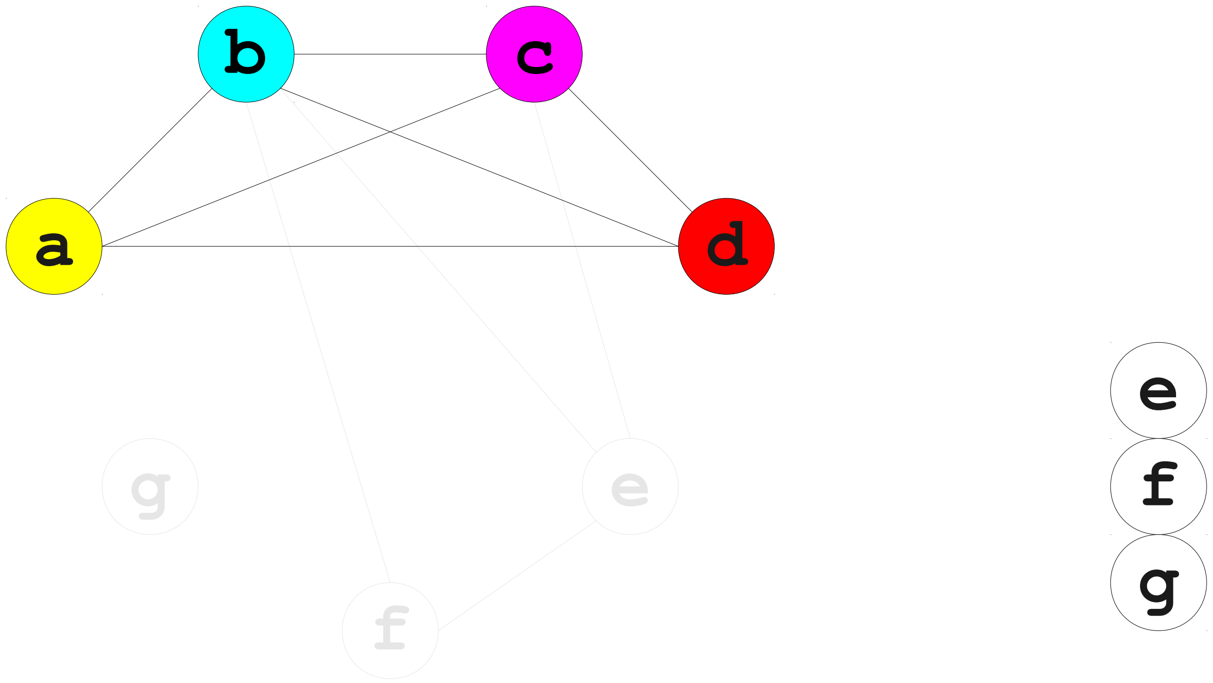
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

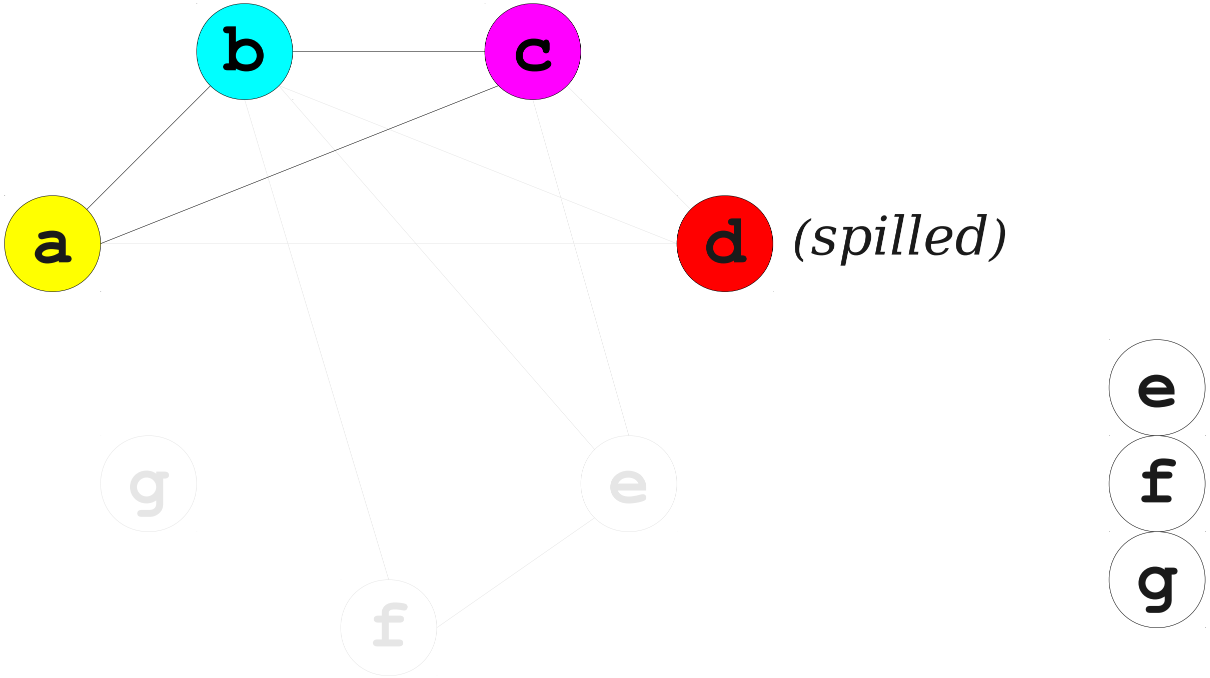
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

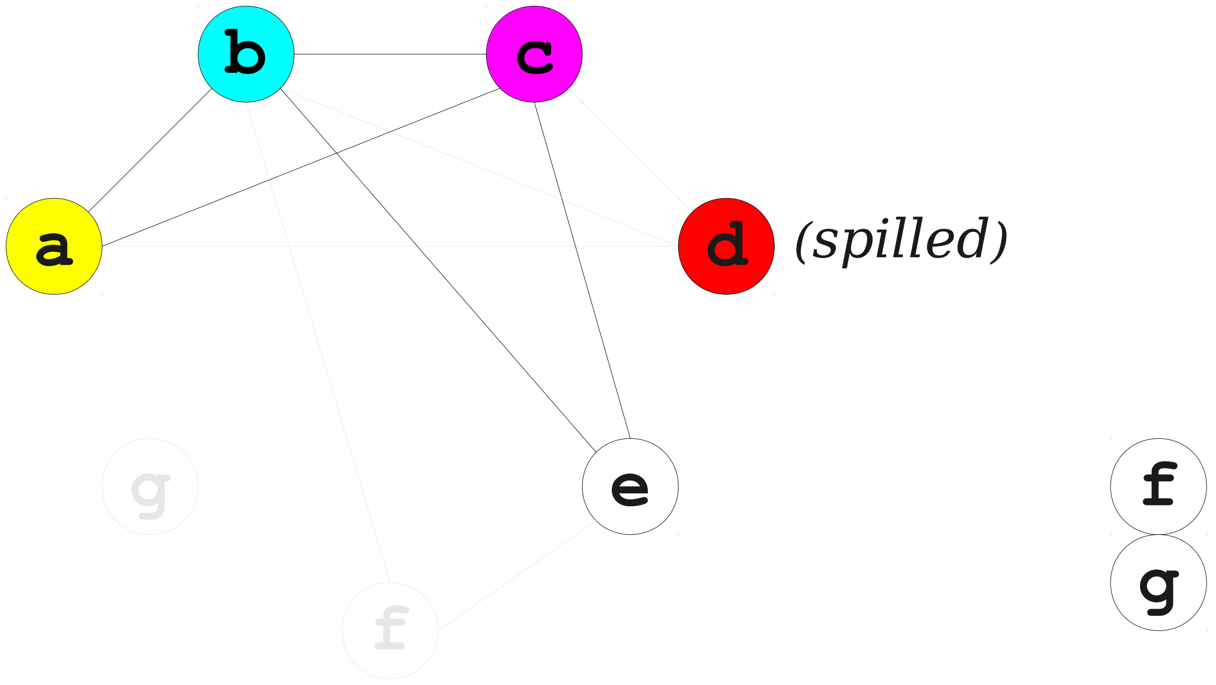
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

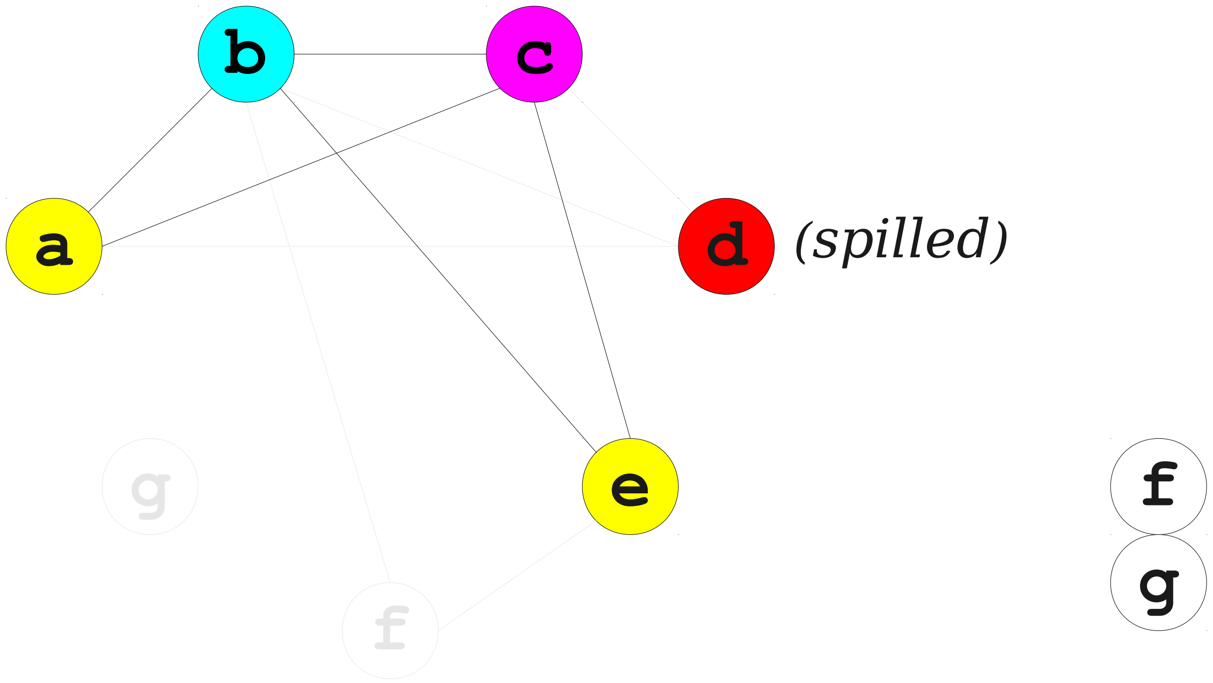
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

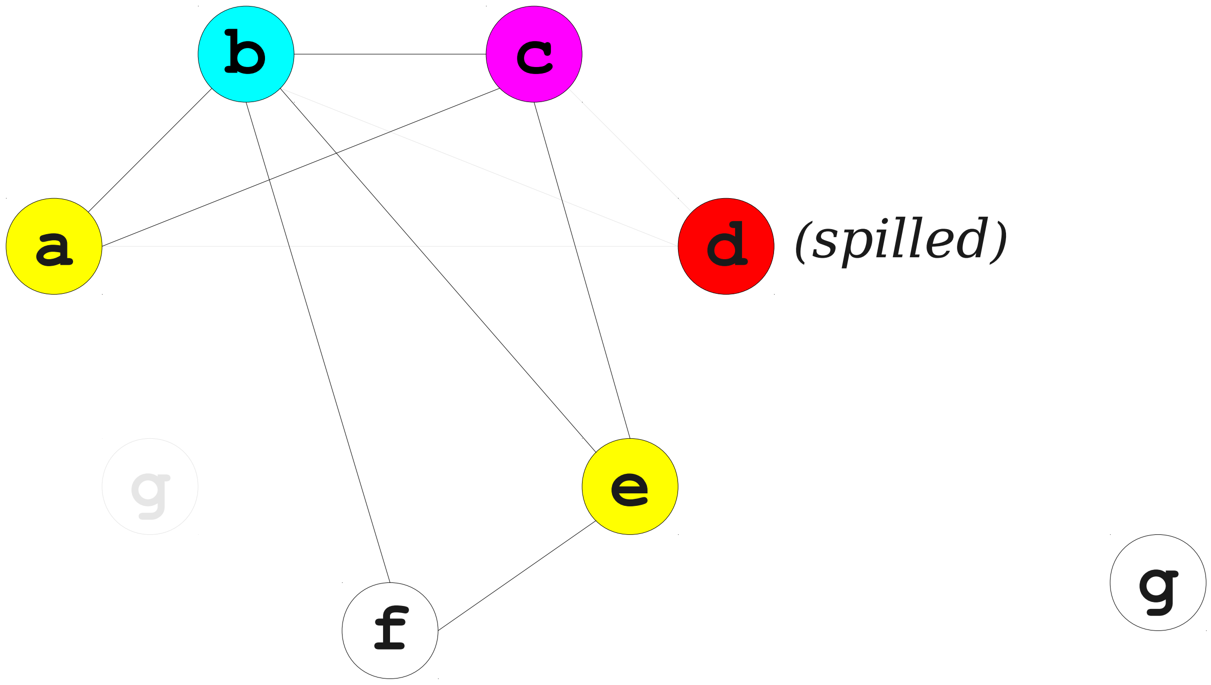
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

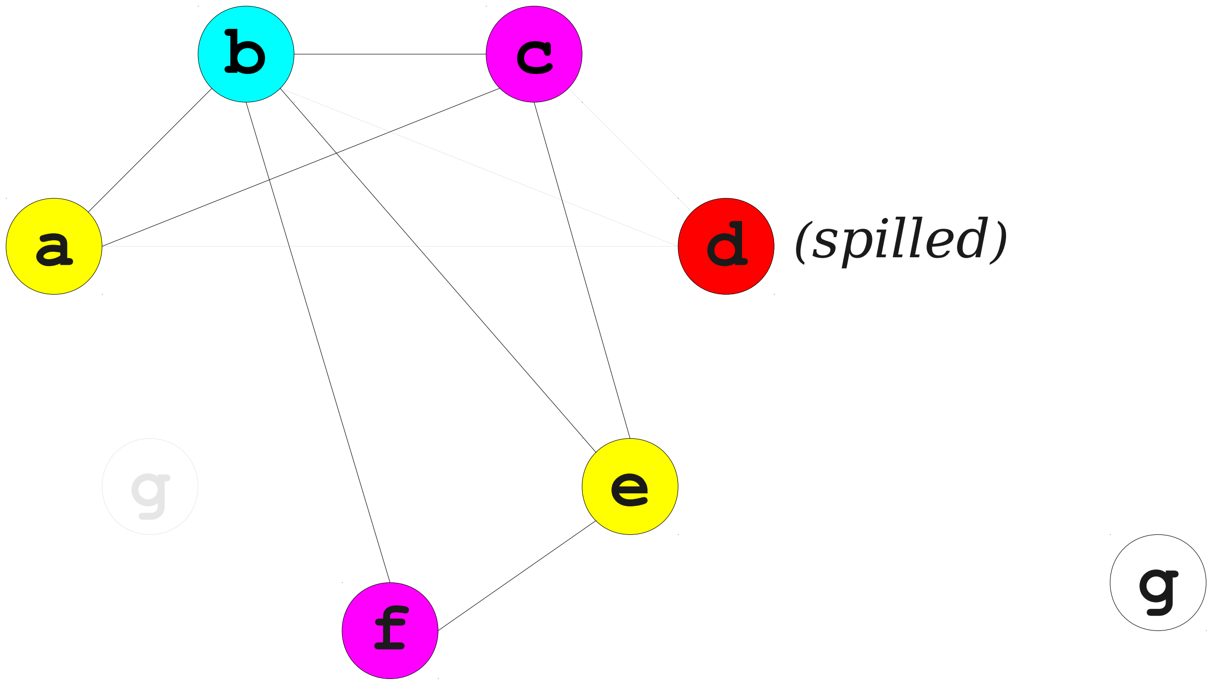
Chaitin's Algorithm Reloaded



Registers

R_0	**R_1**	**R_2**

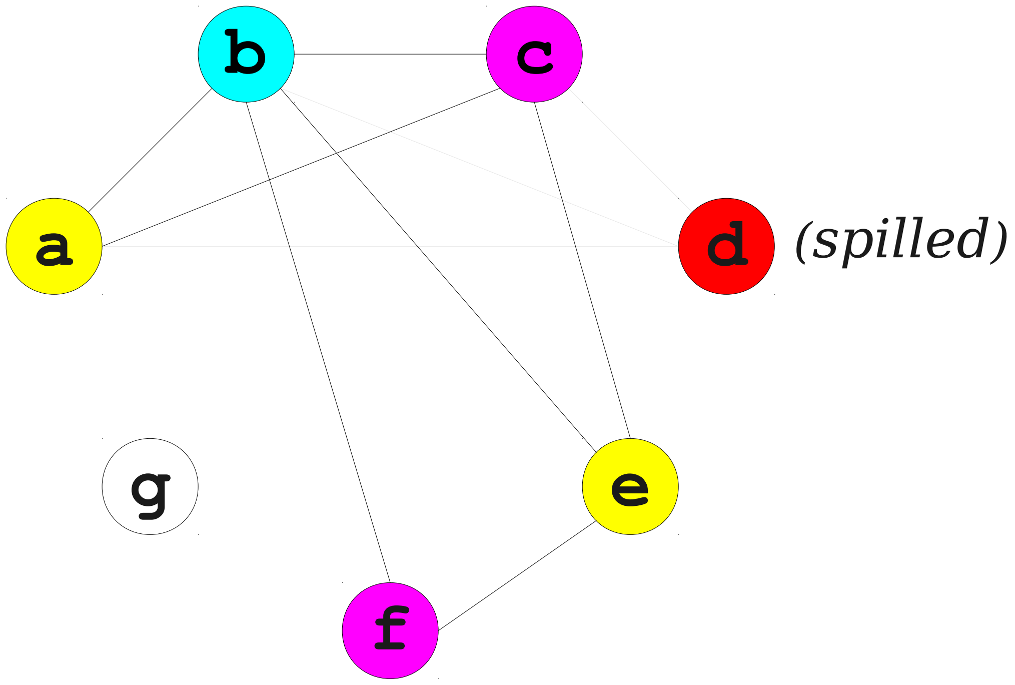
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

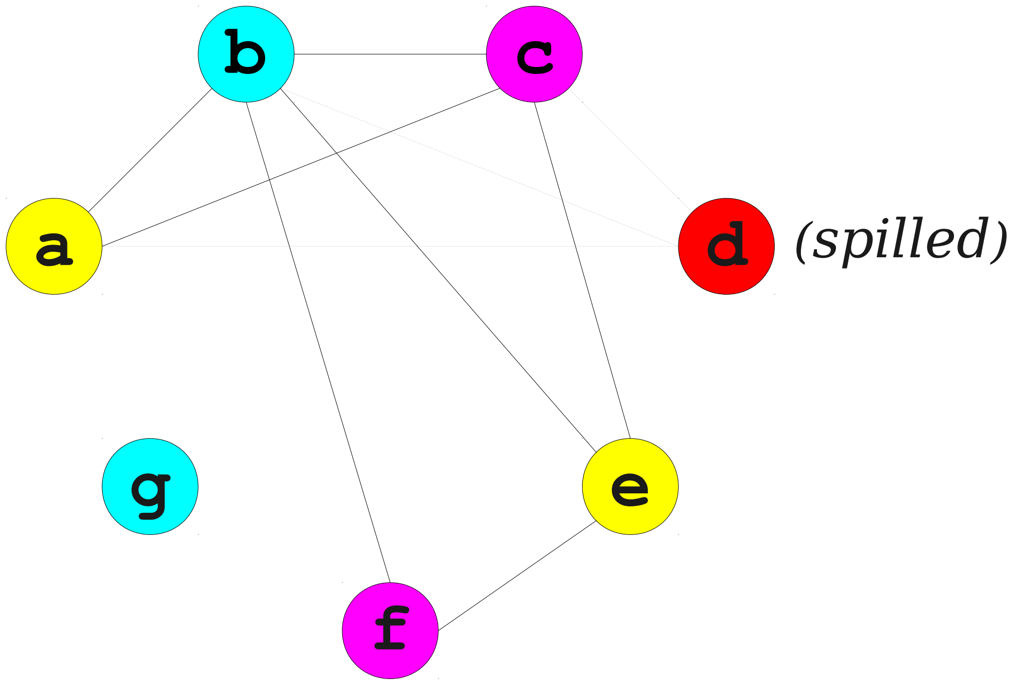
Chaitin's Algorithm Reloaded



Registers

R_0	R_1	R_2

Chaitin's Algorithm Reloaded

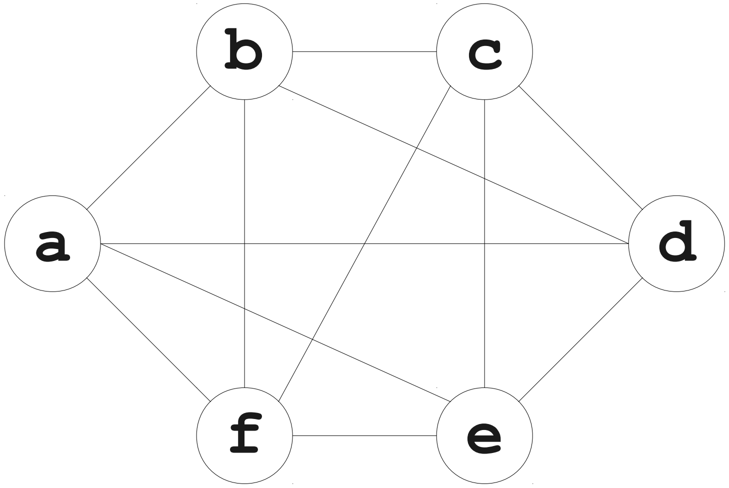


Registers

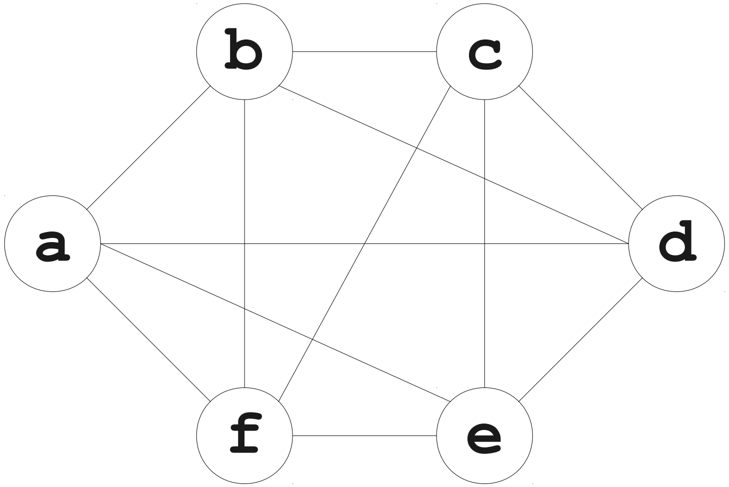
R_0	R_1	R_2

Another Example

Another Example



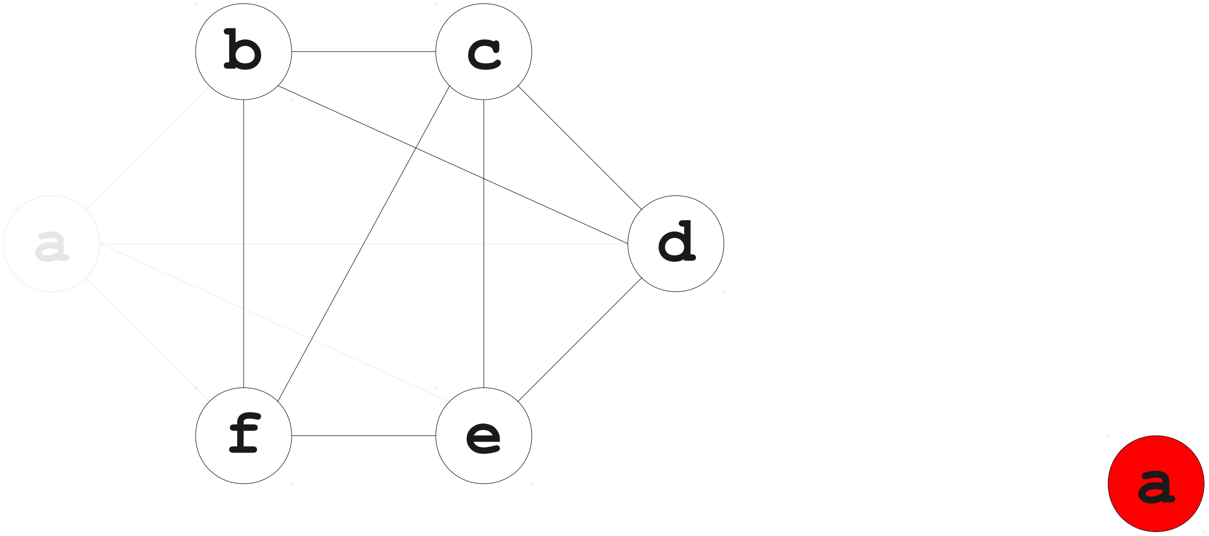
Another Example



Registers

R_0	R_1	R_2

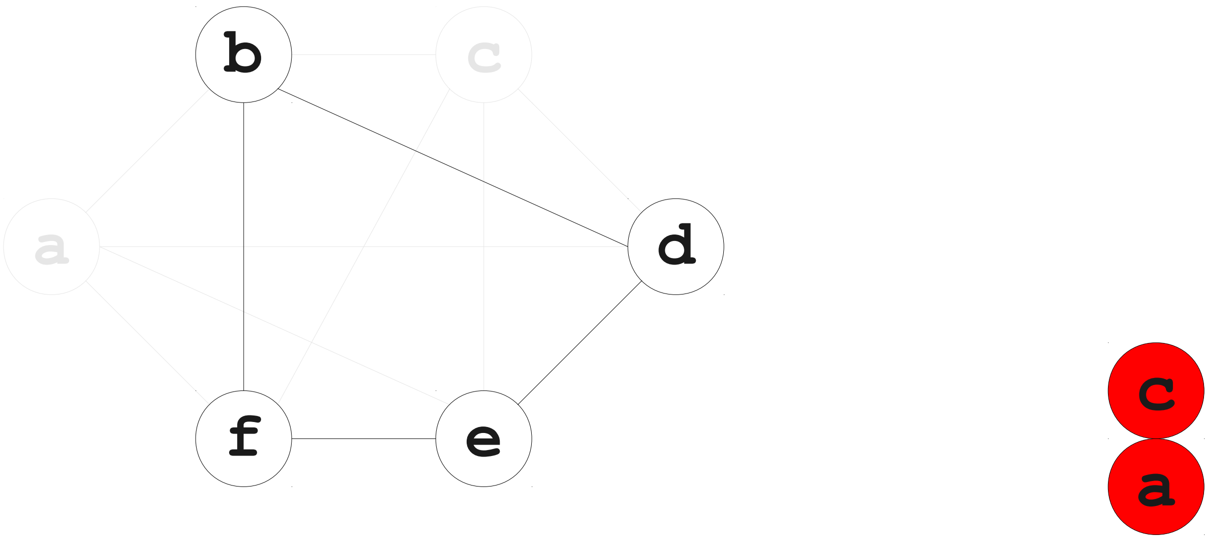
Another Example



Registers

R_0	R_1	R_2

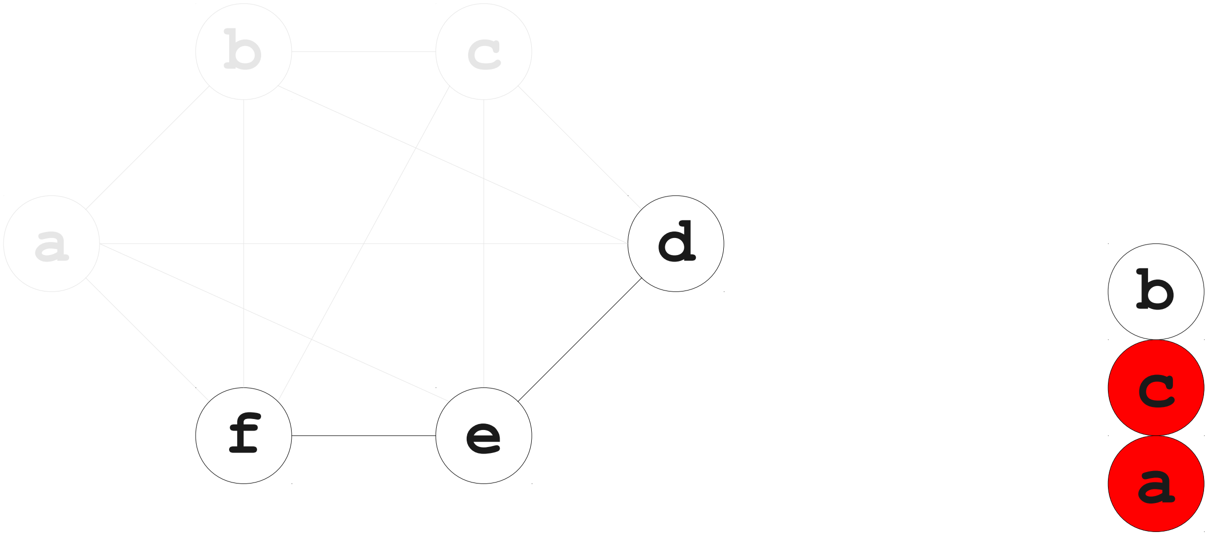
Another Example



Registers

R_0	R_1	R_2

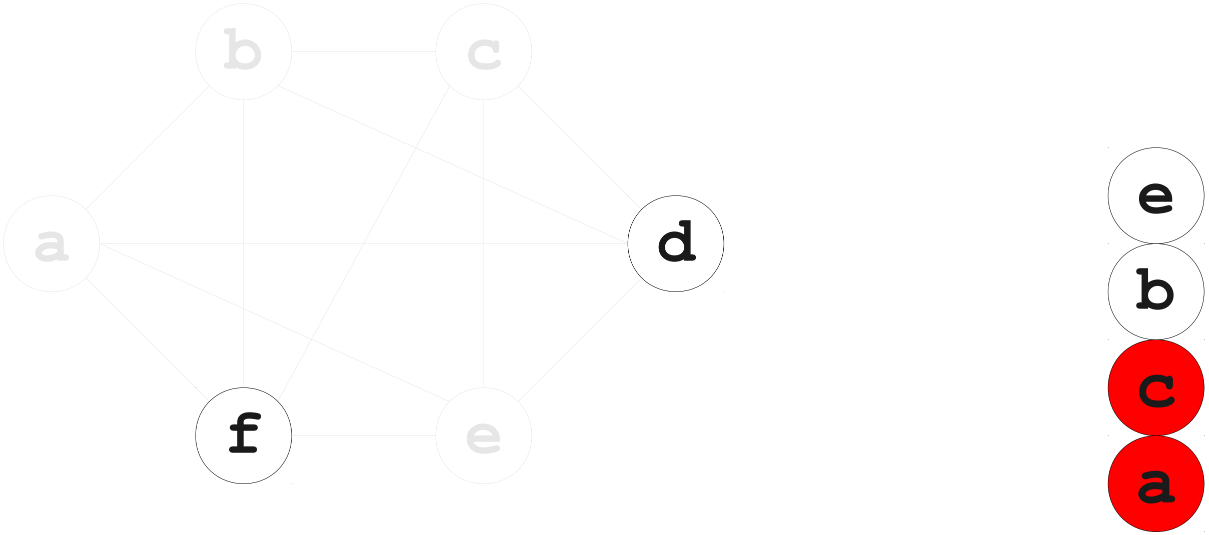
Another Example



Registers

R_0	R_1	R_2

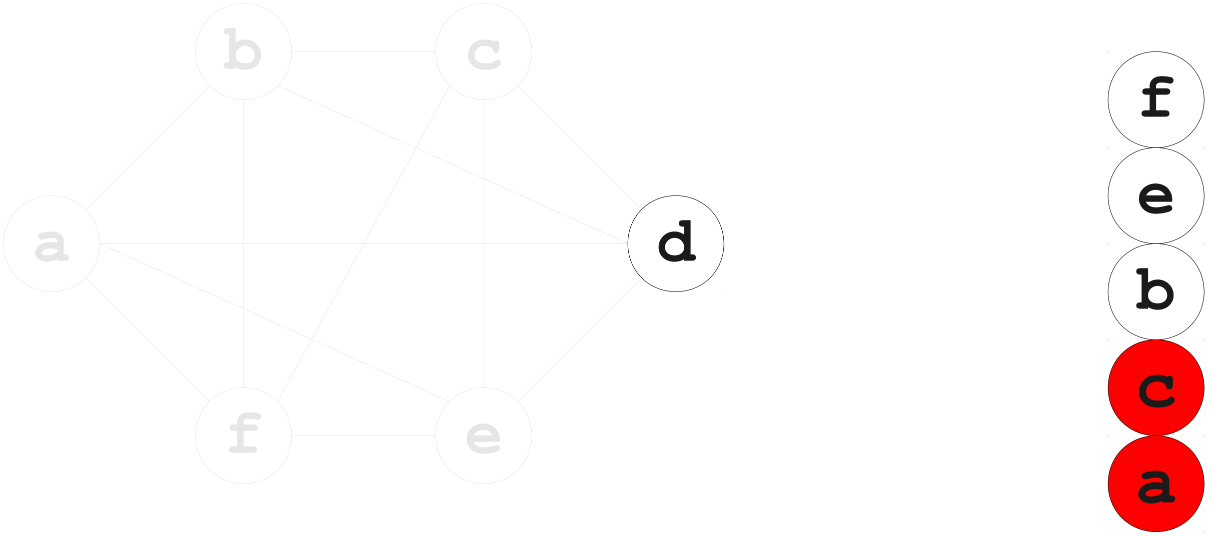
Another Example



Registers

R_0	R_1	R_2

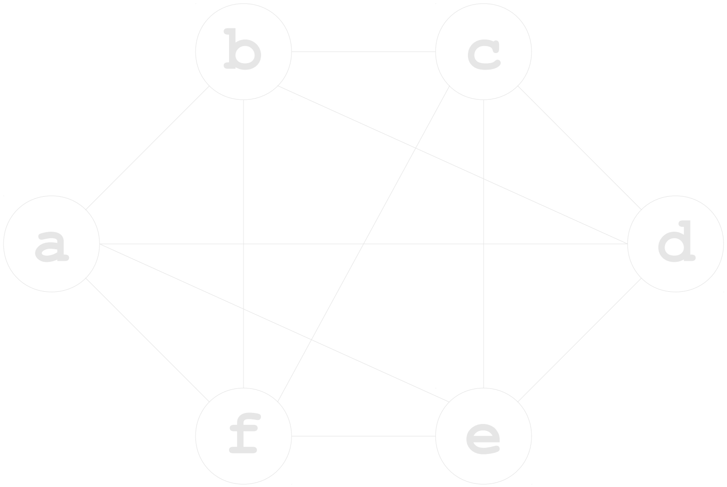
Another Example

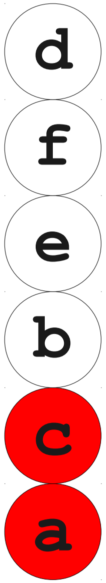


Registers

R_0	R_1	R_2

Another Example

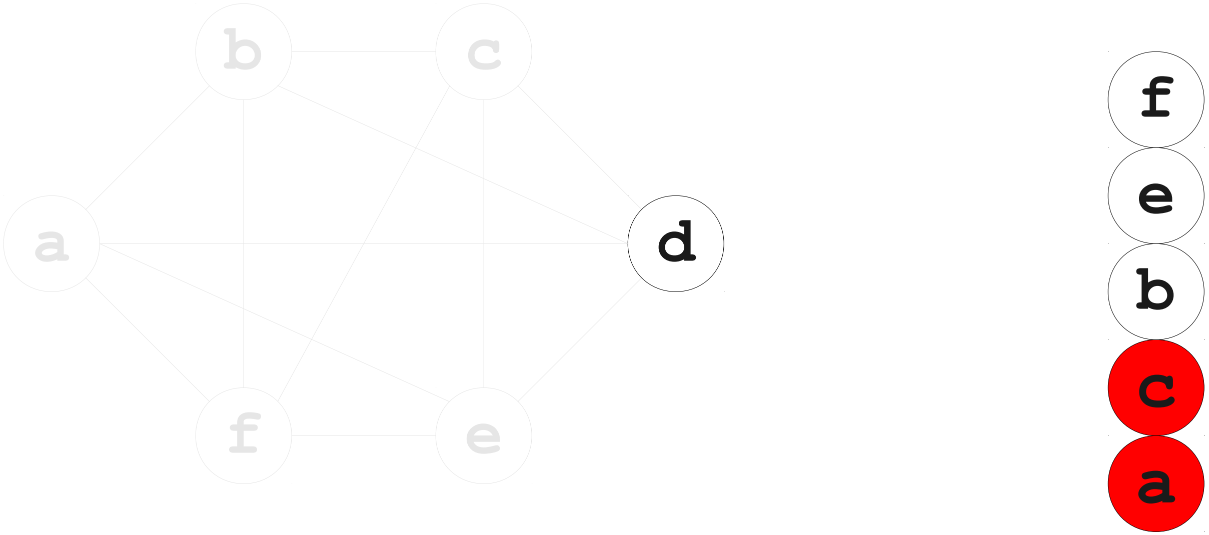




Registers

R_0	R_1	R_2

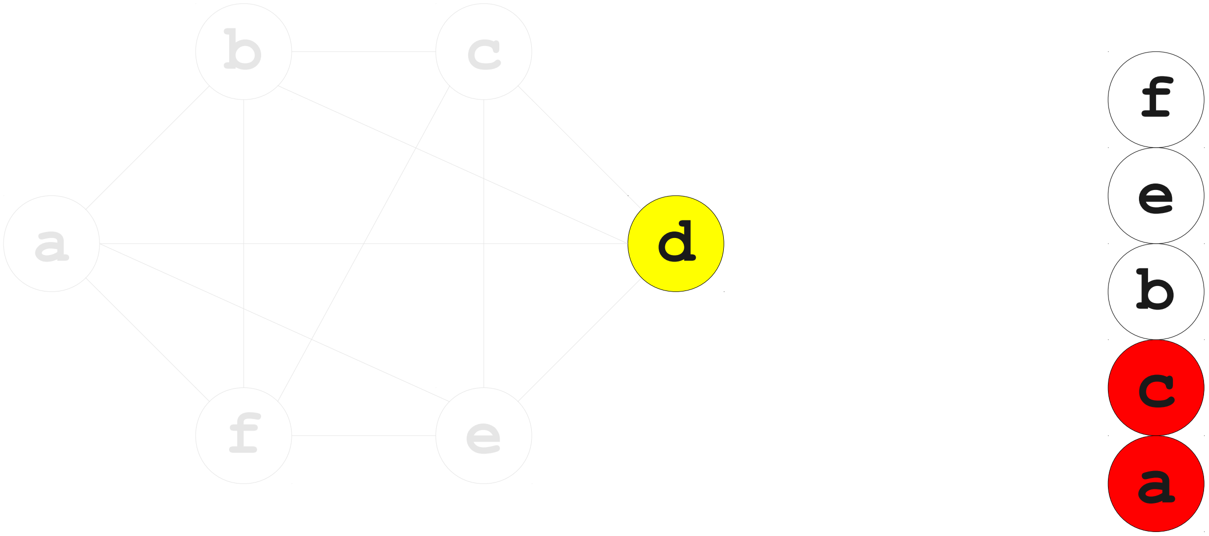
Another Example



Registers

R_0	R_1	R_2

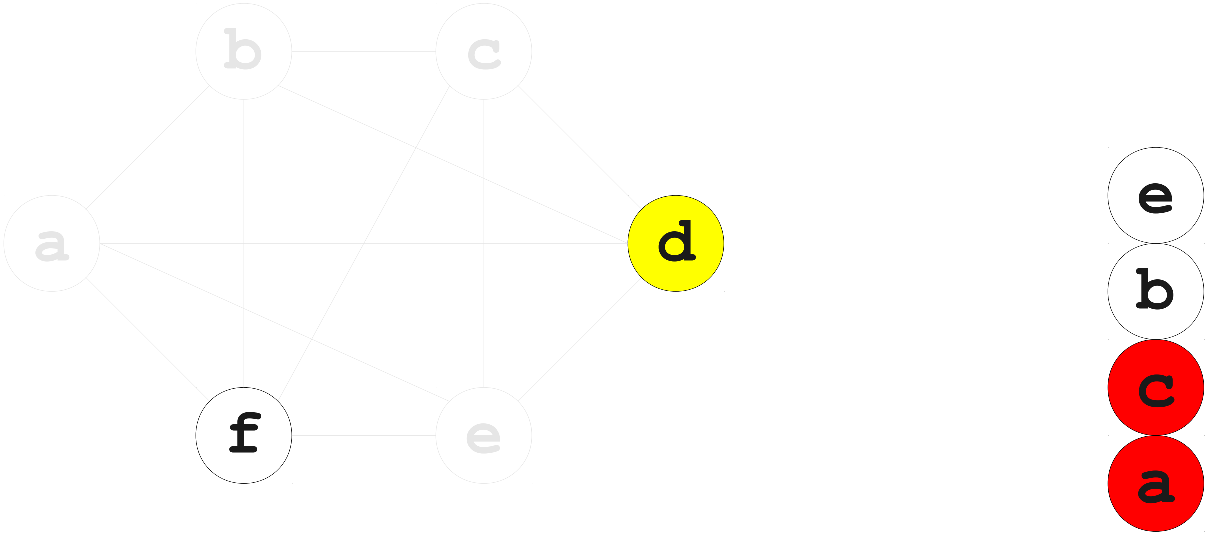
Another Example



Registers

R_0	R_1	R_2

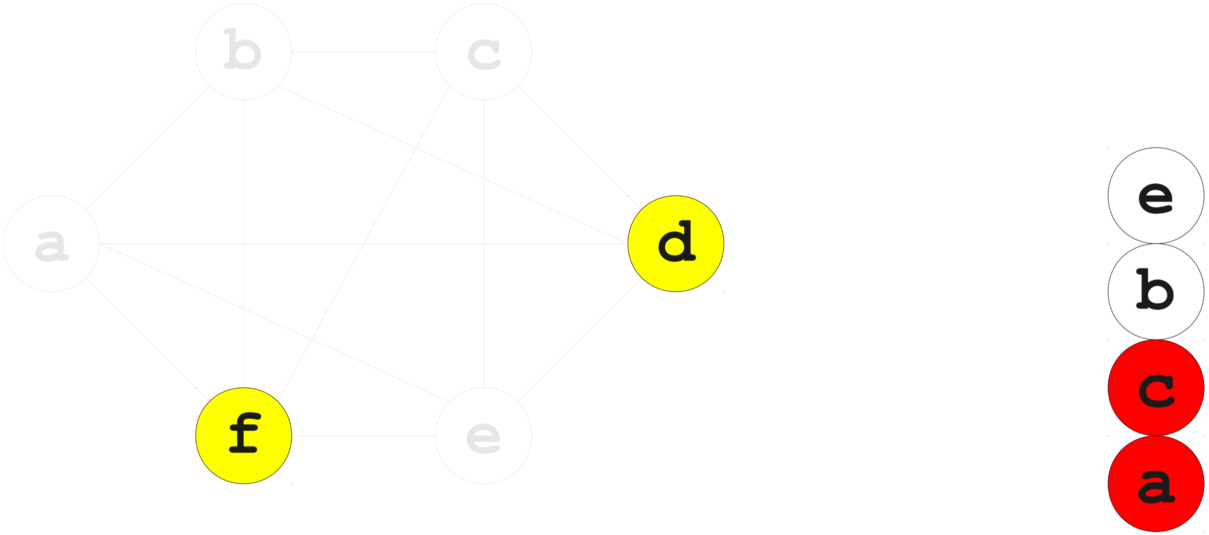
Another Example



Registers

R_0	R_1	R_2

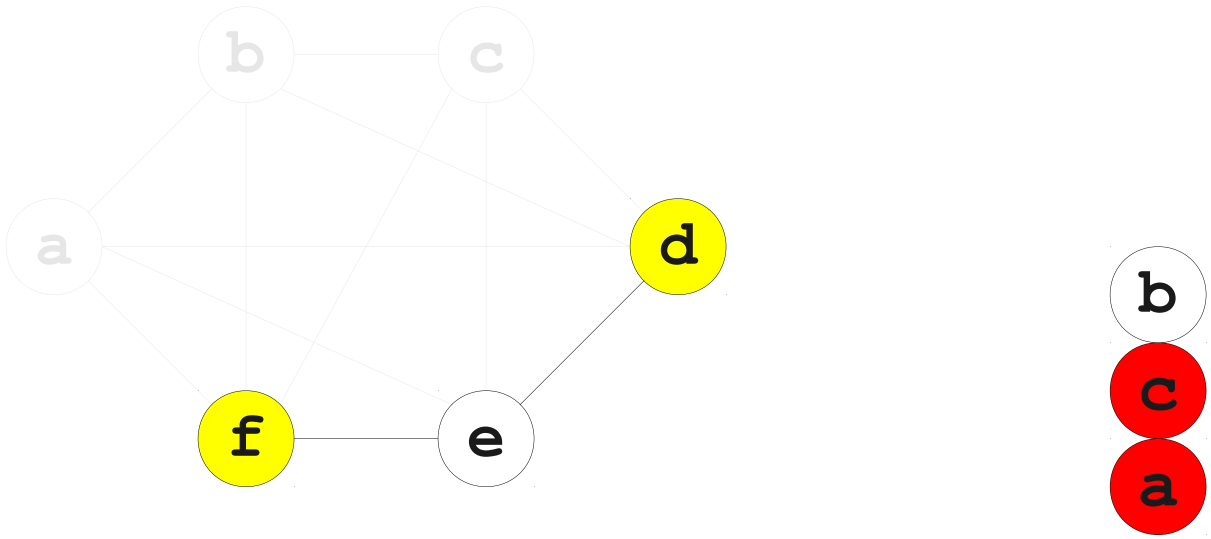
Another Example



Registers

R$_0$	R$_1$	R$_2$

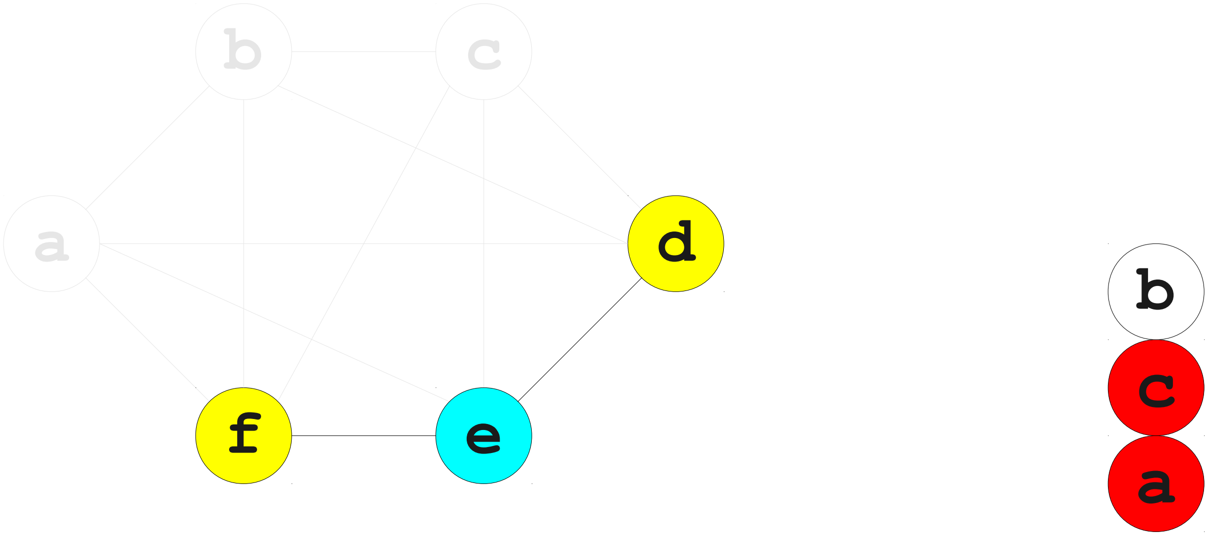
Another Example



Registers

R_0	R_1	R_2

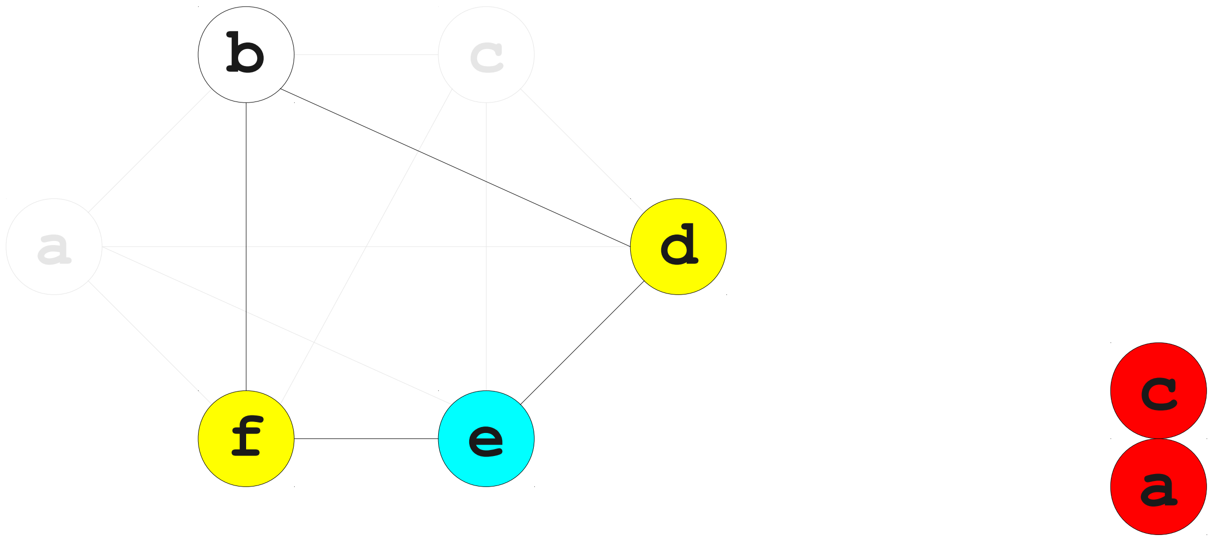
Another Example



Registers

R_0	R_1	R_2

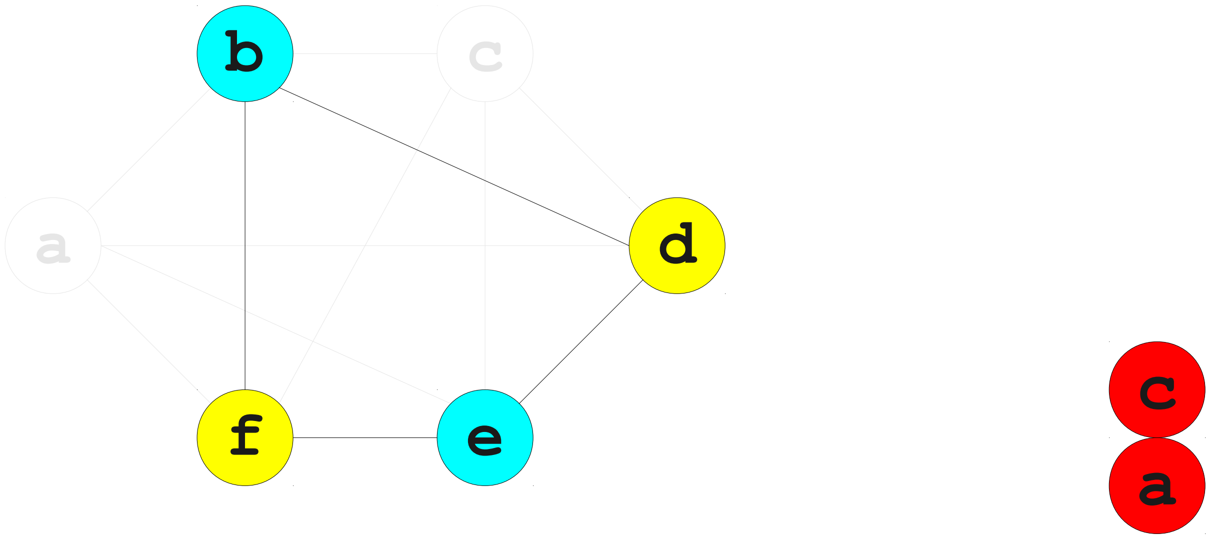
Another Example



Registers

R_0	R_1	R_2

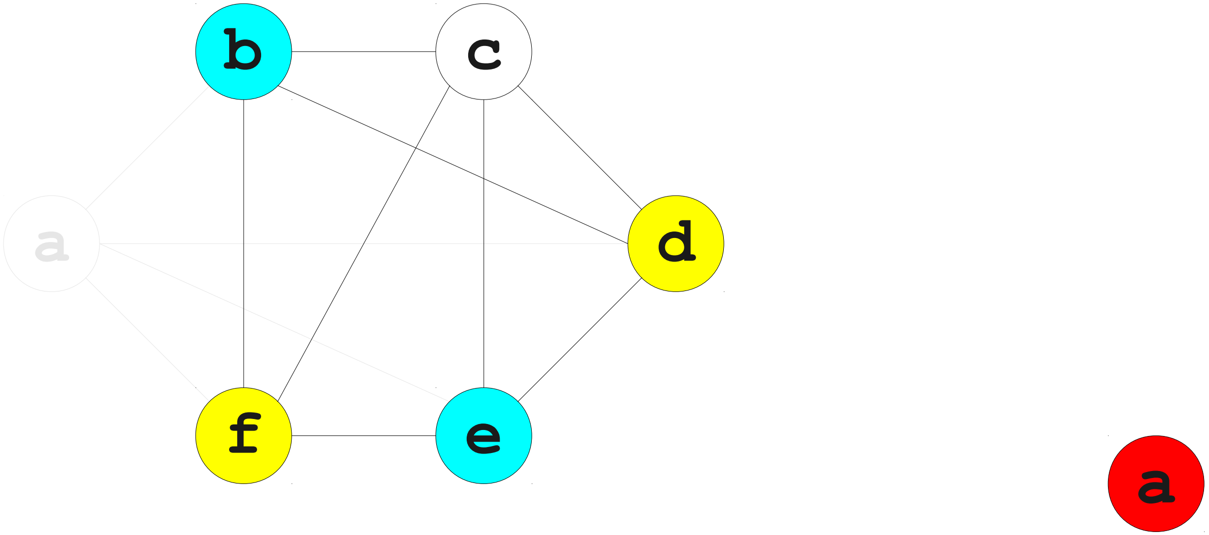
Another Example



Registers

R_0	R_1	R_2

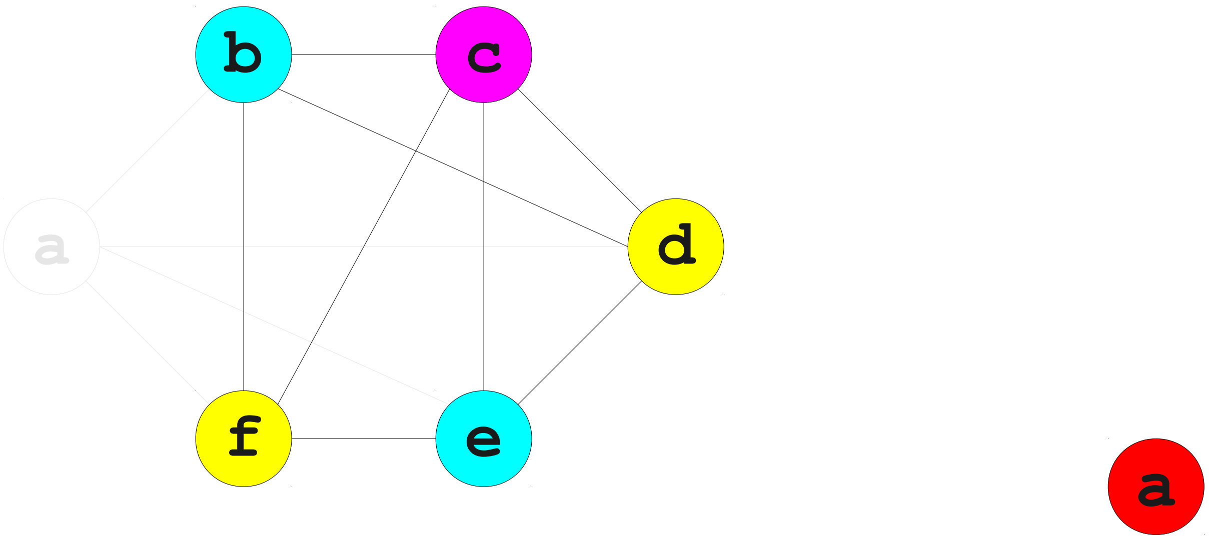
Another Example



Registers

R_0	R_1	R_2

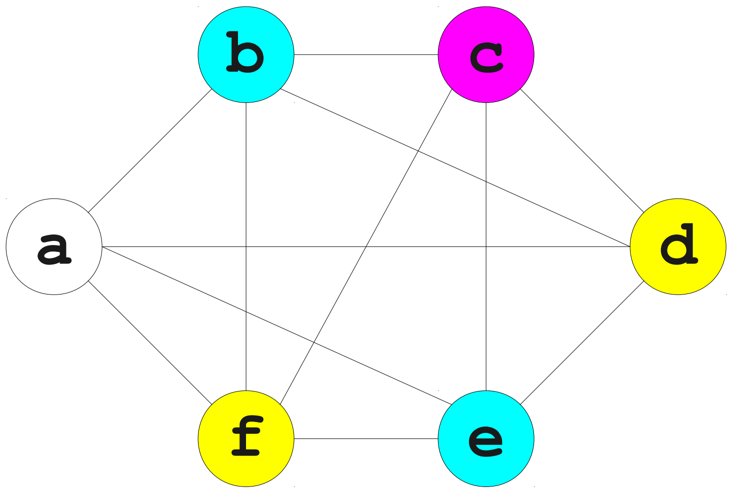
Another Example



Registers

R_0	R_1	R_2

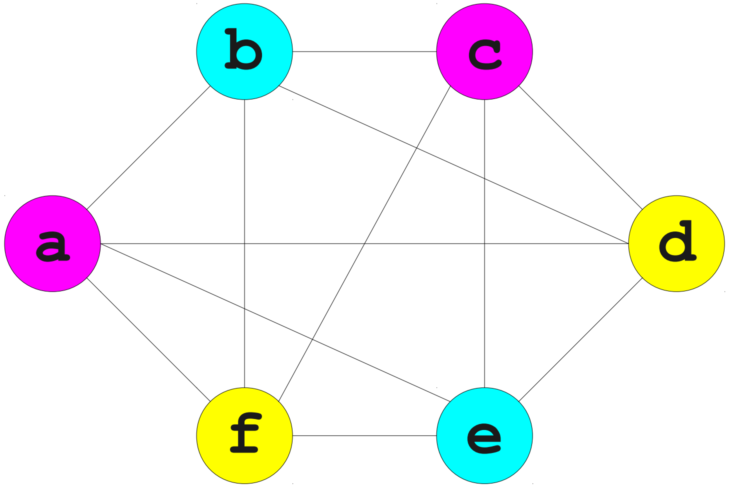
Another Example



Registers

R_0	R_1	R_2

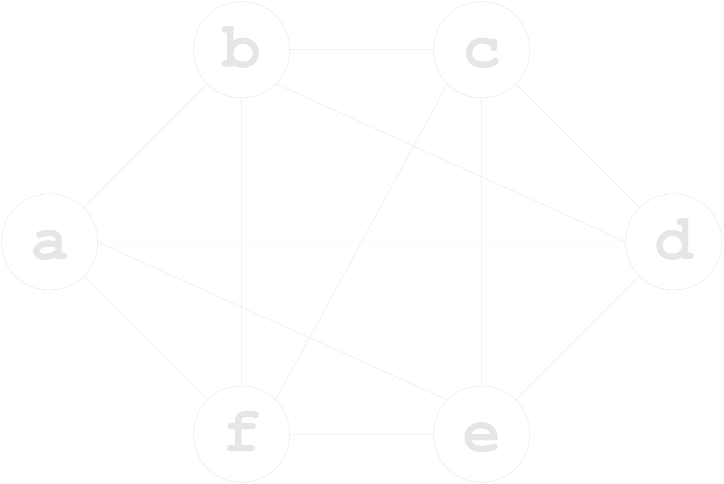
Another Example

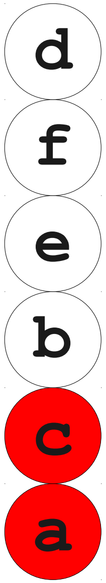


Registers

R_0	R_1	R_2

Another Example

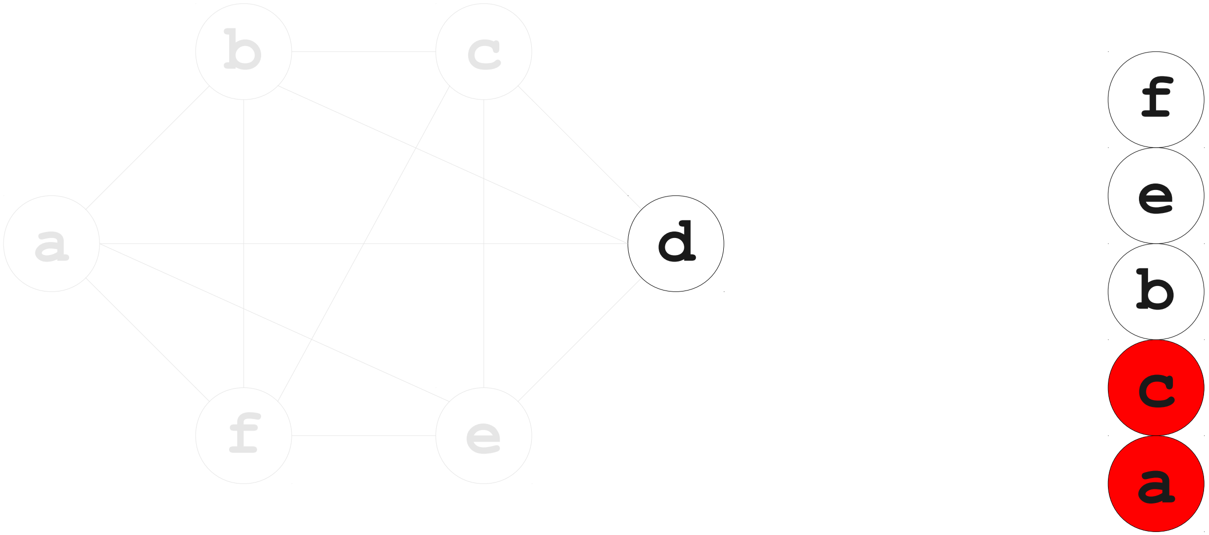




Registers

R_0	R_1	R_2

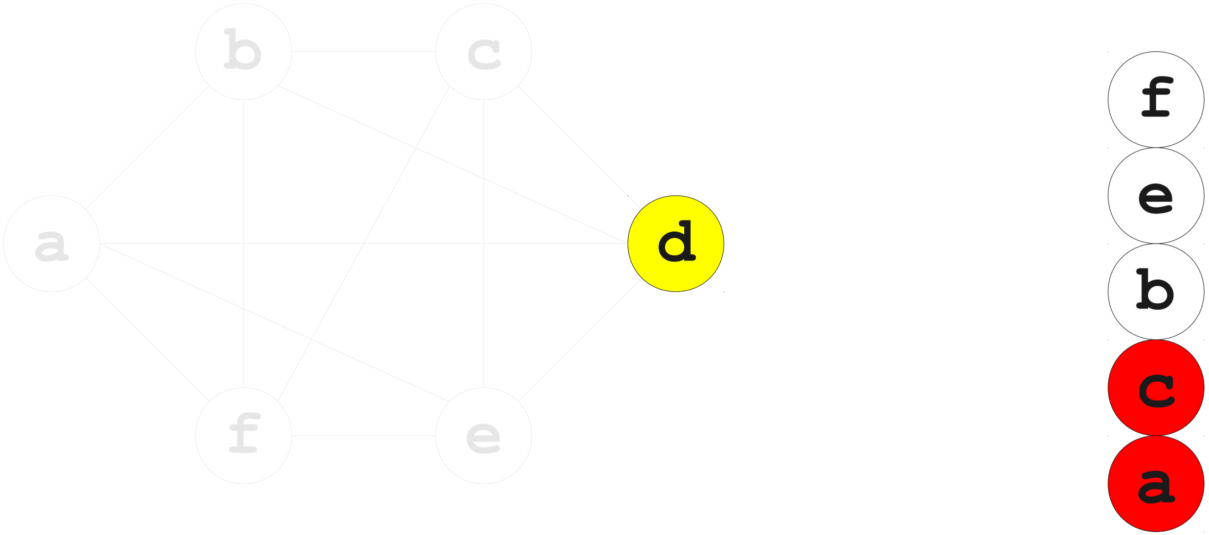
Another Example



Registers

R_0	R_1	R_2

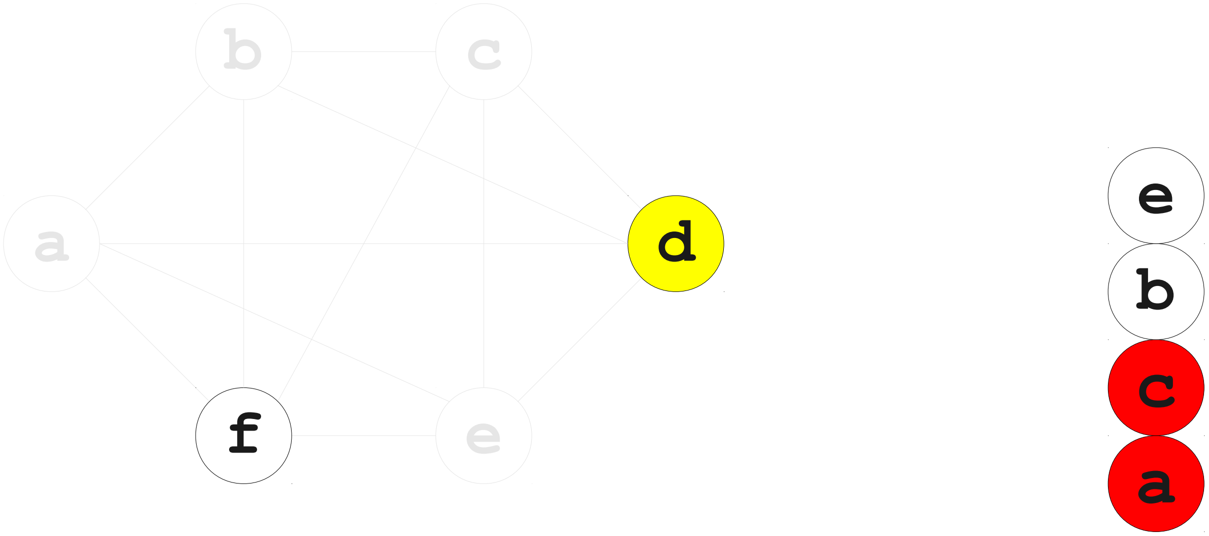
Another Example



Registers

R_0	R_1	R_2

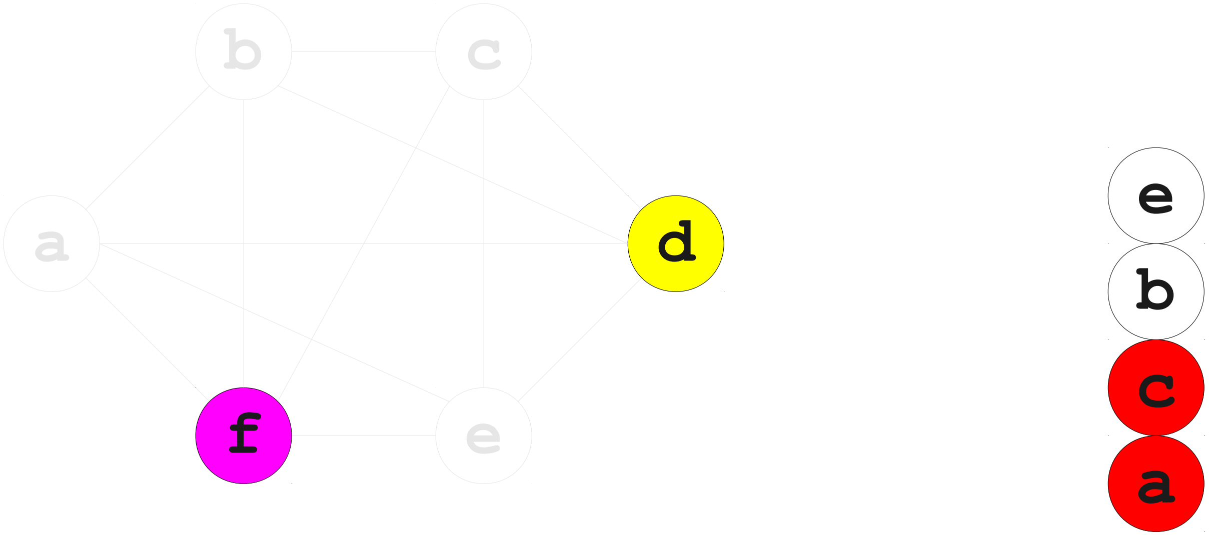
Another Example



Registers

R_0	R_1	R_2

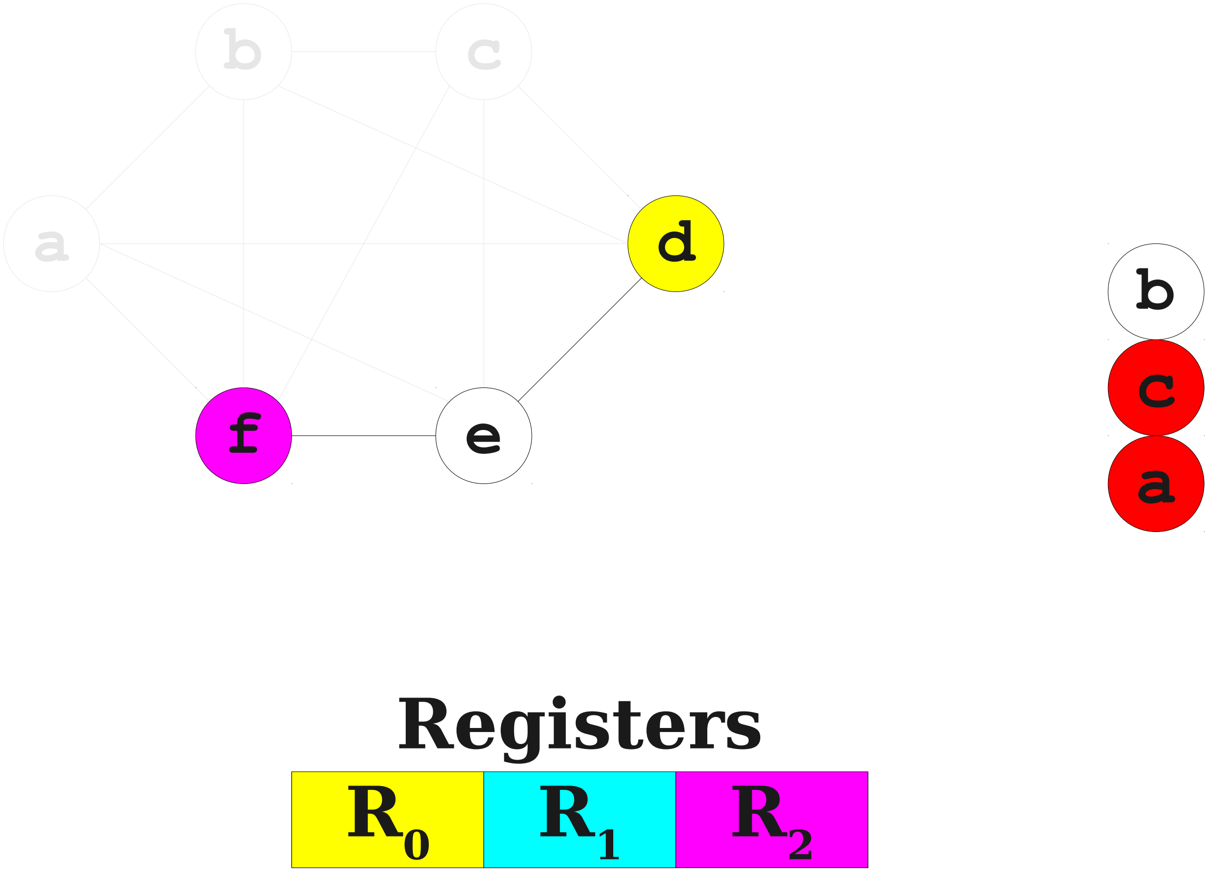
Another Example



Registers

R_0 R_1 R_2

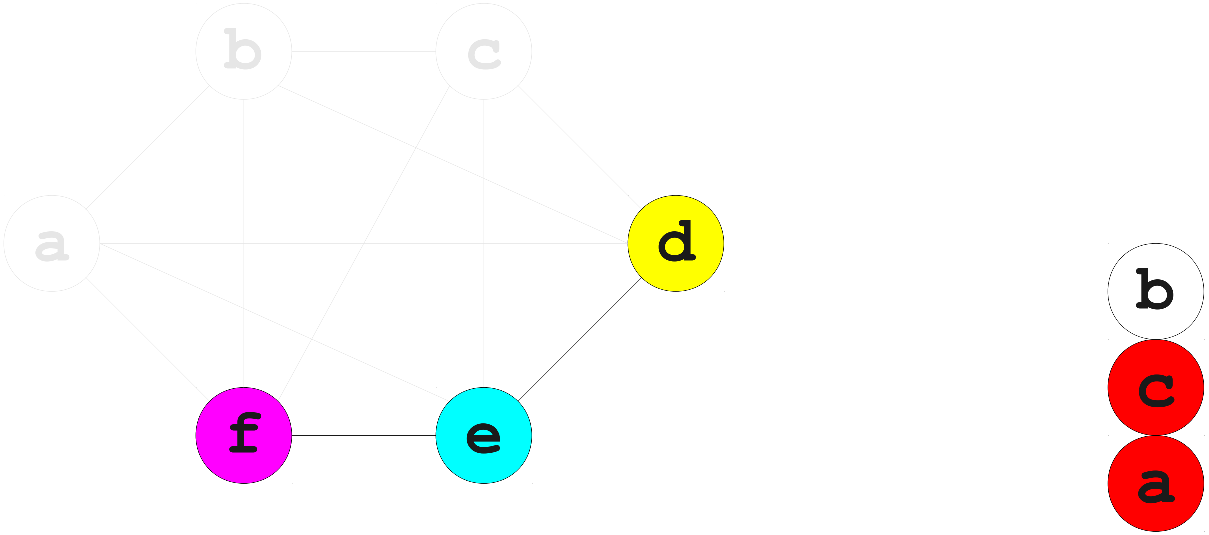
Another Example



Registers

R_0 R_1 R_2

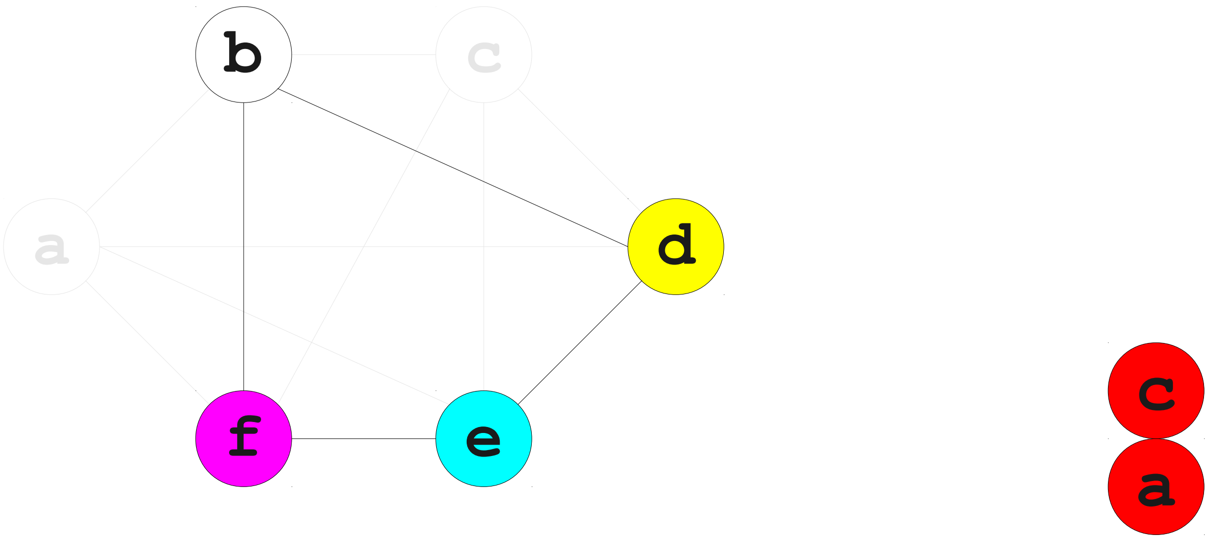
Another Example



Registers

R_0	R_1	R_2

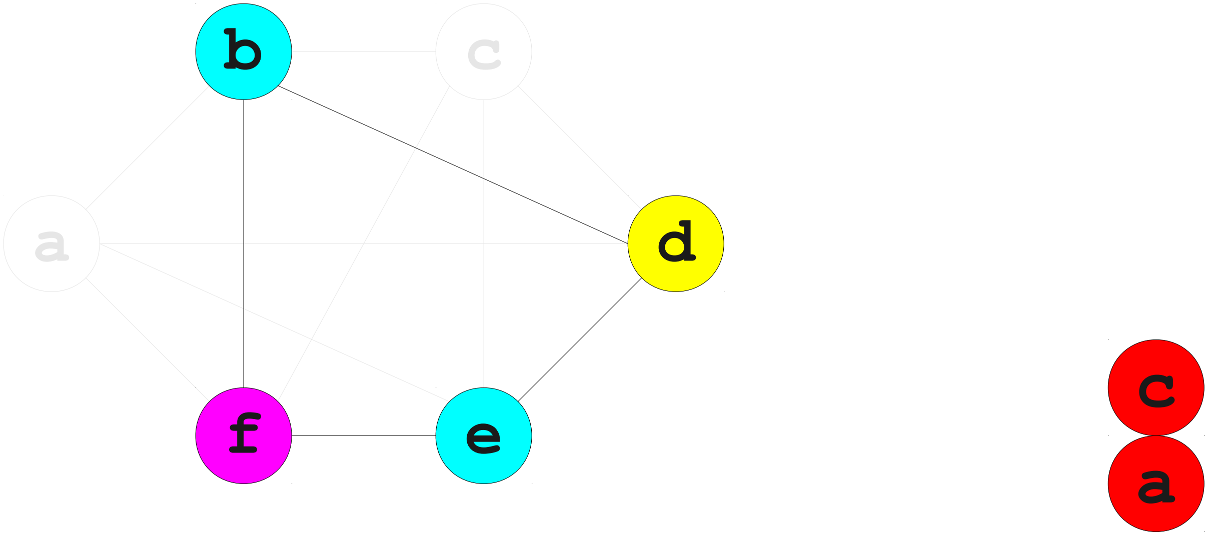
Another Example



Registers

R_0	R_1	R_2

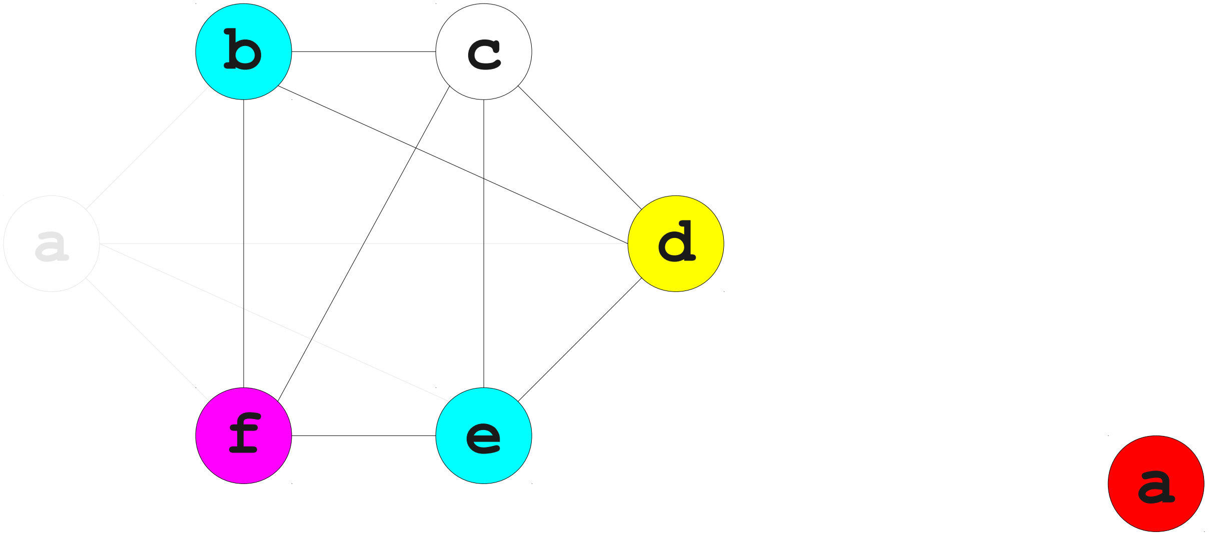
Another Example



Registers

R_0	R_1	R_2

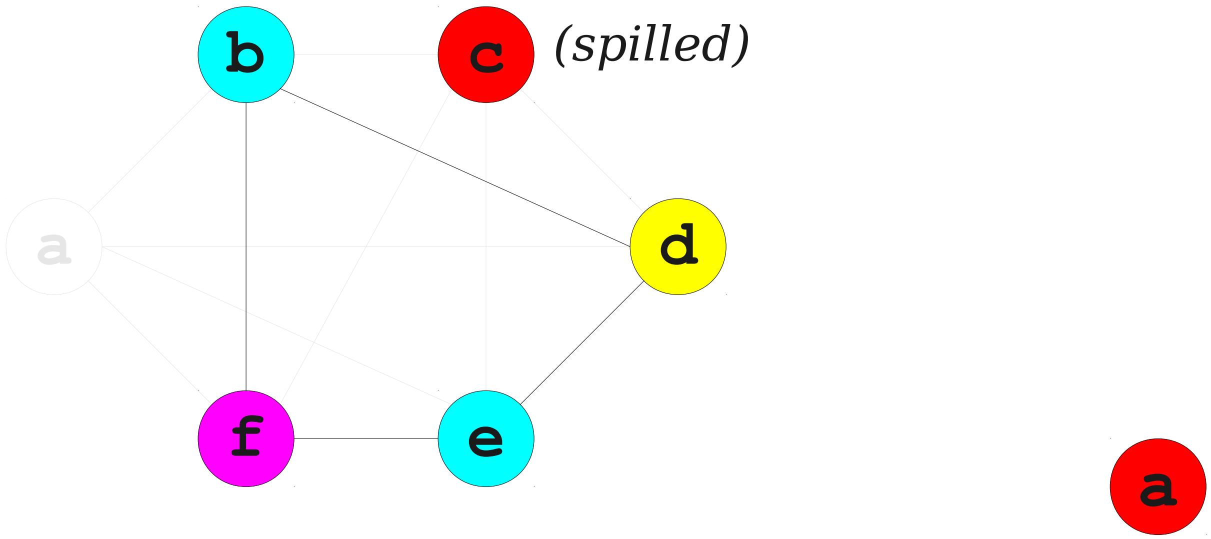
Another Example



Registers

R_0	R_1	R_2

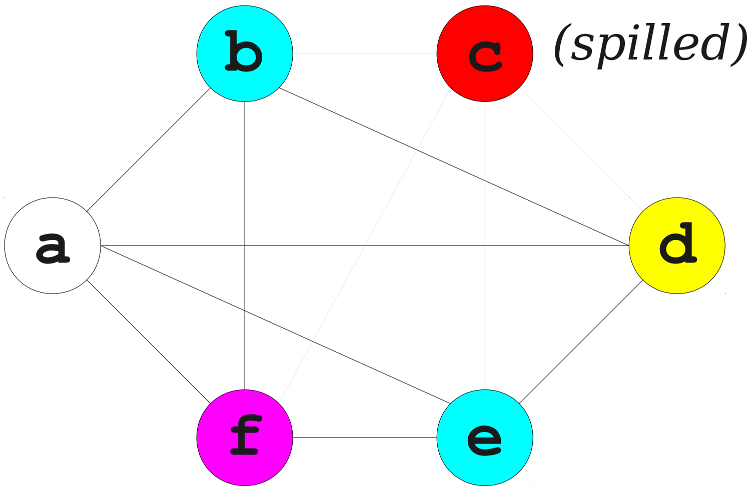
Another Example



Registers

R_0	R_1	R_2

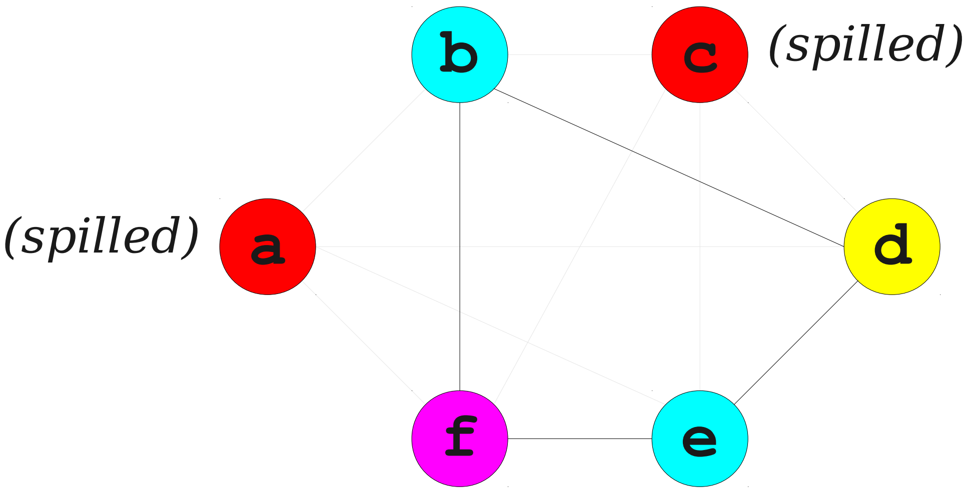
Another Example



Registers

R_0	R_1	R_2

Another Example



Registers

R_0	R_1	R_2

Improvements on Spilling

Improvements on Spilling

1. Use heuristics to decide which variables to spill

- least frequently used variable

Improvements on Spilling

1. Use heuristics to decide which variables to spill

- least frequently used variable

2. Instead of always keeping the spilled variable on the stack, break the live range of the variable up by introducing new temporaries and reconstruct the conflict graph.

- more nodes but fewer edges

Implementing Reg Allocation

1. For each function definition, we’ll run liveness, conflict analysis and register allocation, producing a mapping from variable names to registers/stack offsets.

Implementing Reg Allocation

1. For each function definition, we'll run liveness, conflict analysis and register allocation, producing a mapping from variable names to registers/stack offsets.
2. How does your code generation change?

Implementing Reg Allocation

1. For each function definition, we'll run liveness, conflict analysis and register allocation, producing a mapping from variable names to registers/stack offsets.
2. How does your code generation change?

- Variables

Implementing Reg Allocation

1. For each function definition, we’ll run liveness, conflict analysis and register allocation, producing a mapping from variable names to registers/stack offsets.

2. How does your code generation change?

- Variables
- Function prelude/epilogue

Implementing Reg Allocation

1. For each function definition, we’ll run liveness, conflict analysis and register allocation, producing a mapping from variable names to registers/stack offsets.

2. How does your code generation change?

- Variables
- Function prelude/epilogue
- Function calls