Register Allocation

(via graph coloring and spilling)

Register allocation

- LLVM IR uses an unbounded set of virtual registers.
- Register allocation yields code in terms of hardware registers.
- These are limited. For example, x86_64 has: 16x general purpose (64bit) registers (plus 16x SSE registers, 2x status registers, 6x 32bit registers, 8x FPU/MMX registers).
- Using registers when possible is crucial to performance. Register access for i7-4770 is ~1 CPU cycle, L1 cache is ~4 cycles, L2 cache is ~12 cycles, L3 cache is ~36-58 cycles.
- Register allocation is as old as intermediate languages. The first FORTRAN compiler (April 1957) had a primitive register allocator.

Register allocation

- ***Register pressure*** results from a lack of machine registers for the needed virtual registers.
- If there are too many virtual registers live at once, values must be ***spilled*** into memory, usually extra space on the stack.
- The goals of performing register allocation well are:
    - To guarantee correct output code.
    - To minimize spill code (added loads and stores).
    - To minimize added spill space on the stack.

Two approaches

- The naïve approach: “local” register allocation.
  - Allocates registers in each basic block separately.
  - Traverses the basic block, maintaining a map from virtual registers to either a machine register or offset on the stack.
  - When a virtual register not currently stored in a machine register is accessed, a machine register is ***spilled*** onto the stack using a store instruction and the required register is loaded. The virtual register map is updated for both.
- The graph-coloring approach: “global” register allocation.
  - The new Chez Scheme produces 15-24% faster code.

The naïve approach

Assuming 3 hardware registers: `rax, rbx, rcx`

```
→   ...
    %b = add %a, 1
    %c = mul %a, %b
    %d = add %c, %b
    %e = add %b, %a
    ...
```

```
%a → rax
```

The naïve approach

Assuming 3 hardware registers: `rax, rbx, rcx`

```
→   ...
    %b = add %a, 1
    %c = mul %a, %b
    %d = add %c, %b
    %e = add %b, %a
    ...
```

```
%a → rax
%b → rbx
```

The naïve approach

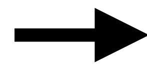
Assuming 3 hardware registers: `rax, rbx, rcx`

```
  ...
  %b = add %a, 1
→ %c = mul %a, %b
  %d = add %c, %b
  %e = add %b, %a
  ...
```

```
%a → rax
%b → rbx
%c → rcx
```

The naïve approach

Assuming 3 hardware registers: `rax, rbx, rcx`



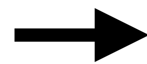
```
...
%b = add %a, 1
%c = mul %a, %b
store rax, rsp+0
%d = add %c, %b
%e = add %b, %a
...
```

```
%a → rsp+0
%b → rbx
%c → rcx
%d → rax
```

The naïve approach

Assuming 3 hardware registers: `rax, rbx, rcx`

```
...
%b = add %a, 1
%c = mul %a, %b
store rax, rsp+0
%d = add %c, %b
store rcx, rsp+16
store rbx, rsp+8
rbx = load rsp+0
%e = add %d, %a
...
```



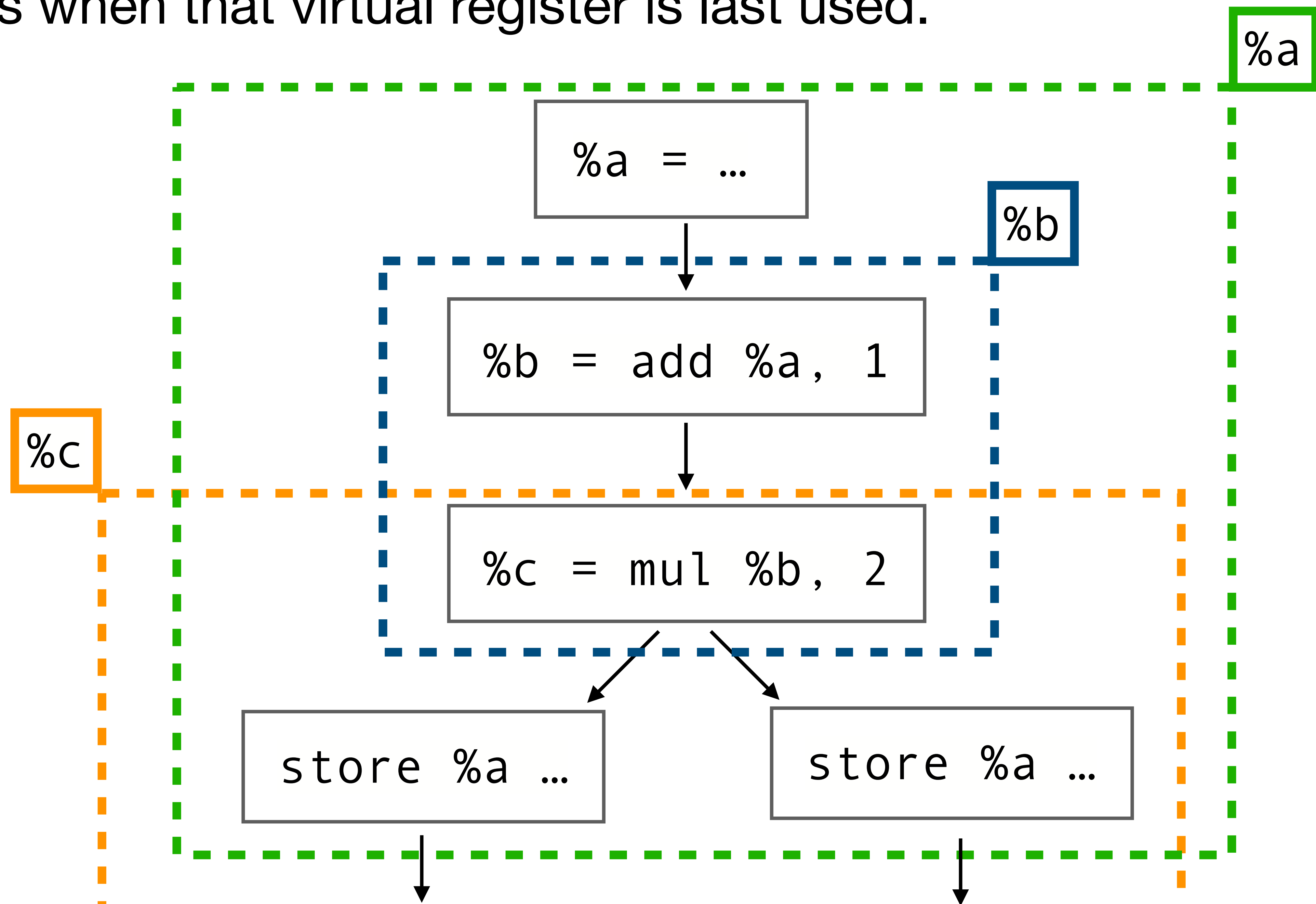
```
%a → rbx
%b → rsp+8
%c → rsp+16
%d → rax
%e → rcx
```

Graph-coloring approach

- Interprets register allocation as a graph coloring problem:
    - Given a graph G and number of colors k, a valid solution is an assignment of G's nodes to colors, numbered [1..k], where no two adjacent nodes have the same color.
    - The problem is NP-Hard for $k > 2$.
- The algorithm constructs a *register interference graph* where:
    - There is a node for each SSA register (or assignment)
    - There is an edge between two nodes when both registers may be live at some point in the code.
    - k is the number of hardware registers in our allocation pool.

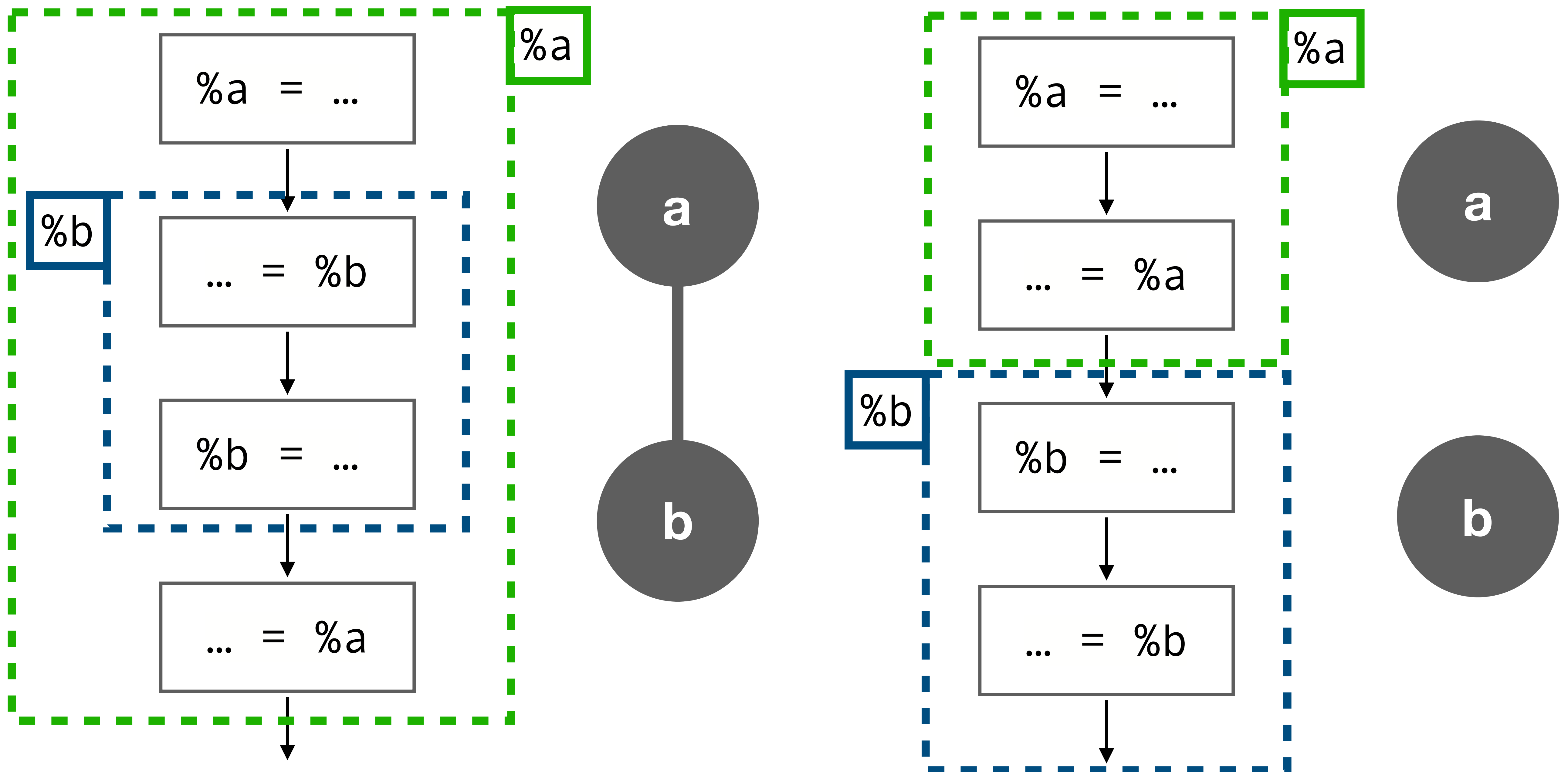
Liveness & live ranges

- Begins when a virtual register is assigned (for SSA this is unique).
- Ends when that virtual register is last used.



Register interference graph

- At each instruction S in the procedure, add an edge (a, b) for all pairs of registers, %a and %b, live at S.

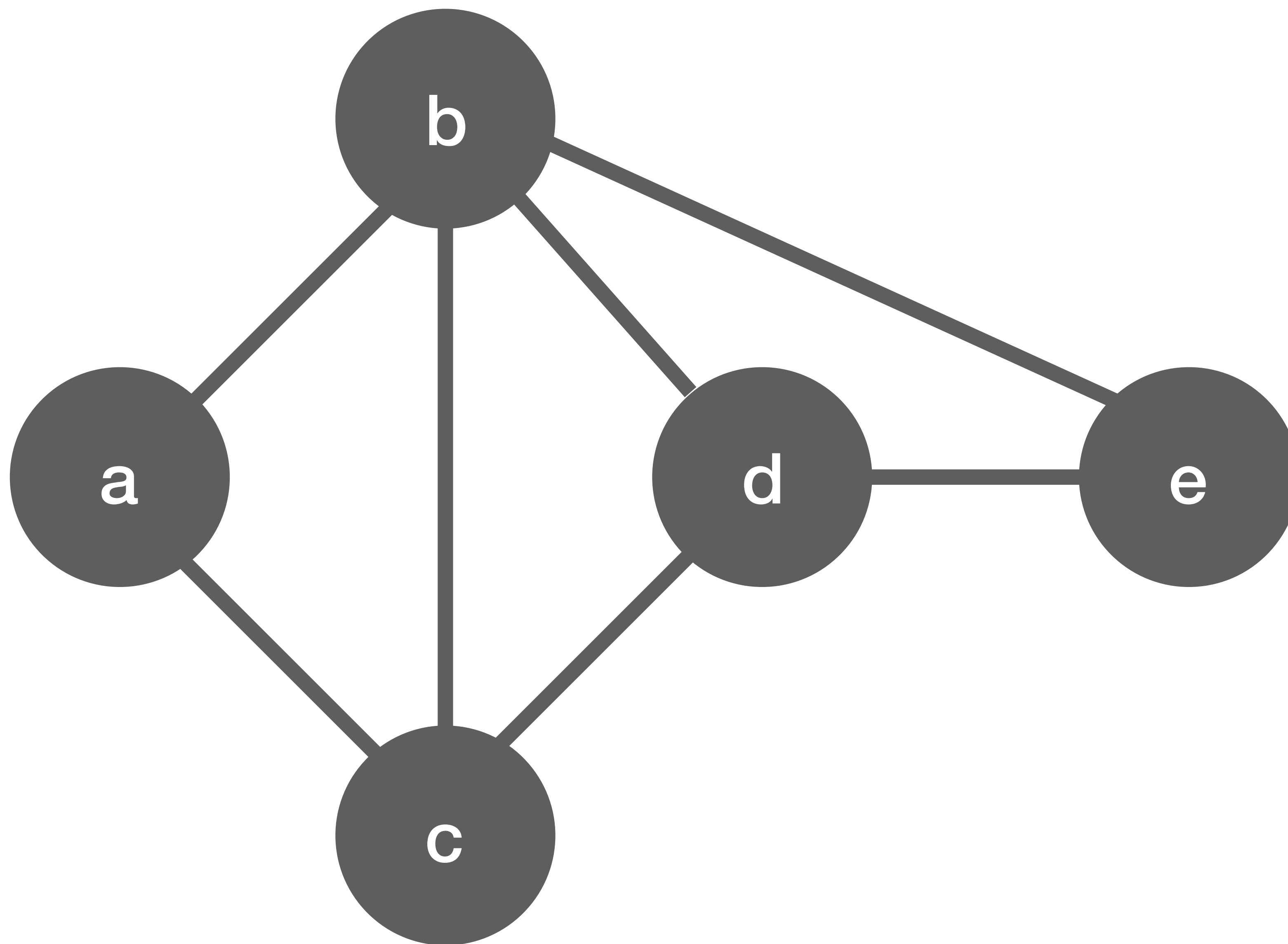


Graph coloring via simplification

- Solve using a heuristic that can't guarantee an optimal coloring.
- First, reduce the complexity of the problem:
    - While there exists a node R with a degree < k: remove R and push it onto a stack of low-degree nodes.
- Then, either all nodes in the graph are removed, or a node with degree of at least k is left over. Such a virtual register must* be spilled to an address on the stack.
- Last, given an empty graph and stack of nodes of degree < k, each node can be popped and inserted into the graph with a kth color not shared by any of its neighbors.

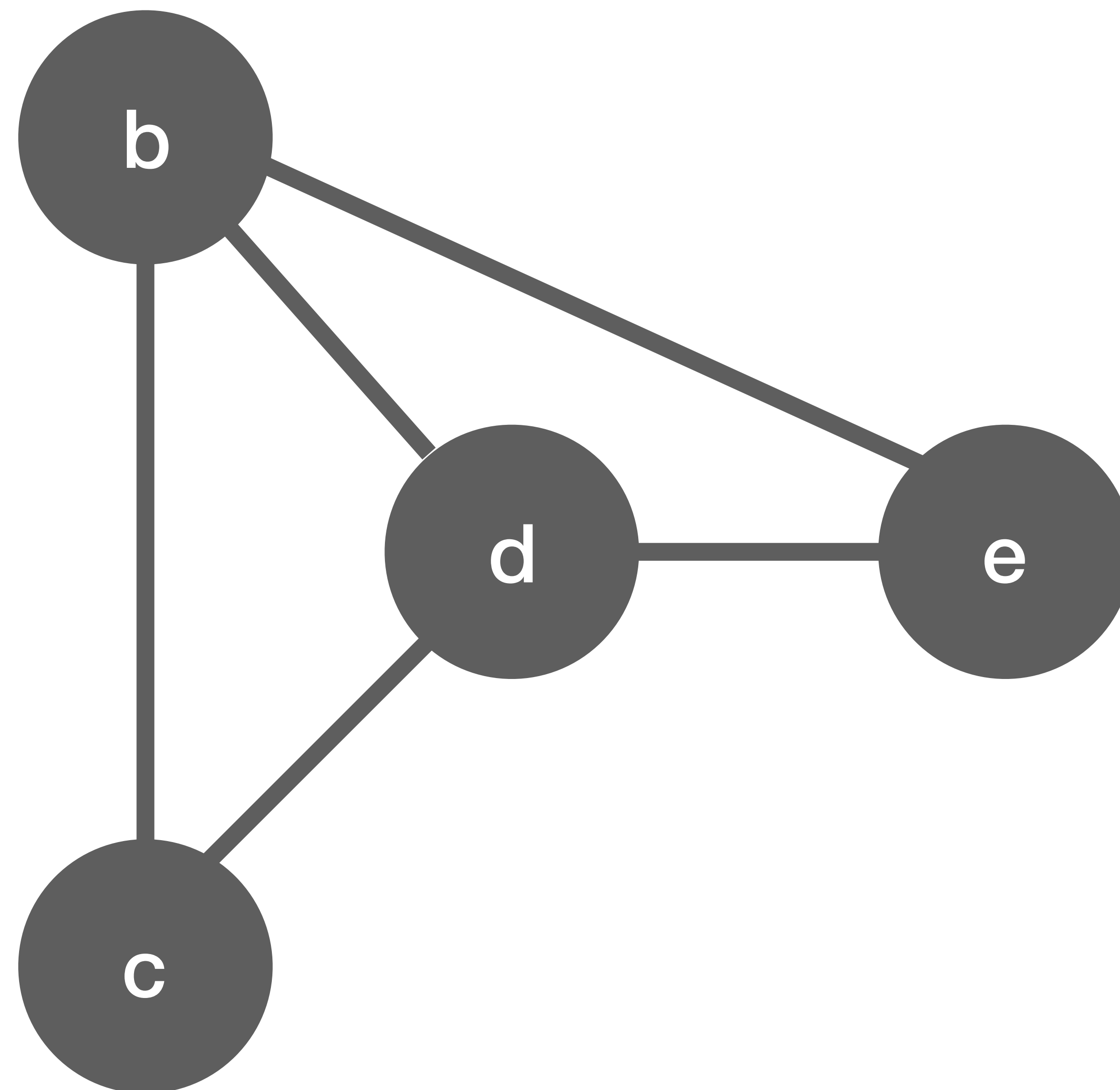
Graph coloring via simplification

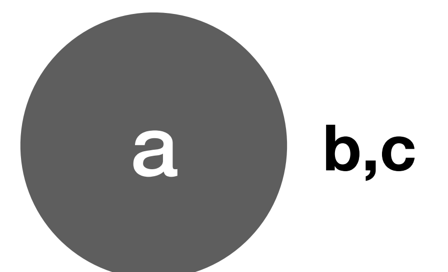
Assuming 3 hardware registers / 3 colors



Graph coloring via simplification

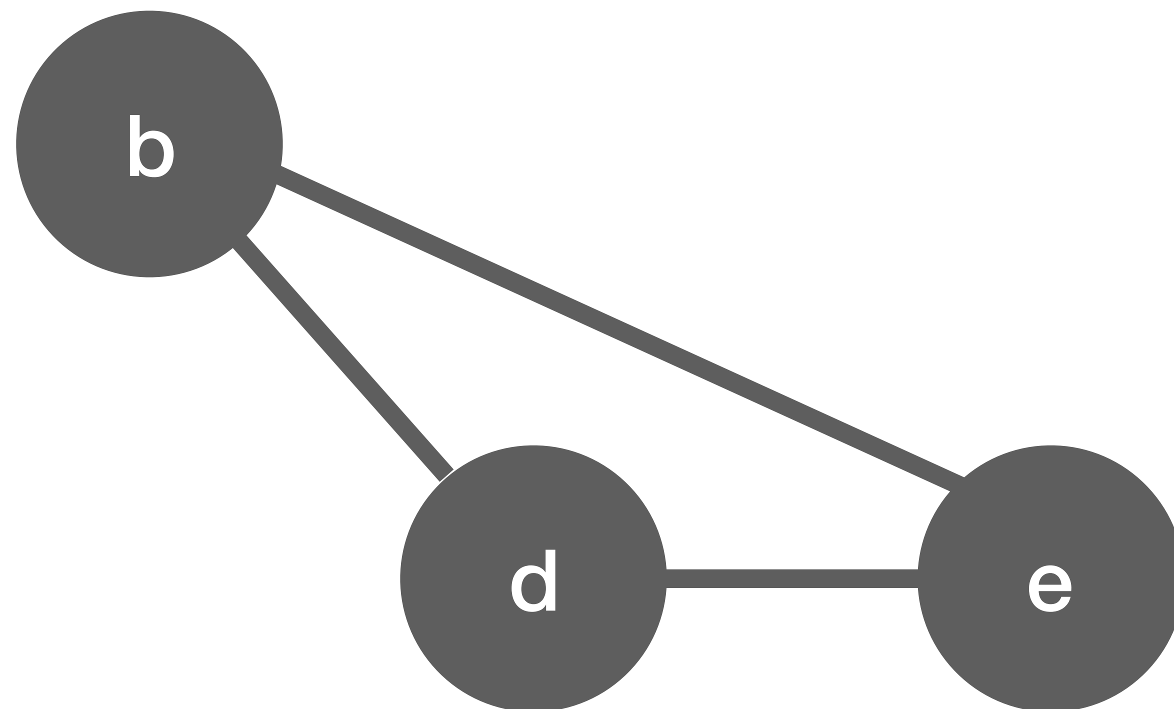
Assuming 3 hardware registers / 3 colors

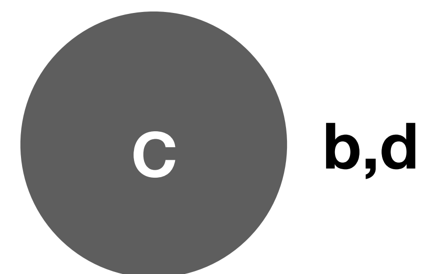




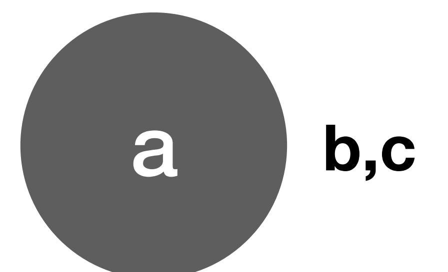
Graph coloring via simplification

Assuming 3 hardware registers / 3 colors





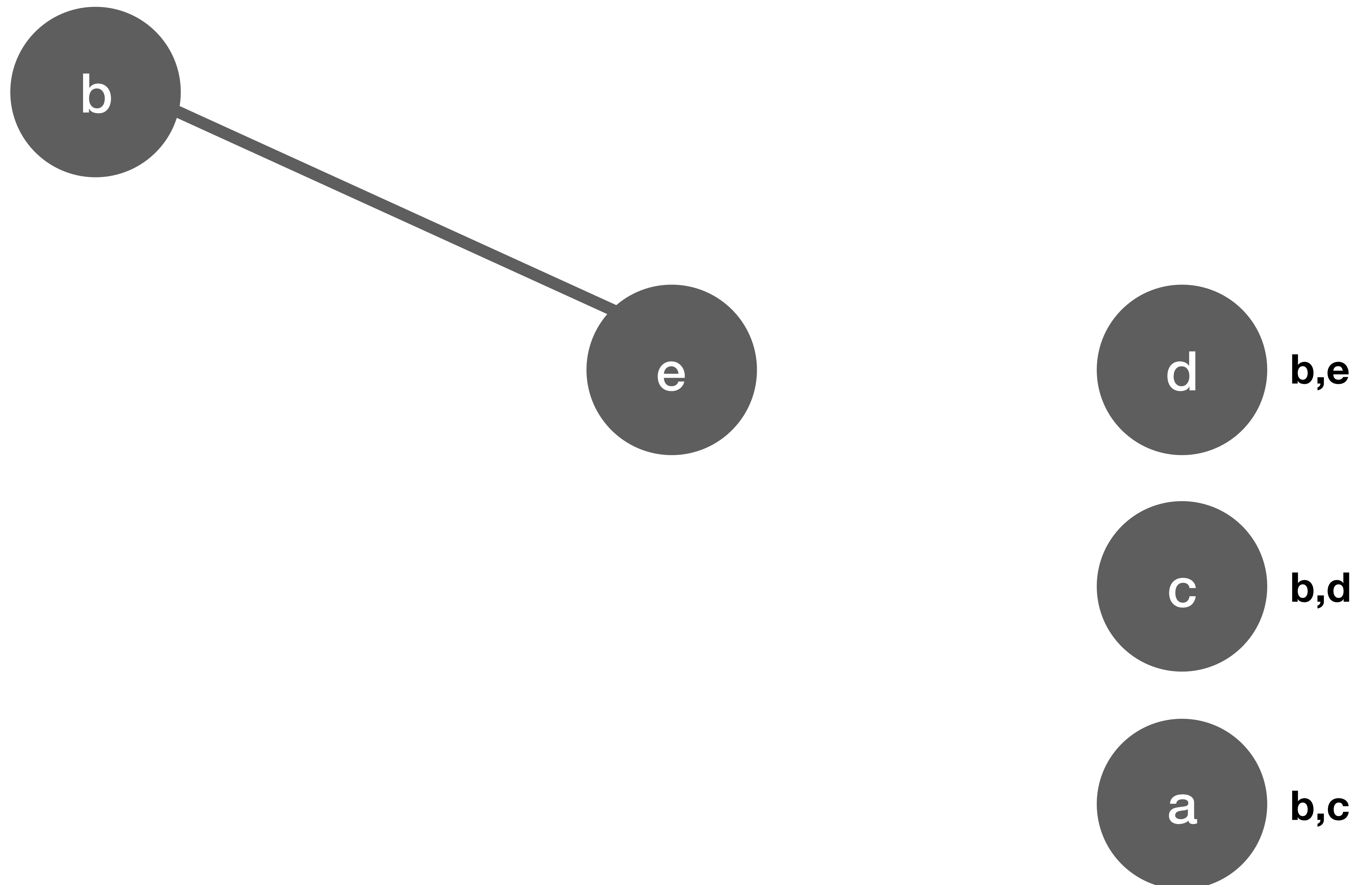
b,d



b,c

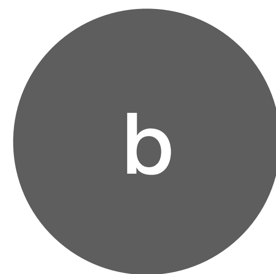
Graph coloring via simplification

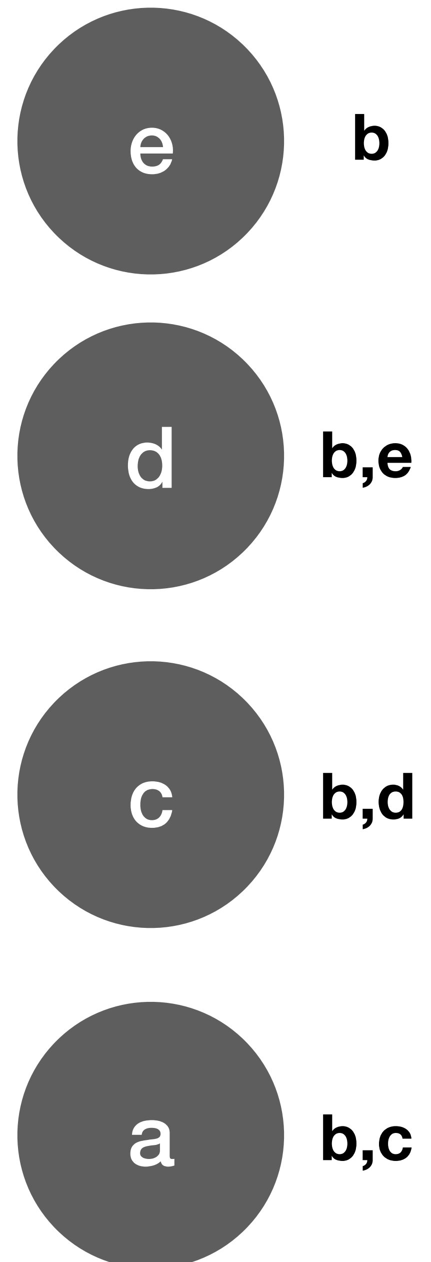
Assuming 3 hardware registers / 3 colors



Graph coloring via simplification

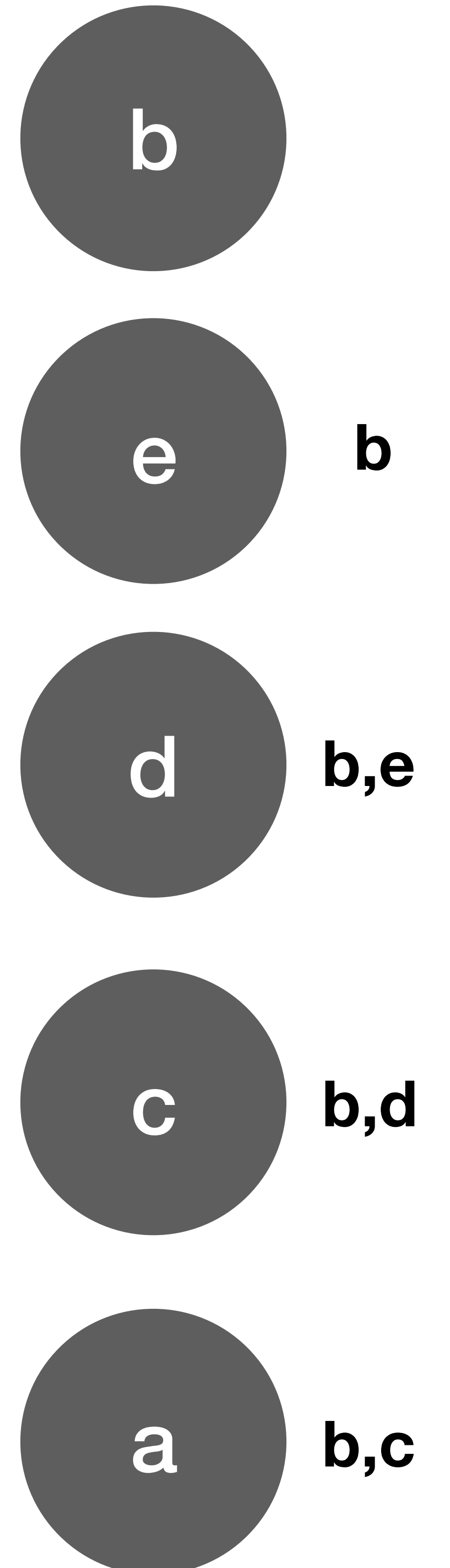
Assuming 3 hardware registers / 3 colors





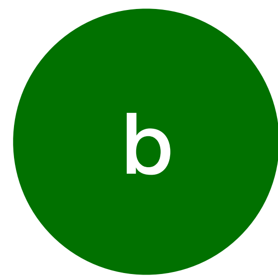
Graph coloring via simplification

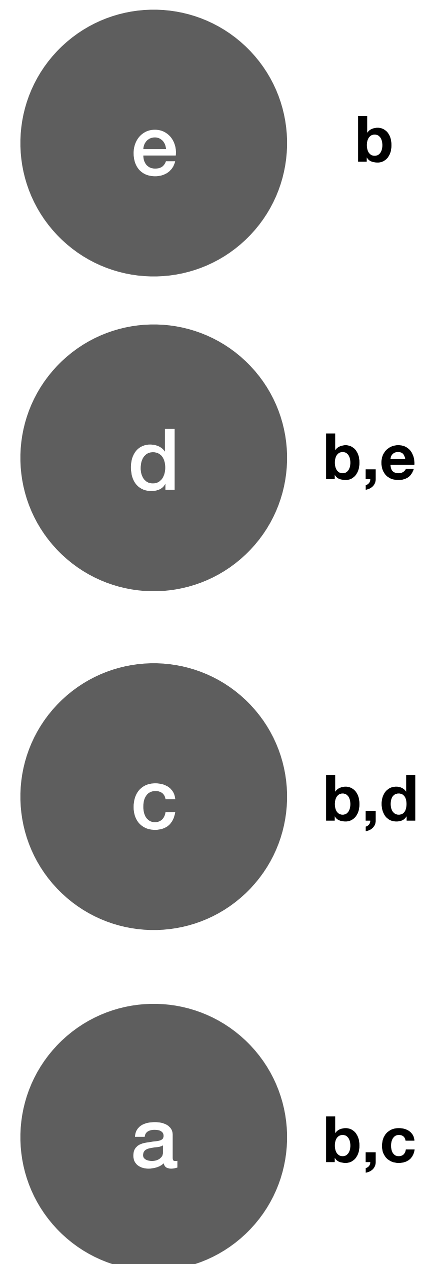
Assuming 3 hardware registers / 3 colors



Graph coloring via simplification

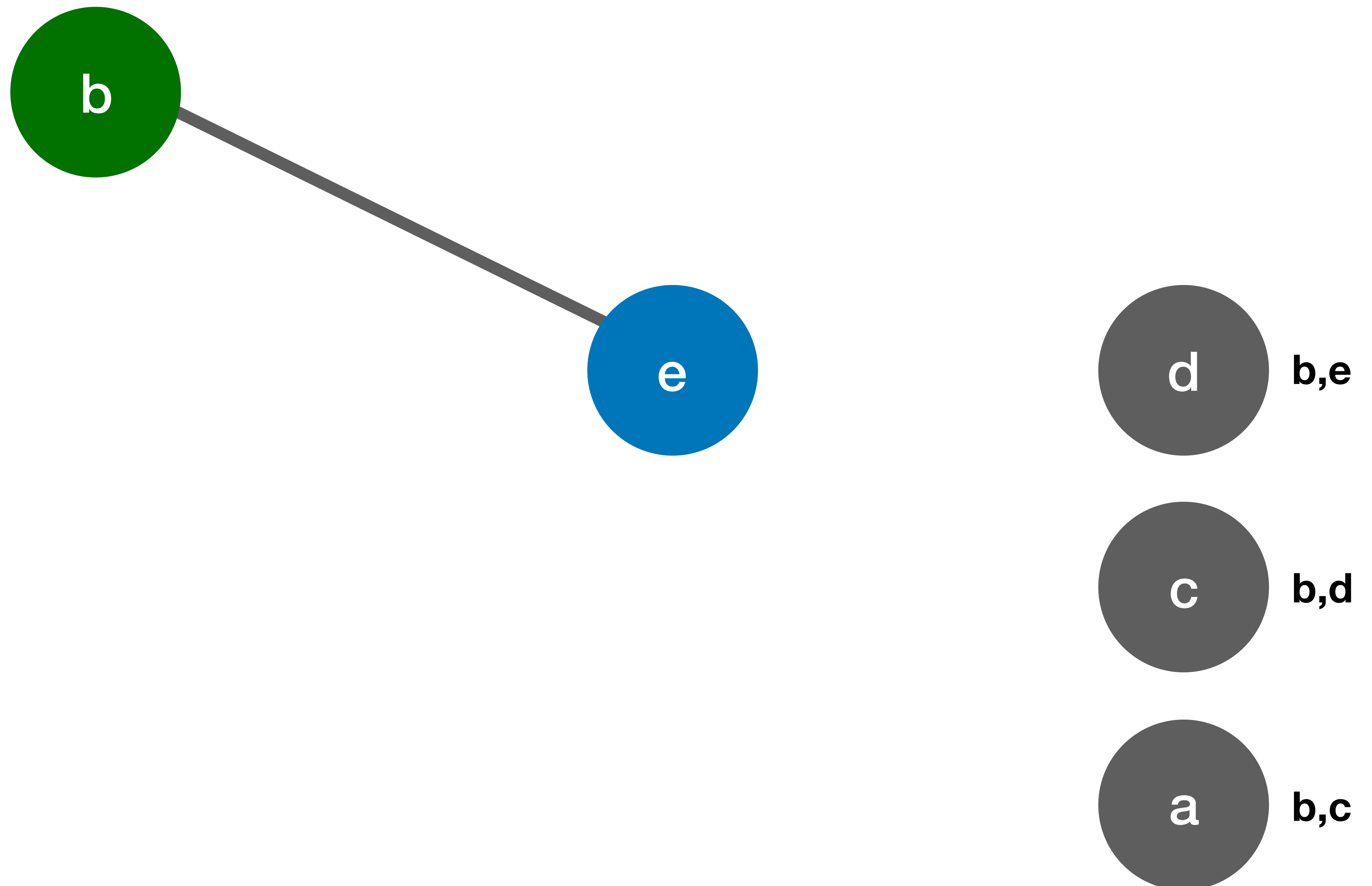
Assuming 3 hardware registers / 3 colors





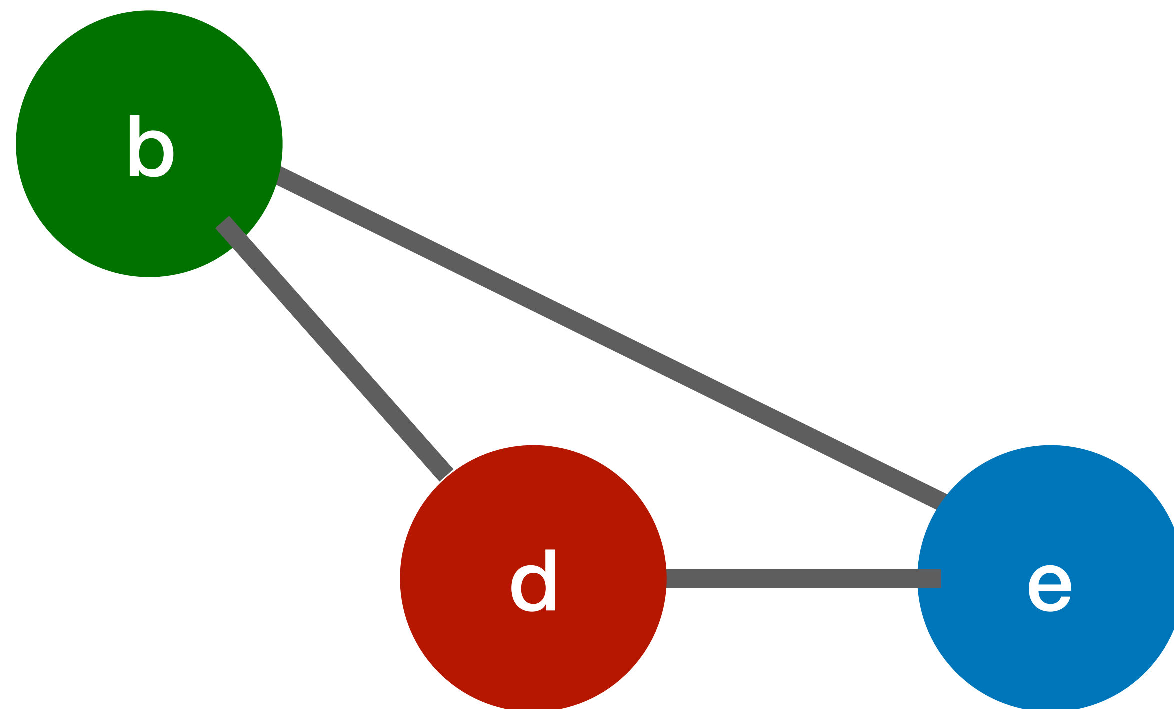
Graph coloring via simplification

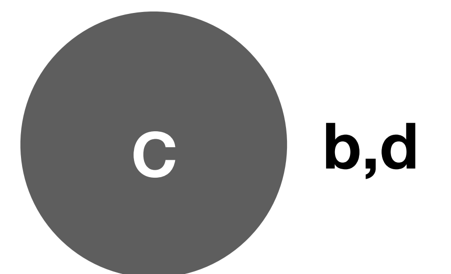
Assuming 3 hardware registers / 3 colors



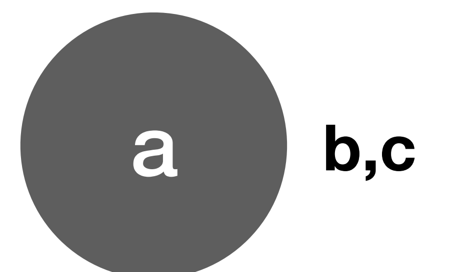
Graph coloring via simplification

Assuming 3 hardware registers / 3 colors





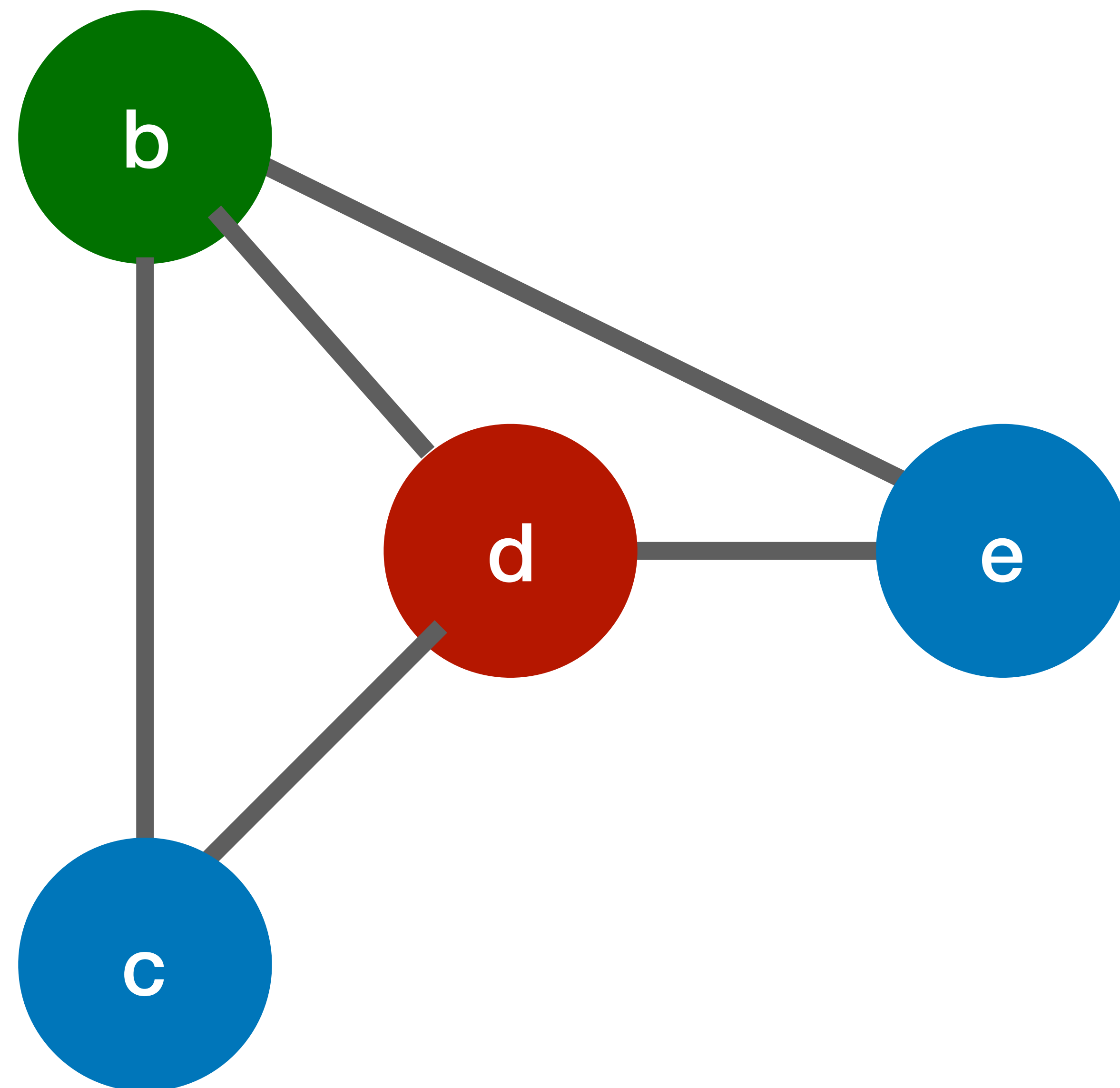
b,d

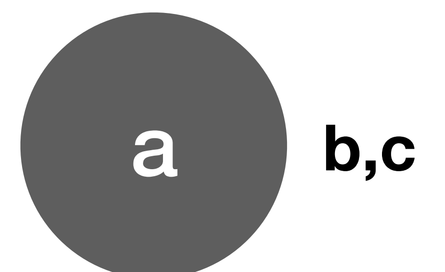


b,c

Graph coloring via simplification

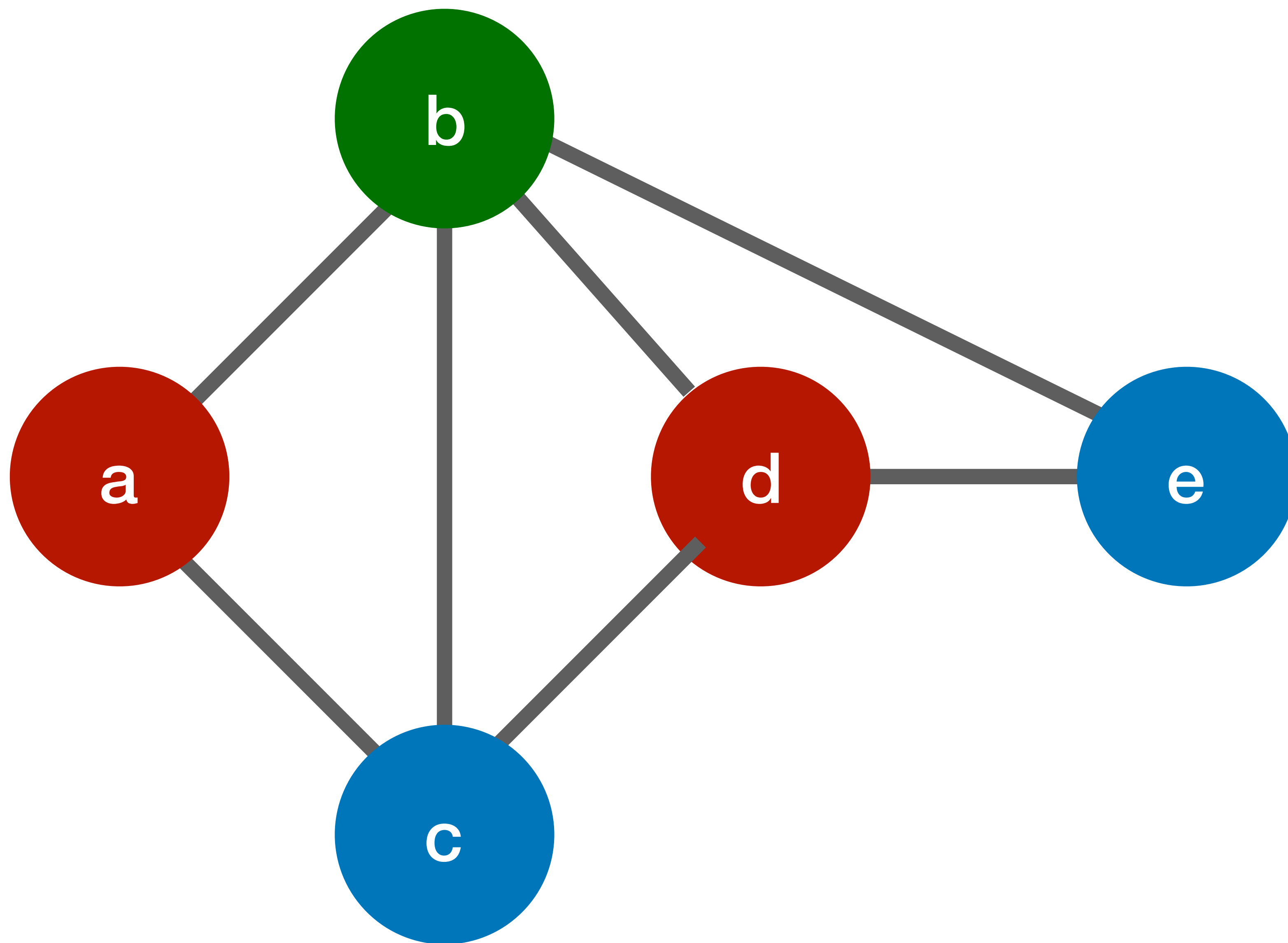
Assuming 3 hardware registers / 3 colors





Graph coloring via simplification

Assuming 3 hardware registers / 3 colors



Graph coloring via simplification

- If, at some point, all remaining nodes have a degree of at least k, then we must spill one of those virtual registers to the stack.
- Pick the virtual register with lowest ***spill cost*** by some heuristic.
- There are various strategies for spilling. Two common ones:
    - Reserve a subset of registers for spilled values and wrap every use of the register with a load and store.
    - Split the register into 2^+ virtual registers (connected with a store & load) and recompute live ranges; the next iteration would then begin with larger but simpler graph where the spilled node is replaced by 2^+ lower-degree nodes.