PROGRAMMING LANGUAGES LABORATORY

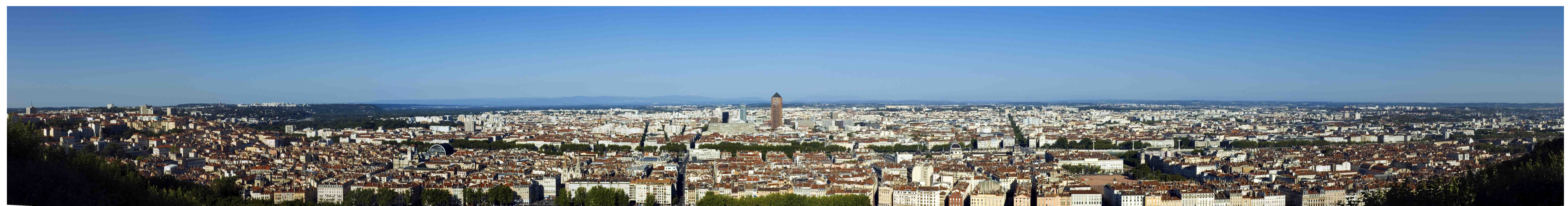
Universidade Federal de Minas Gerais - Department of Computer Science

REGISTER ALLOCATION

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

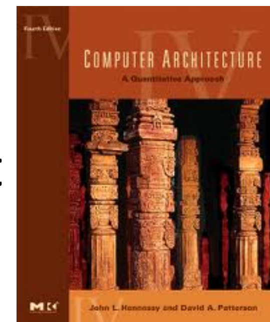


RESEARCH SCHOOLS OF THE ÉCOLE NORMALE SUPÉRIEURE DE LYON

Register Allocation

- Register Allocation is the problem of finding storage locations to the values manipulated by a program.
- These values might be stored either in registers, or in memory.
- Registers are fast, but they come in small quantity.
- Memory is plenty, but it has slower access time.
- A good register allocator should strive to keep in registers the variables used more often.

"Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important - if not the most important - of the optimizations."



Hennessy and Patterson (2006) - [Appendix B; p. 26]

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DCC - UFMG

Register Allocation

- Register Allocation is the problem of finding storage locations to the values manipulated by a program.
- These values might be stored either in registers, or **in memory**.
- Registers are fast, but they come in small quantity.
- Memory is plenty, but it has slower access time.
- A good register allocator should strive to keep in registers the **variables used more** often.

1) How the values mapped into memory are accessed?
2) How to determine the variables that are used more often?
3) Which other optimizations can register allocation improve?

"Because of the central role that register allocation plays, both in speeding up the code and in **making other optimizations useful**, it is one of the most important - if not the most important - of the optimizations."

Hennessy and Patterson (2006) - [Appendix B; p. 26]

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

The Three Aspects of Register Allocation

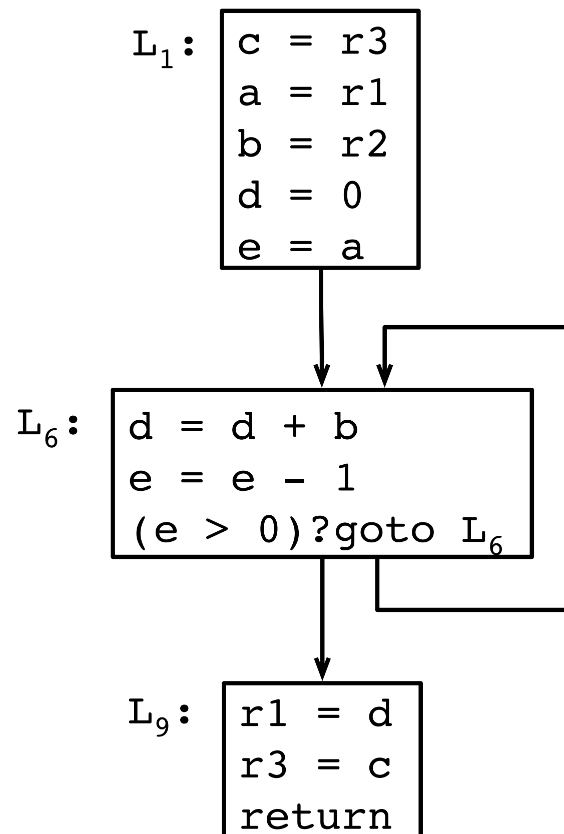
- The task of determining in which register each variable will be kept is called *register assignment*.
- If a variable must be mapped into memory, we call it *a spill*. Spilling is the task of determining which variables must be mapped to memory.
- If we can assign the same register to two variables related by a move instruction, then we can eliminate this move. This optimization is called *coalescing*.

1) Which of these three aspects do you think yields the largest performance gains?

2) Is it clear why we can remove a copy instruction a := b, if both a and b are assigned the same register?

Register Constraints

- Some variables must be given specific registers, due to constraints imposed by the compiler or the architecture.



- In this example, we assume that arguments are passed in through registers r1, r2 and r3, and return values are passed out via registers r1 and r3.

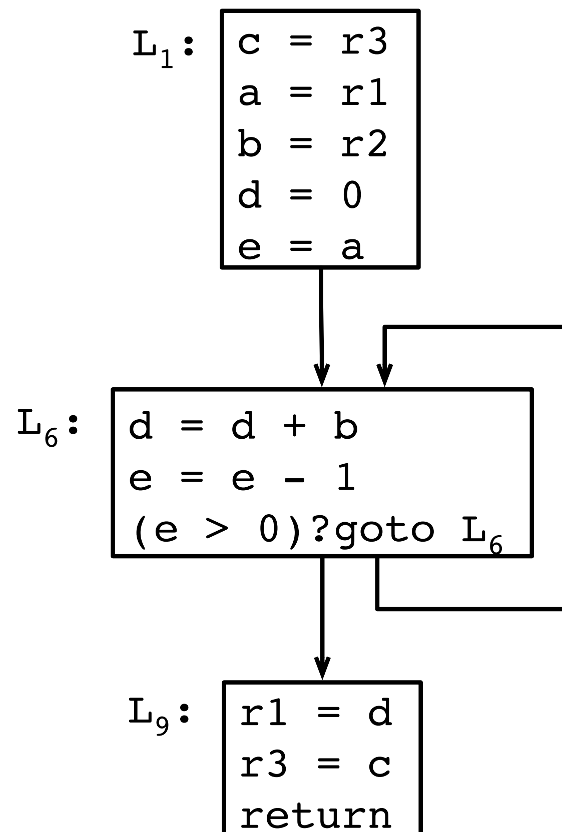
Some authors call the variables in the program *virtual registers*, and the registers themselves *physical registers*.

Which other constraints can you think about, besides calling conventions?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DCC-UFMG

Register Allocation and Liveness

- The main constraint that the register allocator must take into consideration is the fact that variables that are alive at the same program point must be assigned different registers.



1) Is it clear why two variables that are simultaneously alive must be given different registers?

2) Is there any situation in which two variables that are simultaneously alive can be assigned the same register?

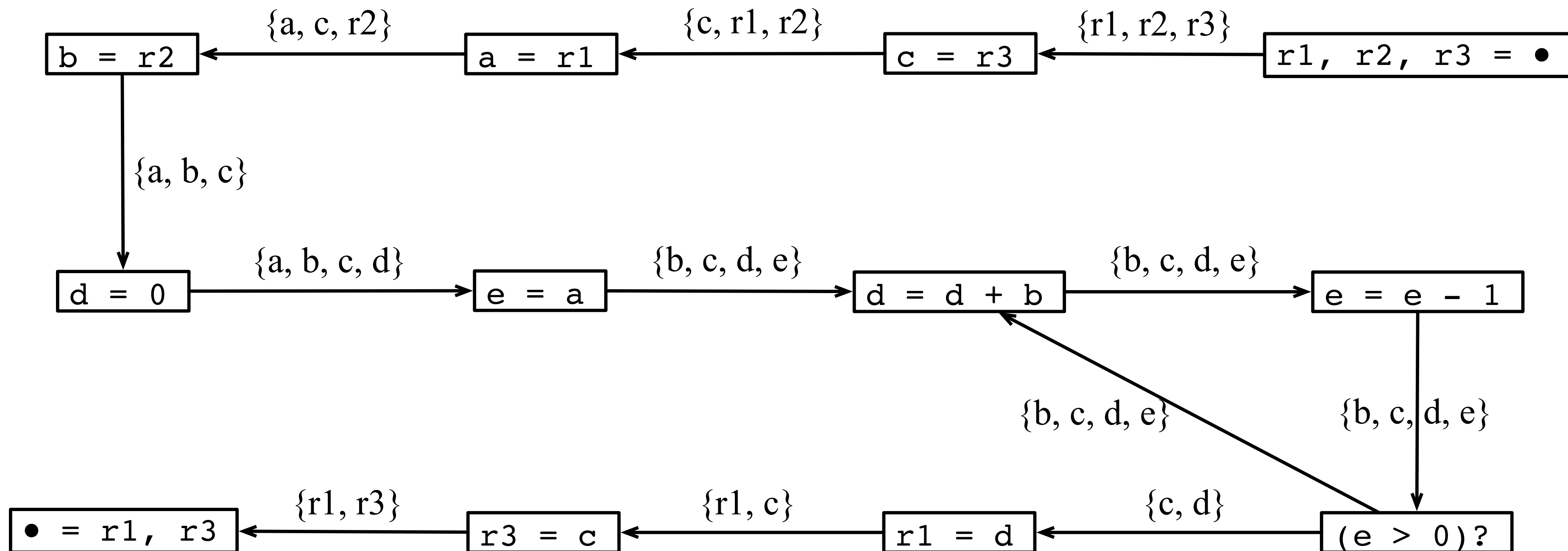
3) Can you solve liveness analysis for the program on the left?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Register Allocation and Liveness

1) How many registers do you think we would need to compile the program below?

2) In the absence of physical register constraints, is it always the case that the maximum number of live variables equals the minimum number of registers necessary to compile the program?

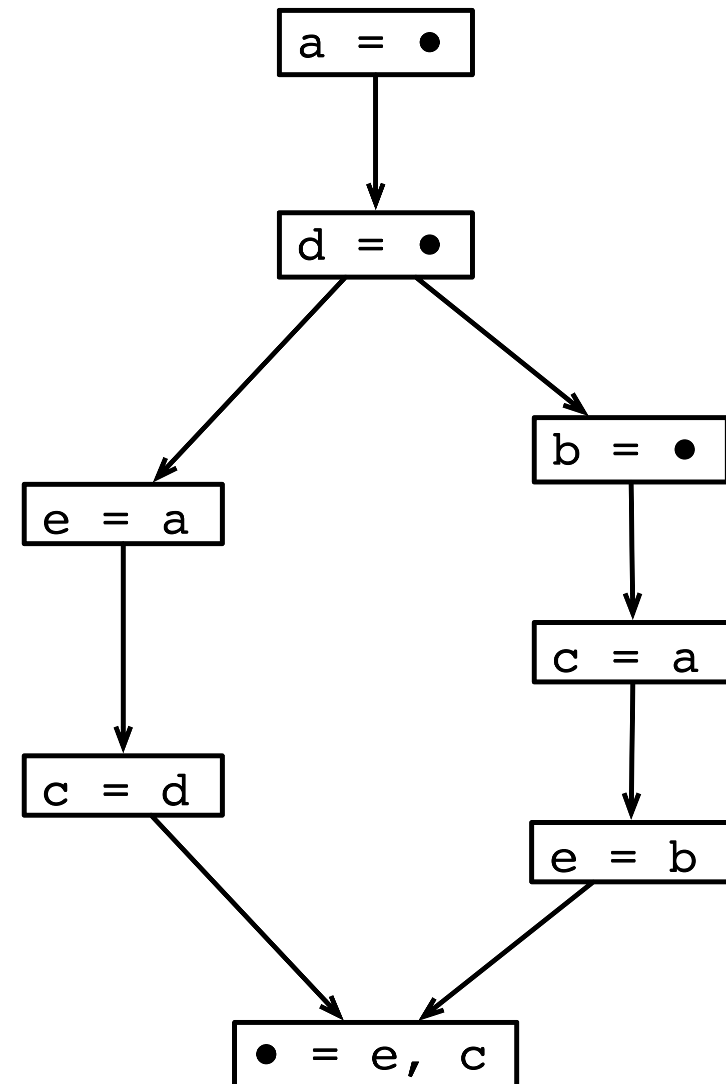


MinReg vs MaxLive

- MaxLive is the maximum number of registers that are simultaneously alive at any program point of the program's control flow graph.
- The minimum number of registers that a program needs, e.g., MinReg, is greater than or equal MaxLive.
- This difference is strict: there exist programs where MinReg > MaxLive.

1) What is MaxLive for the program on the right?

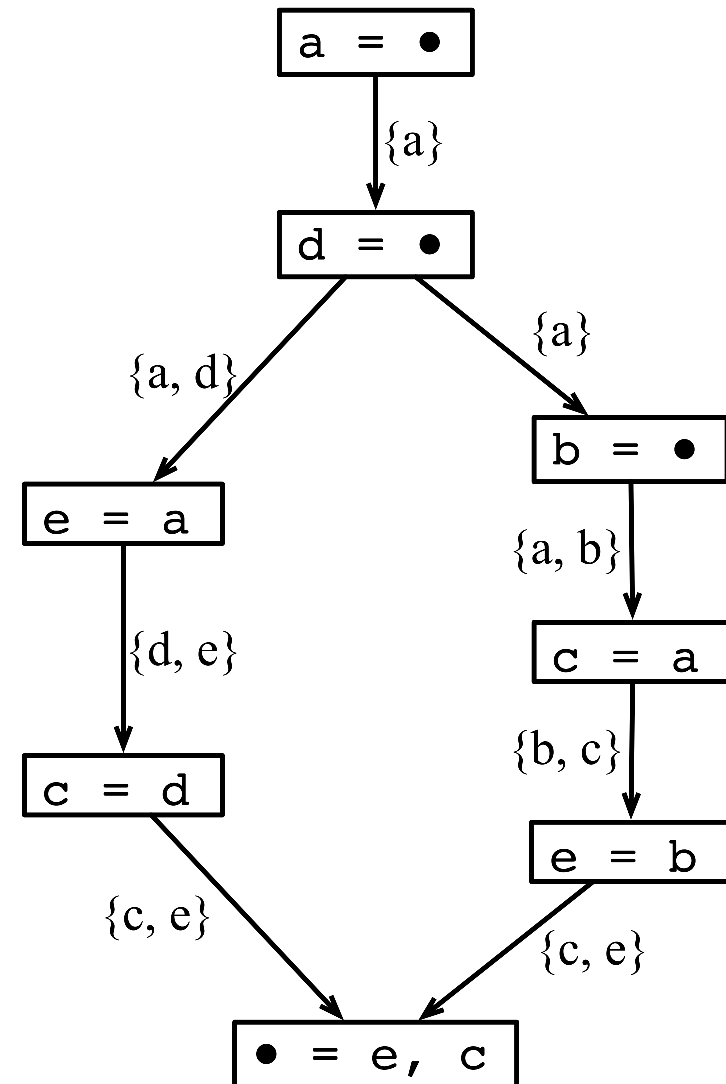
2) What is MinReg for this program?



PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DCC-UFMG

MaxLive

The maximum number of variables simultaneously alive at any program point is two in the example on the right. Therefore, MaxLive for this program is two.



2) What is MinReg for this program?

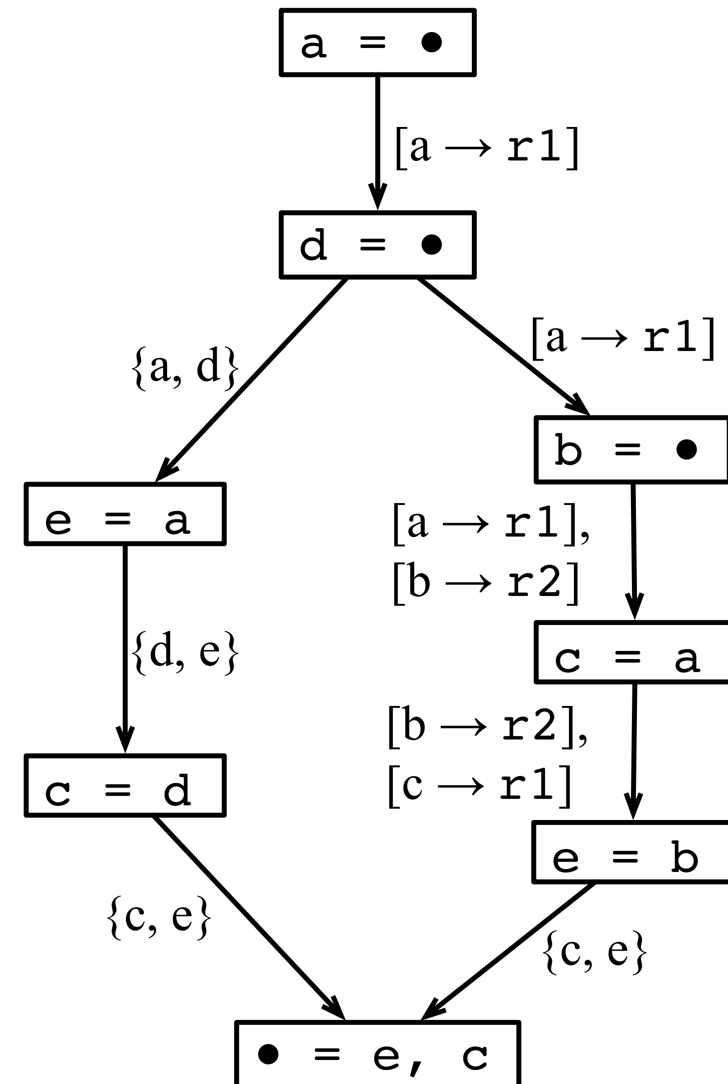
MaxLive

If we assume that a is assigned register r1, the only way that we can compile this program with two registers is assigning to c the same register.

1) Can you provide a reasoning on why this is true?

2) By choosing the register assigned to a, we automatically force the registers that must be assigned to e. Why is that so?

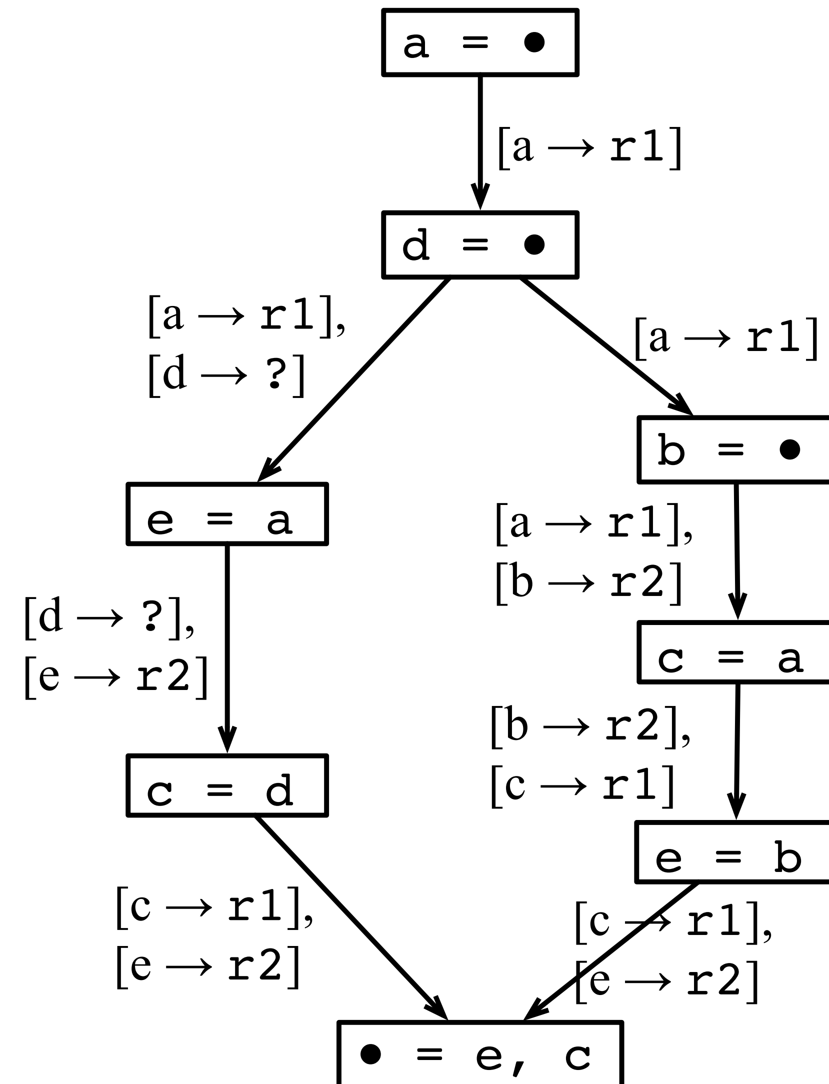
3) Can you finish the proof that we need at least three registers for this example?



MinReg > MaxLive

In the end, by naming the register of variable a, and trying to keep the MinReg ≤ 2, we end up in a situation in which both the variables that are simultaneously alive together with variable d are given different registers. E.g., a is given `r1`, and e is given `r2`. Therefore, we must necessarily pick a third register for d.

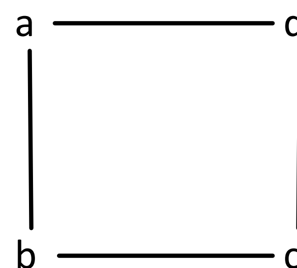
This example shows a program in which MinReg = MaxLive + 1. It is possible to build programs in which MinReg = MaxLive + n, for any n > 1.



Register Assignment is NP-Complete

Given a program P, and K general purpose registers, is there an assignment of the variables in P to registers, such that (i) every variable gets at least one register along its entire live range, and (ii) simultaneously live variables are given different registers?

- Gregory Chaitin has shown, in the early 80's, that the register assignment problem is NP-Complete♤.
- Chaitin's proof was based on a reduction between register assignment and graph-coloring.
  - We start with a graph that we want to paint with K colors, such that adjacent vertices get different colors.
  - From this graph, we build a program, that has a K-register assignment **if, and only if,** the graph has a K-coloring.



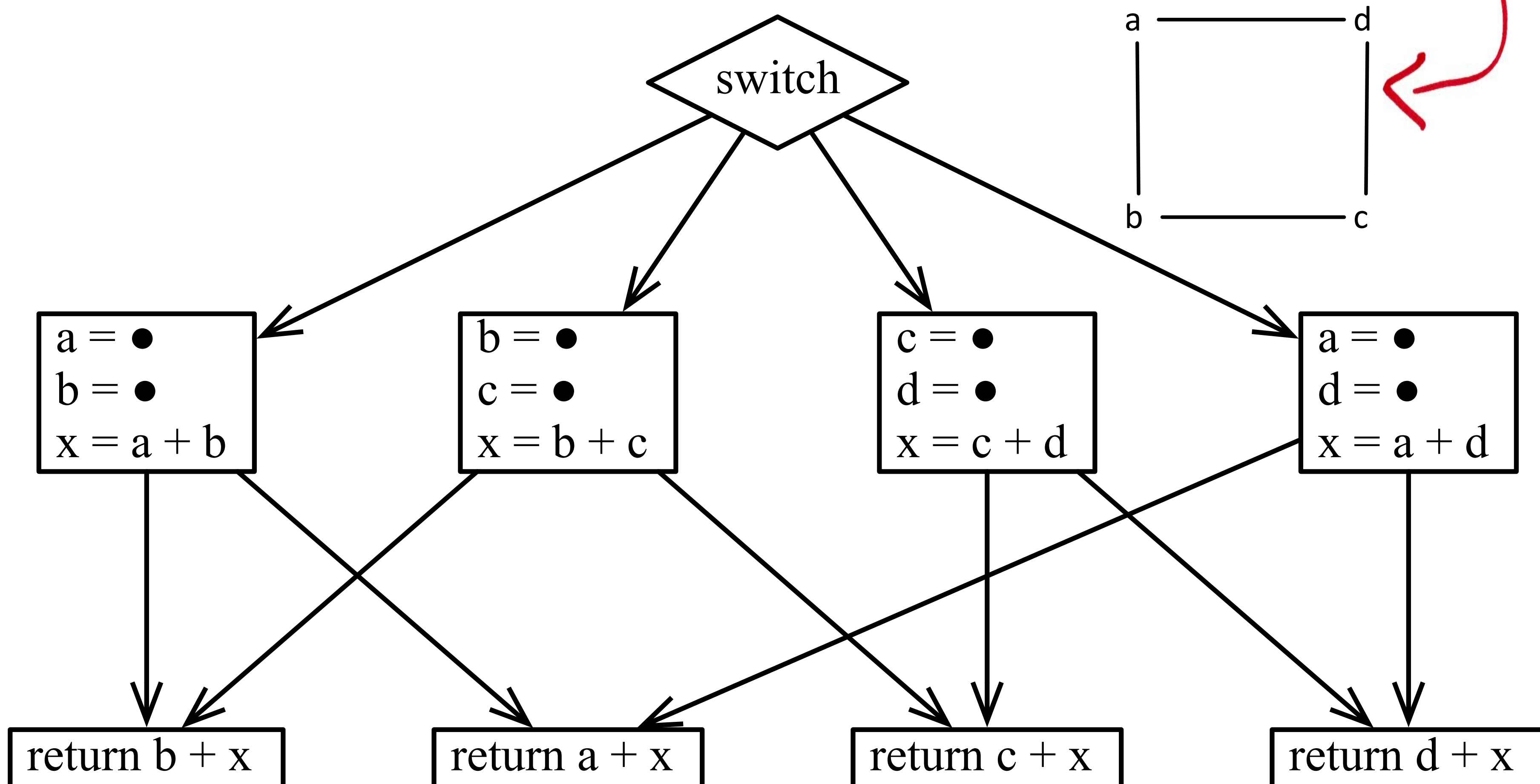
How the program of this graph would look like?

♤: Register allocation via coloring, 1981

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DCC - UFMG

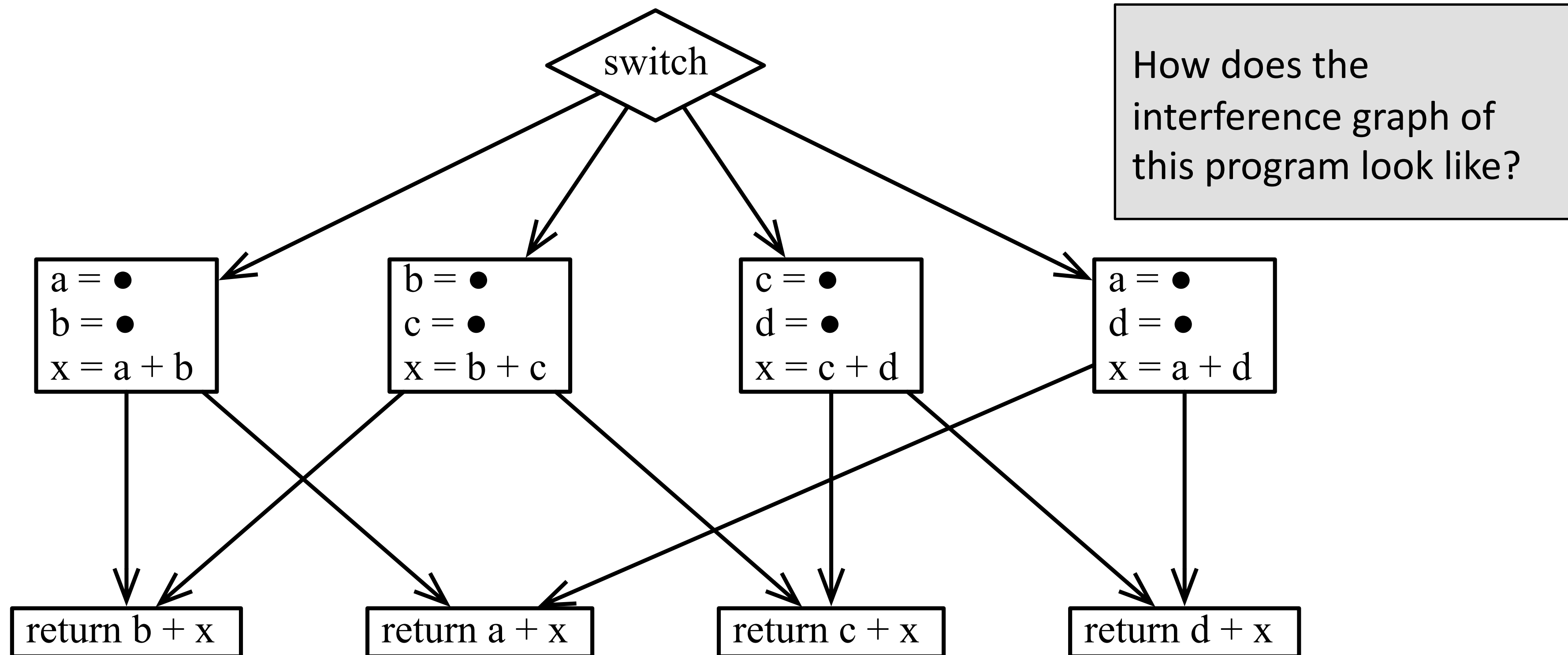
Chaitin's Proof

Chaitin's proof would create a program like this one below to represent C_4, the cycle with four elements. This program cannot be easily optimized by the compiler, and an allocation to it can be easily translated to a coloring of C_4.



Chaitin's Proof and the Interference Graph

The proof uses the notion of an *interference graph*. This graph has a vertex for each variable in the target program. Two variables are adjacent if their corresponding live ranges overlap. A coloring of the interference graph can be directly translated to an assignment of registers to variables, and vice-versa.

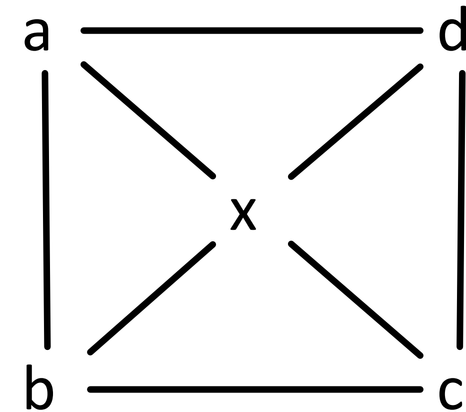


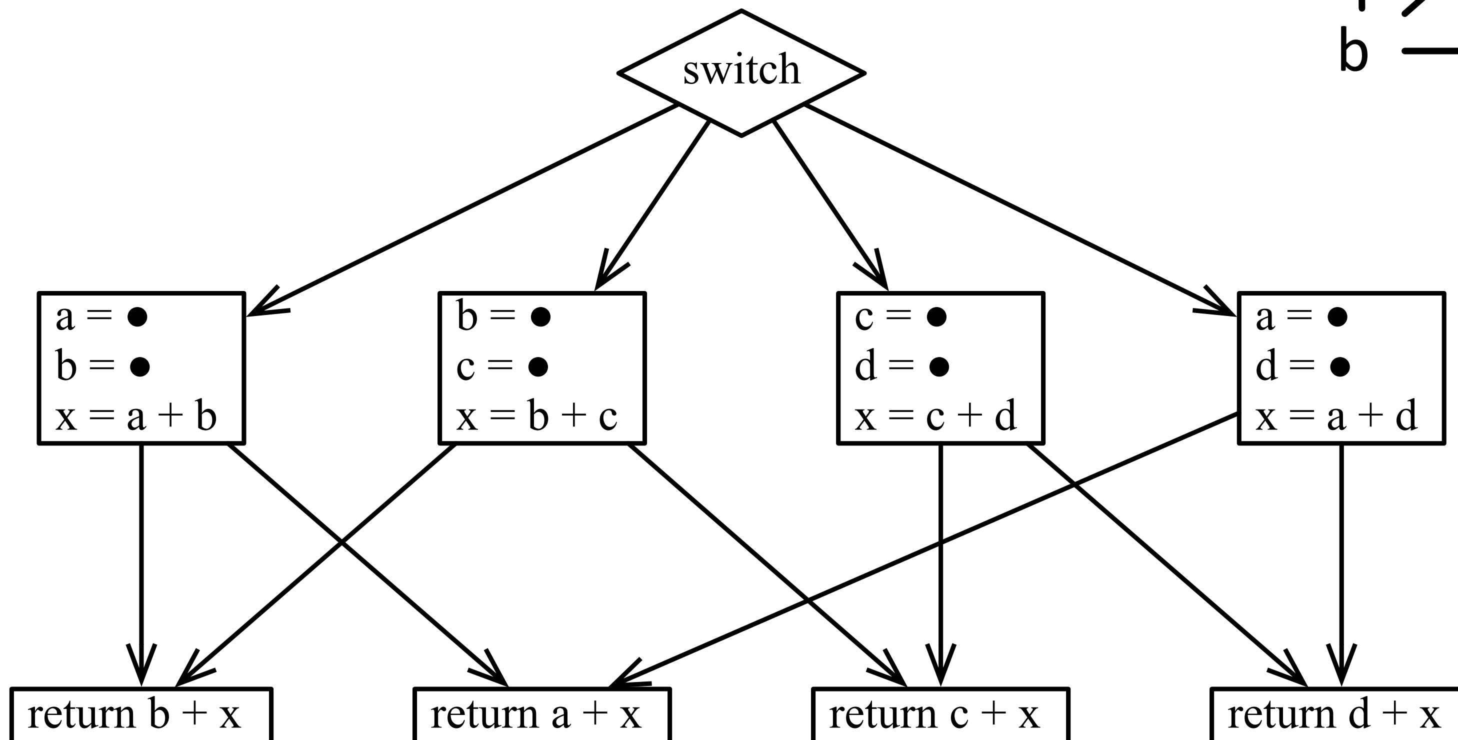
How does the interference graph of this program look like?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Chaitin's Proof and the Interference Graph

The interference graph is like C_4, but it has an additional vertex x that interferes with all the other vertices. Thus, the chromatic number of this graph is one more than the chromatic number of C_4.

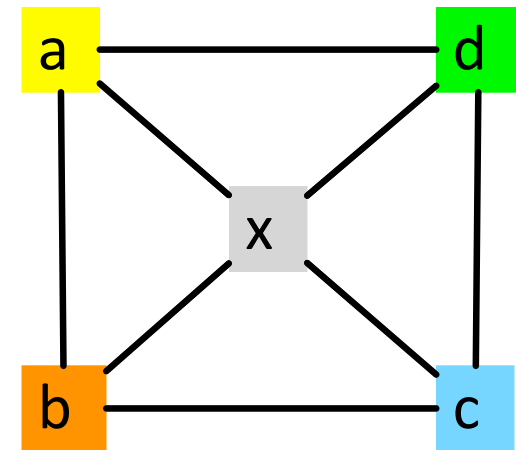


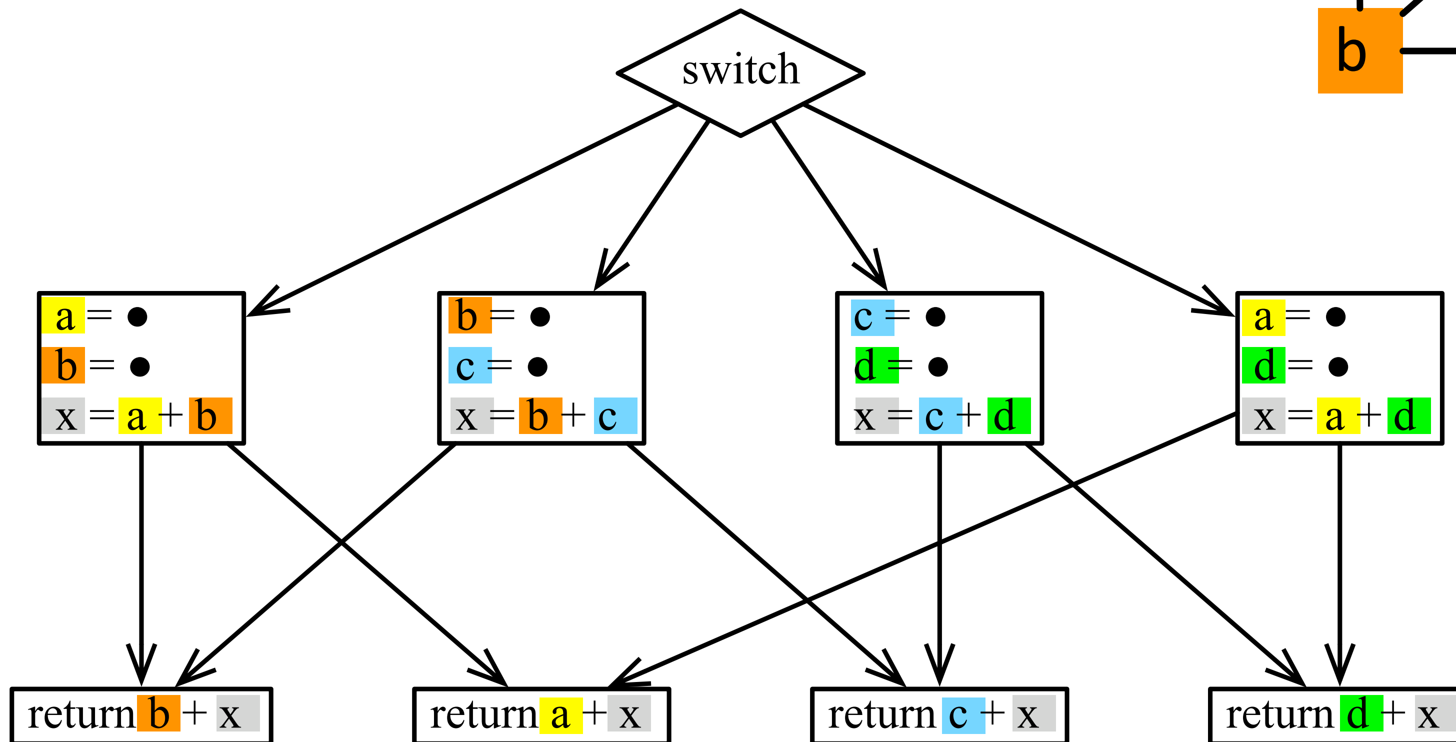


PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DCC - UFMG

Chaitin's Proof and the Interference Graph

A given graph G has a coloring with K colors, if, and only if, the equivalent Chaitin's program has an allocation with K + 1 registers.





LINEAR SCAN

DCC 888

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Linear Scan

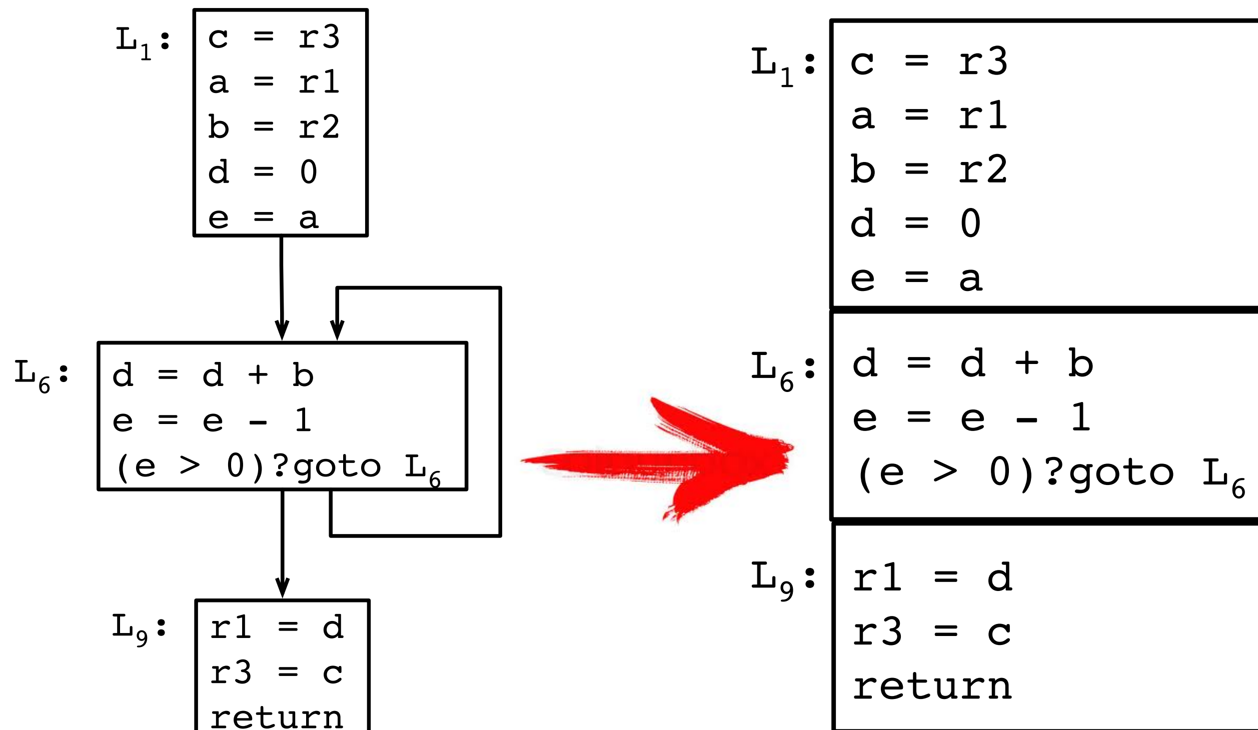
- Linear Scan, and its many variations, gives us one of the most popular register allocation algorithms in industrial quality compilers.
- It is based on the greedy coloring of interval graphs:
  - Given a sequence of intervals, we want to find the minimum number of colors necessary to paint them, so that overlapping intervals are given different colors.
  - There are optimal algorithms to solve this problem♤.
- Linear scan is not optimal, but it uses this optimal algorithm as an approximation of the register allocation problem.

How can we approximate register allocation as the coloring of interval graphs?

♤: Algorithmic Graph Theory and Perfect Graphs (2004)

Linearization of Basic Blocks

- The algorithm starts by arranging the program's basic blocks along a line.
  - There are many ways we can order these blocks, and some orderings will give us better results than others.
  - As a simplification, we can just sort the blocks in the reverse post-order of the program's Control Flow Graph. Usually this ordering is good.



1) Do you remember what is the reverse post ordering of a direct graph?

2) When have we used reverse post-order before?

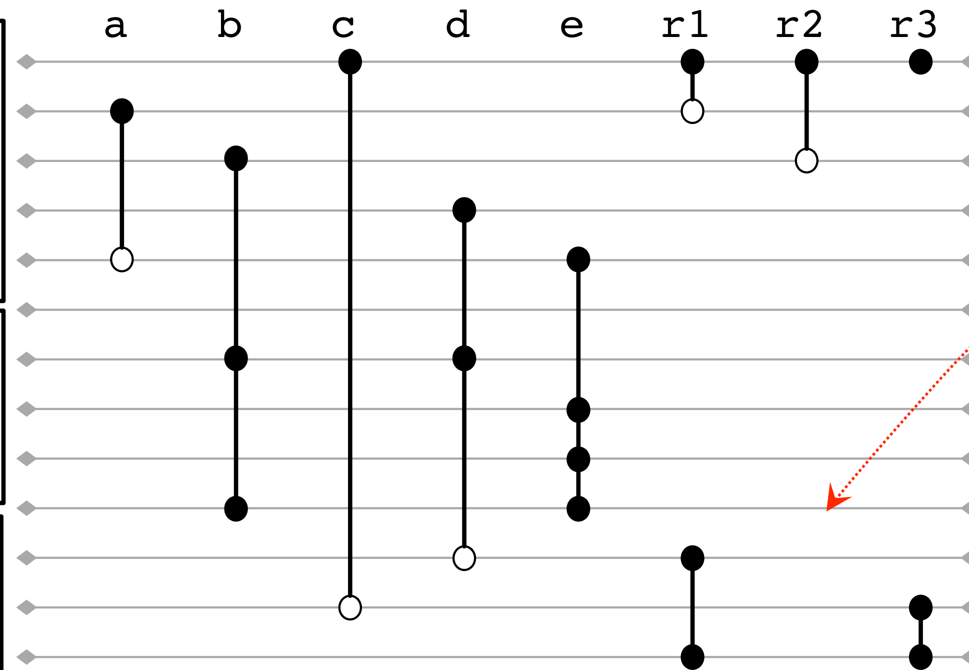
Finding Interval Lines

- Given an ordering of the basic blocks, we create intervals associated with each variable name.
  - An interval I_v for a variable v starts at the first program point where v is alive.
  - The interval I_v ends at the last program point where v is alive.

```
c = r3
a = r1
b = r2
d = 0
e = a
```

```
d = d + b
e = e - 1
(e > 0)?goto L6
```

```
r1 = d
r3 = c
return
```



Why do the intervals of b and e end at **this** point, and not at the last use sites?

Linear Scanning of Intervals

```
LINEARSCANREGISTERALLOCATION♣
  active = {}
  foreach interval i, in order of increasing start point
    EXPIREOLDINTERVALS(i)
    if length(active) = R then
      SPILLATINTERVAL(i)
    else
      register[i] = a register removed from the pool of free registers.
      Add i to active, sorted by increasing end point
```

This is the very algorithm that we had seen in our second class, when we talked about local register allocation.

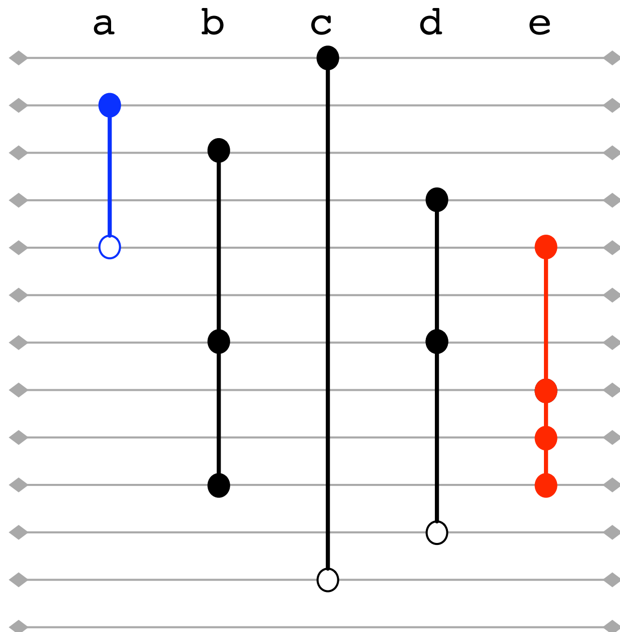
1) Can you guess what ExpireOld Interval is doing?

2) And what is the method SpillAt Interval doing?

♣: code taken from "Linear Scan Register Allocation", by Poletto and Sarkar, 1999

Expiring Old Intervals and Freeing Registers

```
EXPIREOLDINTERVALS(i)
  foreach interval j in active, in order of increasing end point
    if endpoint[j] ≥ startpoint[i] then
      return
    remove j from active
    add register[j] to pool of free registers
```



If we are analyzing the interval that corresponds to variable e, e.g., I[e], then we see that the end point of I[a] ≤ start point of I[e]. Therefore, we no longer need to consider I[a] in our allocation. This interval is behind the *scanning line*.

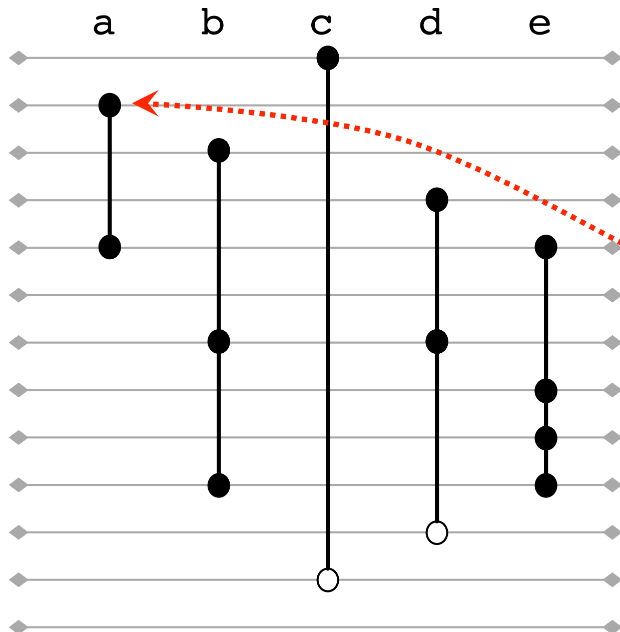
As a consequence, we can give the register previously used to allocate "a" back to the set of available registers.

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Spilling

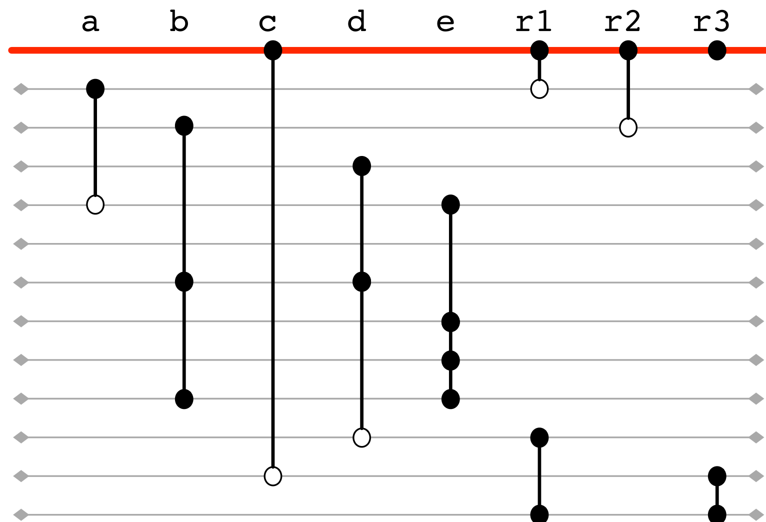
SPILLATINTERVAL(i)
spill = last interval in active
register[i] = *register*[*spill*]
location[*spill*] = new stack location
remove *spill* from *active*
add i to *active*, sorted by increasing end point

1) What is the complexity of finding **this** interval?
2) This heuristic has some deficiencies. Can you name one?

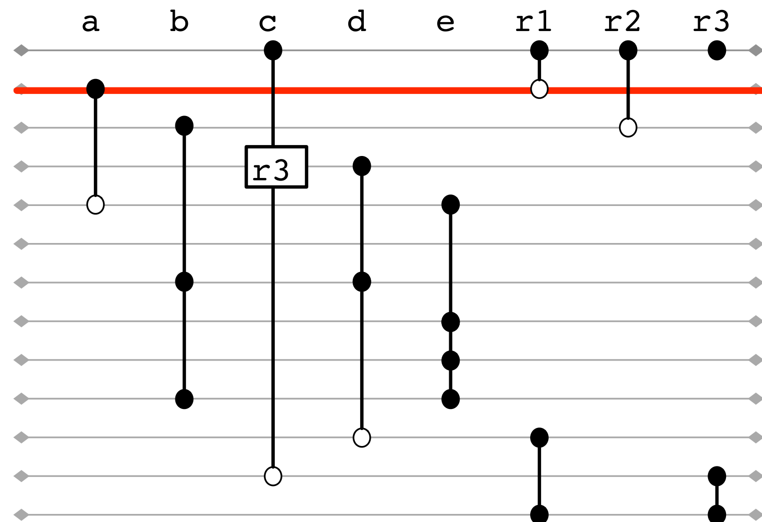


The original description of the linear scan register allocator spills the interval that has the farthest end point from the spilling point.

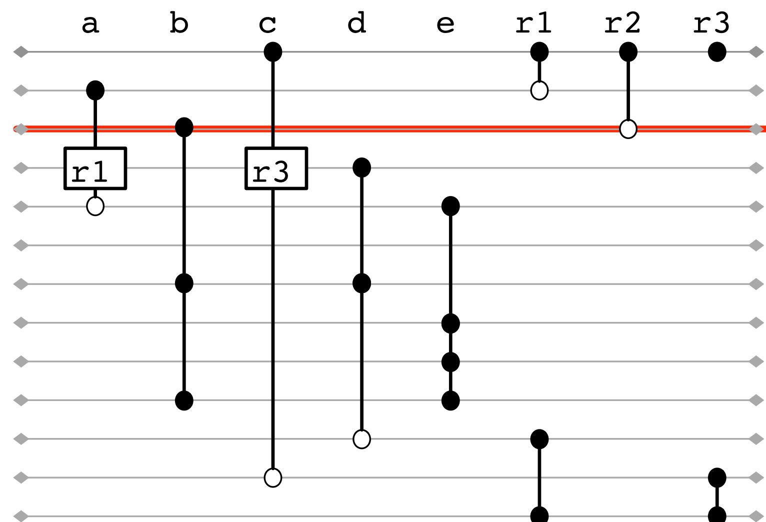
Continuing with our example, if we were to spill at **this** point, we would have to evict the interval associated with variable c. This interval is the active interval which has the farthest end point.



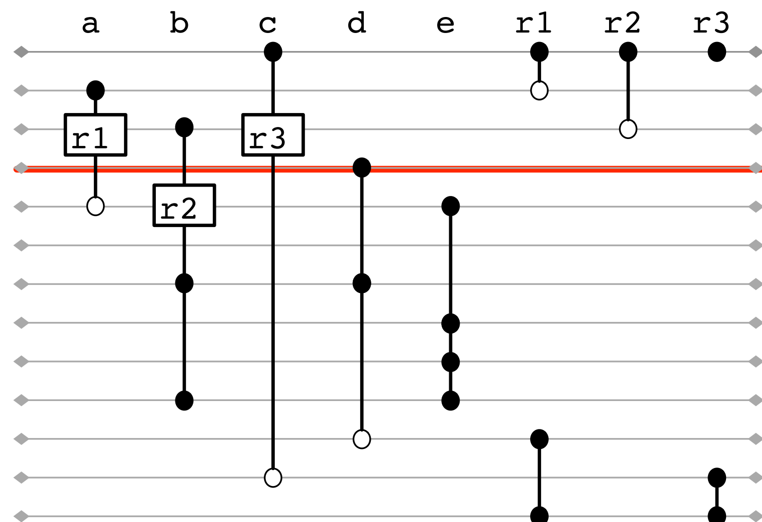
Assign r3 to c, as this is the only free register that we have at this point



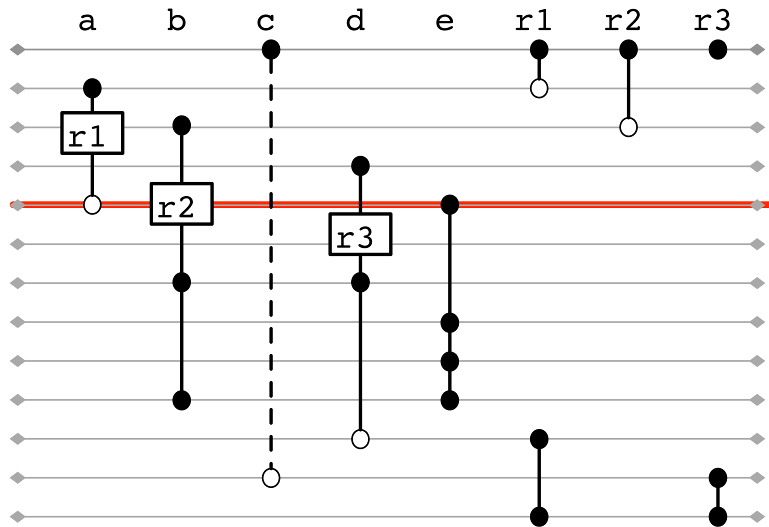
Put r1 back into the list of free registers, and assign it to variable a, as this is the only register that is free.



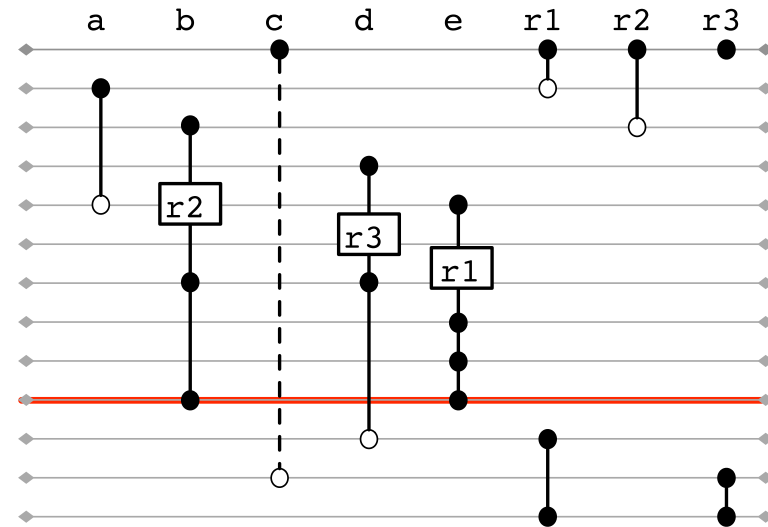
Put r2 back into the list of free registers, and assign it to variable b, as this is the only register that is free.



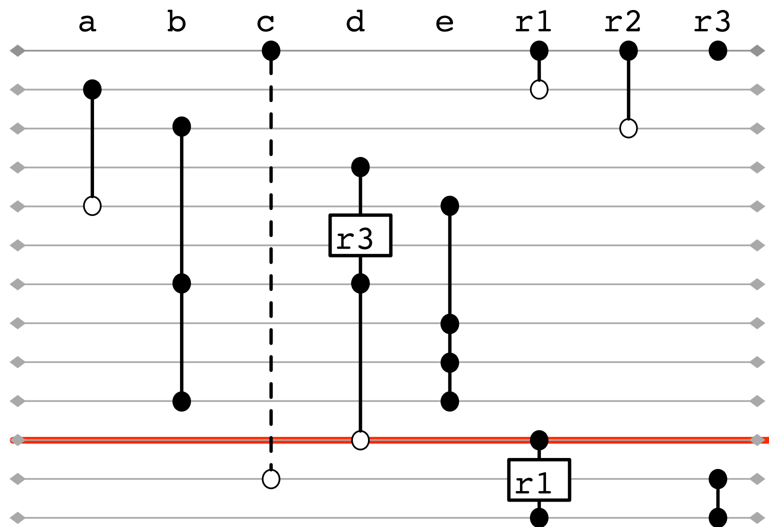
There is no free register, and there is no interval that we can remove from the list of active intervals. We need to spill c, as its interval has the farthest end point.



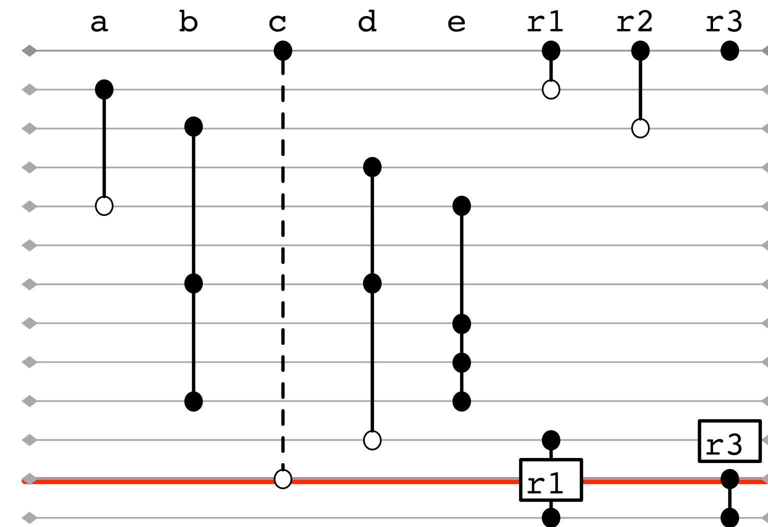
Remove a from the list of active intervals, and move its register to e, which is now an active interval.



Remove b and e from the list of active intervals, and add r1 and r2 to the pool of free registers



Remove d from the list of active intervals, free its register, r1, but remove it also from the list of free registers, due to an architectural constraint.



Add c back to the list of active intervals, and assign it to r2, the only free register available, and we are done!

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

What Happens with the Intervals Once we Spill?

- Every variable needs to be used in a register. Hence, if we spill an interval, we still need to find registers for its uses.
- When we spill an interval, we remove it from the list of available intervals, but we then add a new interval to each use of the spilled variable that has not been seen yet.
- These new intervals are very short: they only exist at a program point, and they will ensure that in the end we have a place to load a variable from memory, and a source location for the stores.
- This approach is called "spill everywhere", because it causes us to insert loads and stores to every use of a spilled variable that has *not been seen by the scanning line*.

What do I mean by "not been seen by the scanning line"?

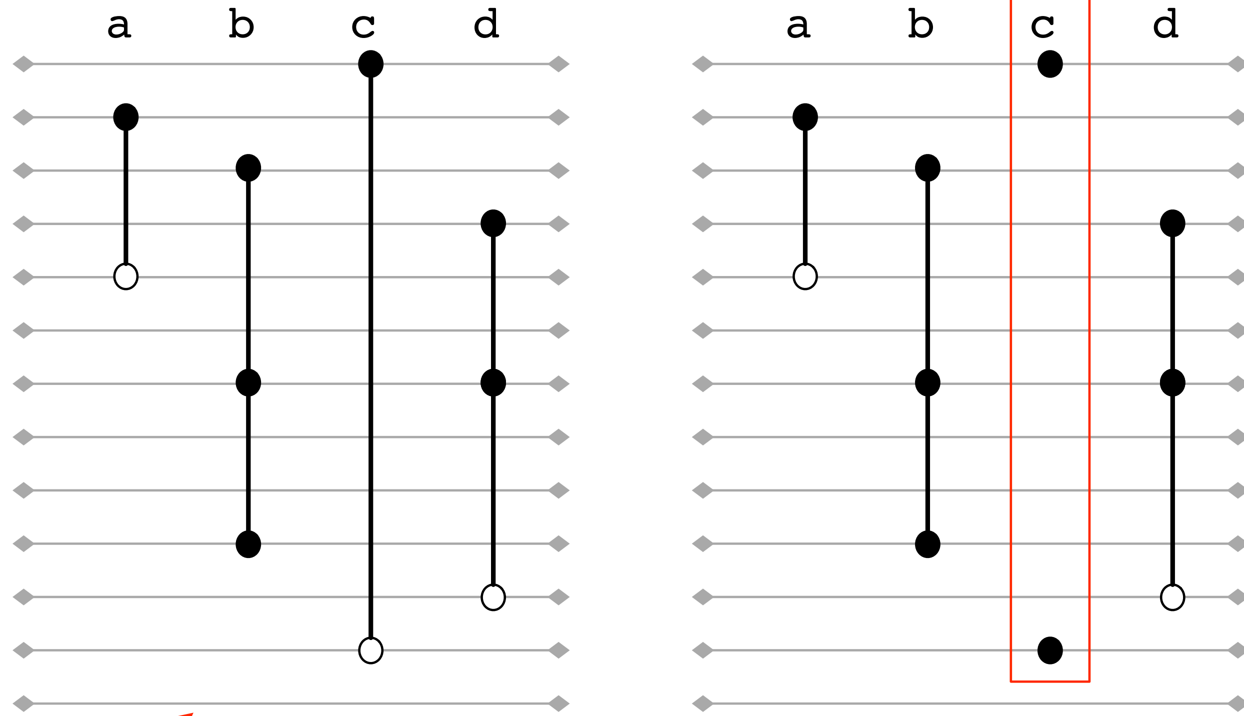
PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DCC - UFMG

What Happens with the Intervals Once we Spill?

```
c = r3
a = r1
b = r2
d = 0
e = a
```

```
d = d + b
e = e - 1
(e > 0)?goto L6
```

```
r1 = d
r3 = c
return
```



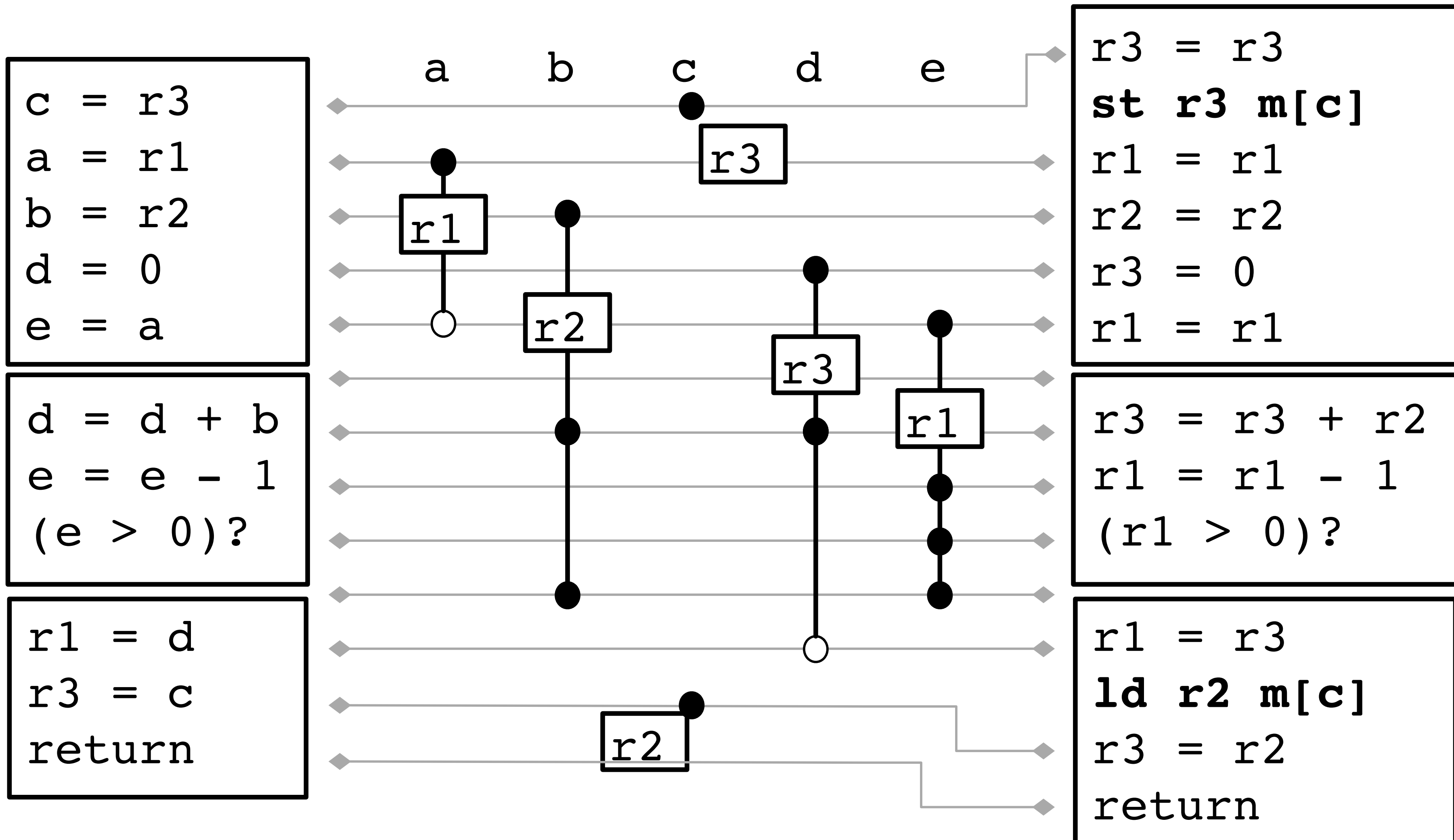
The grid on the **left** shows the intervals before the spilling took place. On the **right** we have the intervals after spilling. We have replaced the interval of c by two new micro intervals, which exist only at a single program point. Notice that these two intervals are unrelated, in the sense that they do not need to be given the same register.

Store/Load Placement

- The allocation gives us a guide to sow loads and stores in the code; hence, ensuring the correct mapping between variables and memory slots.
- We can place these memory access instructions in the code while we scan the interval lines.
  - If we spill, then we add a store to the definition of the variable, where the target of the store is the register originally given to the variable.
  - If we find the micro-interval of a spilled variable, then we add a load before its use, where the target of the load is the newly found register for that interval.

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Store/Load Placement



Coalescing

1) Take a further look into our final program. How could we optimize it?

2) Could this optimization be done as part of register allocation?

3) Has our register allocator being designed to gives us more opportunities to perform this optimization?

```
r3 = r3
st r3 m[c]
r1 = r1
r2 = r2
r3 = 0
r1 = r1
```

```
r3 = r3 + r2
r1 = r1 - 1
(r1 > 0)?
```

```
r1 = r3
ld r2 m[c]
r3 = r2
return
```

Coalescing

- If two variables related by a move instruction are given the same register, then we can remove this move instruction.
- This optimization is called coalescing.
- Our simple implementation of linear scan is not tailored to gives us more opportunities to do coalescing.

```
r3 = r3
st r3 m[c]
r1 = r1
r2 = r2
r3 = 0
r1 = r1
```

```
r3 = r3 + r2
r1 = r1 - 1
(r1 > 0)?
```

```
r1 = r3
ld r2 m[c]
r3 = r2
return
```

> How could we change our implementation of linear scan to do some biased register assignment, so that move related variables have a greater chance of receiving the same register?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DCC - UFMG

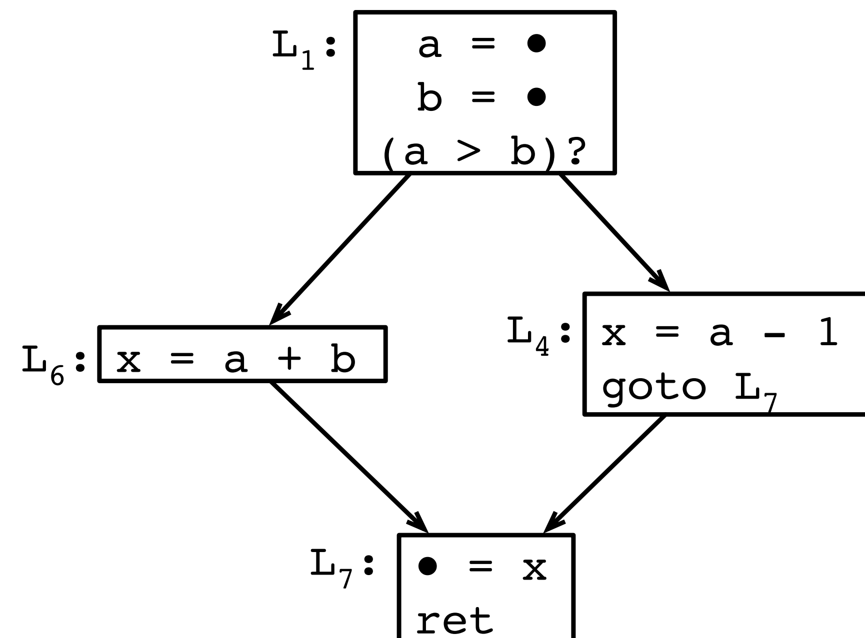
Coalescing

```
LINEARSCANREGISTERALLOCATIONWITHCOALESCING
  active = {}
  foreach interval i, in order of increasing start point
    EXPIREOLDINTERVALS(i)
    if length(active) = R then
      SPILLATINTERVAL(i)
    else
      if i is "a = b" and register[b] ∈ free registers
        register[i] = register[b]
        remove register[b] from the list of free registers
      else
        register[i] = a register removed from the list of free registers
      add i to active, sorted by increasing end point
```

Live Ranges with Holes

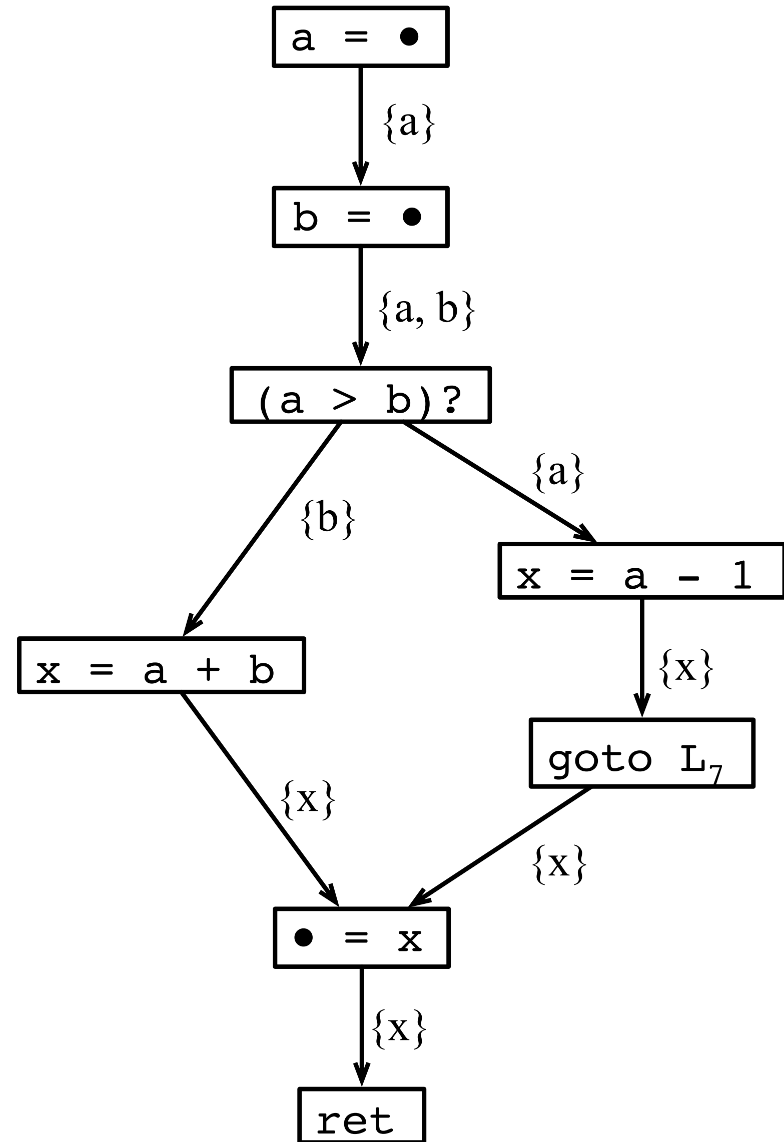
- Linear scan is suboptimal, i.e., its results are not guaranteed to be optimal, neither in terms of minimum register assignment, coalescing nor spilling.
- One of the most serious deficiencies of Linear Scan is related to the way it handles holes in the live ranges.

Can you run liveness analysis on the program, to see which variables are alive at each program point?



Line Ranges with Holes

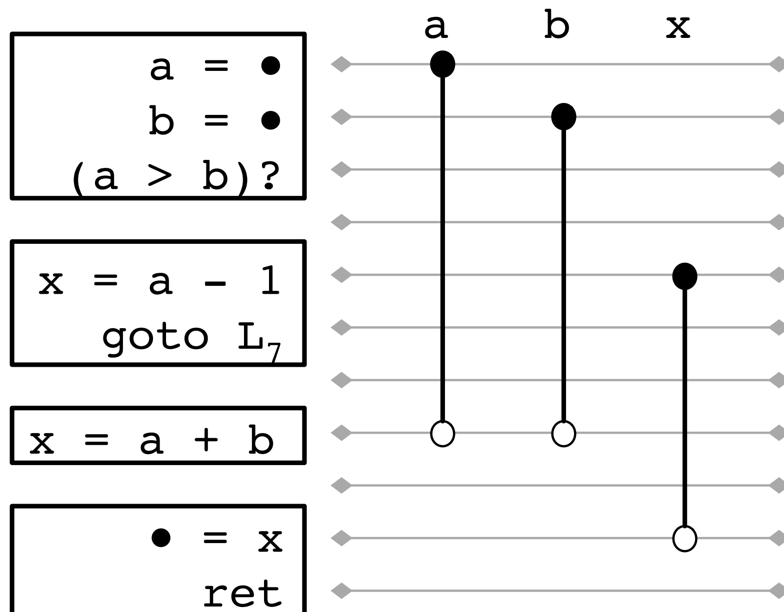
If we linearize the program on the right by sorting the labels, how would the intervals that we obtain look like?

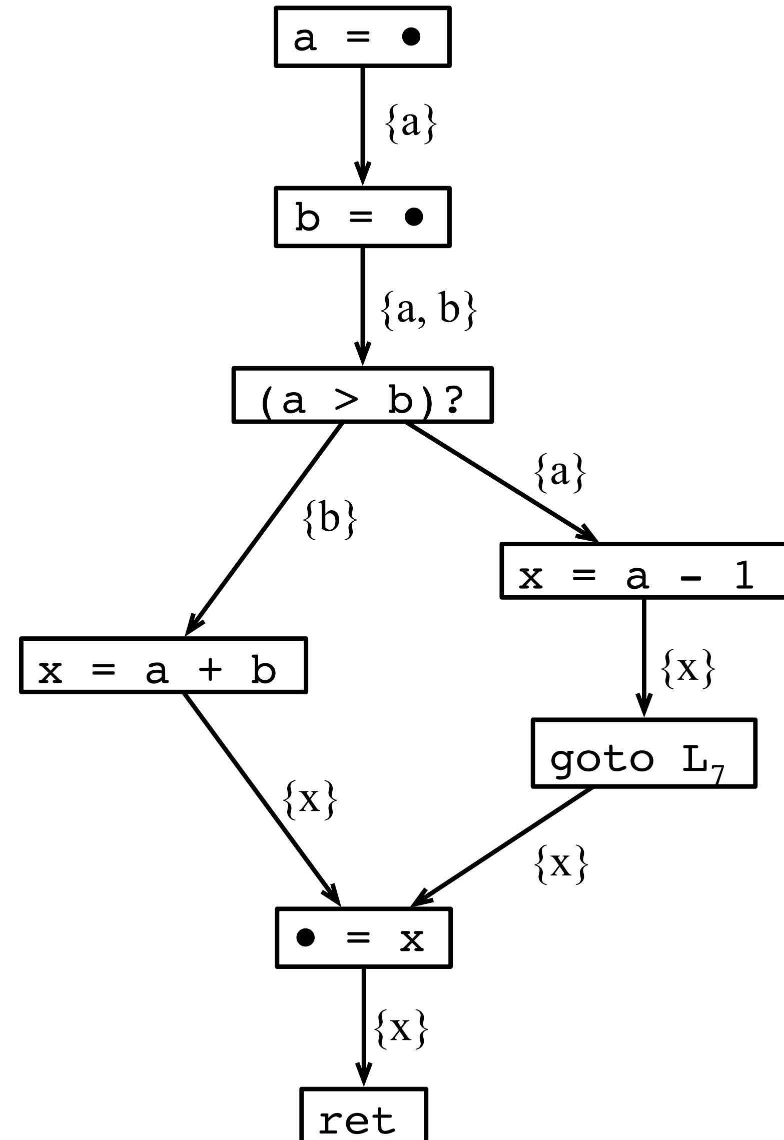


Holes in Live Ranges

1) In the actual program, can x share a register with either a or b?

2) What is the view that we give to linear scan?



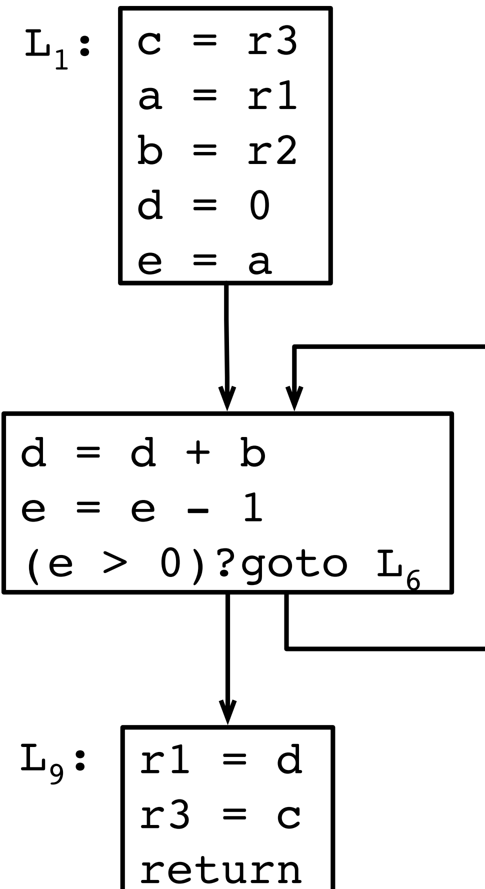


GRAPH COLORING REGISTER ALLOCATION

DCC 888

The Interference Graph

- Possibly the most well-known register allocation family of algorithms is based on the notion of an interference graph.
- The interference graph is the intersection graph of the live ranges of variables on the Control Flow Graph.
  - We have one vertex for each variable.
  - Two vertices are adjacent if, and only if, their corresponding live ranges overlap.



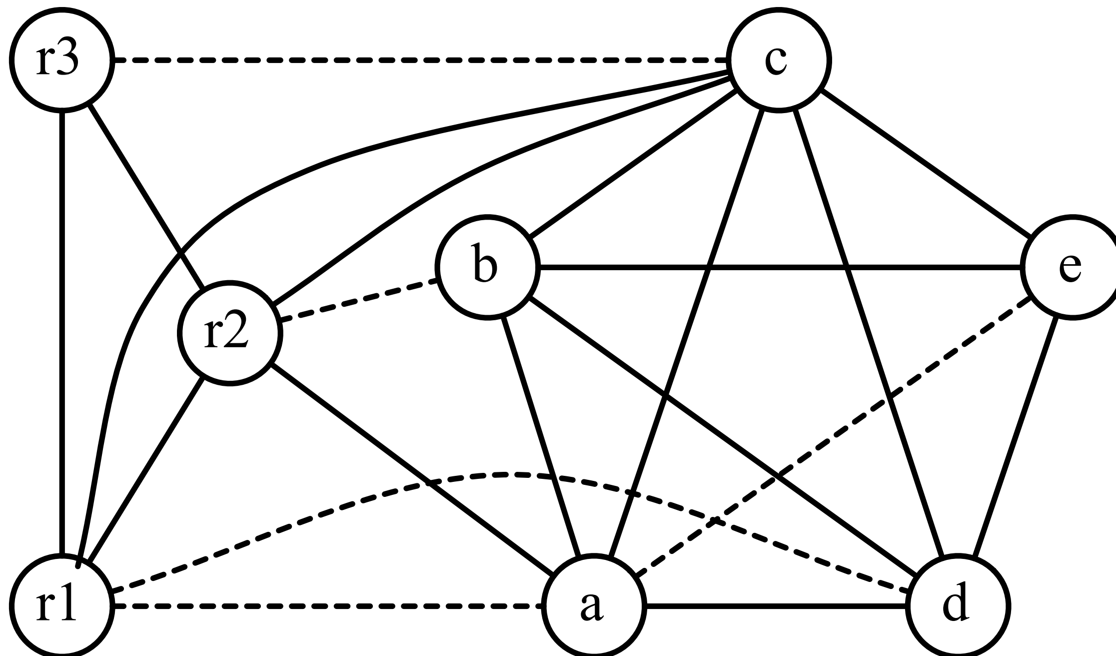
Gregory Chaitin, the father of register allocation via graph coloring.

1) How is the interference graph of the program on the right?
2) How many vertices does it have?
3) How many edges?

The Interference Graph

Can you give me a heuristics to know if we can color a graph with K colors?

In addition to the interference edges (solid) we will also show *coalescing edges*. These edges connect two variables if, and only if, they are related by move instructions. In the figure, these edges are dashed.




```
L1:  c = r3
     a = r1
     b = r2
     d = 0
     e = a

L6:  d = d + b
     e = e - 1
     (e > 0)?goto L6

L9:  r1 = d
     r3 = c
     return
```

Kempe's Simplification Algorithm

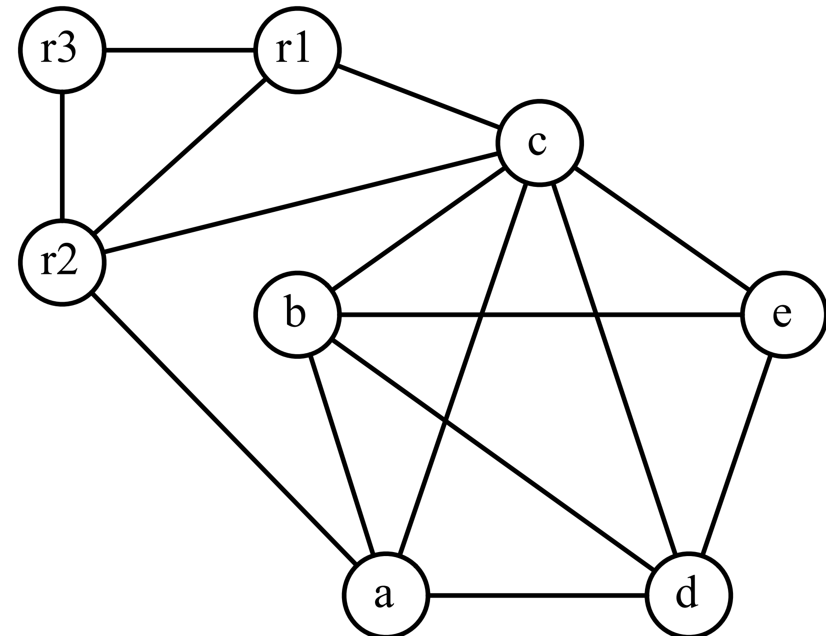
Suppose the graph contains a node m with fewer than K neighbors. Let G' be G \ {m}. *If G' can be colored, then G can be colored as well*.

1) Why is this *fact* true?

3) Can you apply Kempe's heuristics to check if the graph below if 4-colorable?

Thus, we can remove low-degree nodes, until either we *find only high-degree nodes*, or we remove all the nodes in the graph. In the latter case, the graph is K-colorable♧.

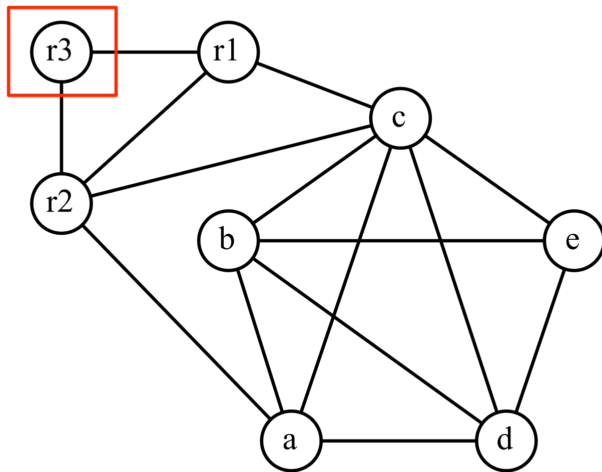
2) Can we say that in the *former case* the graph is not K colorable?

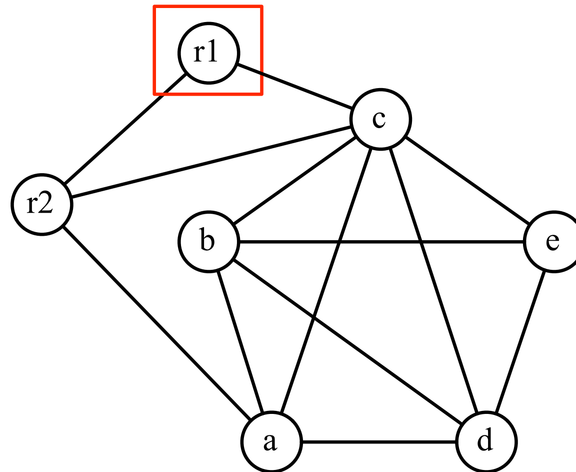


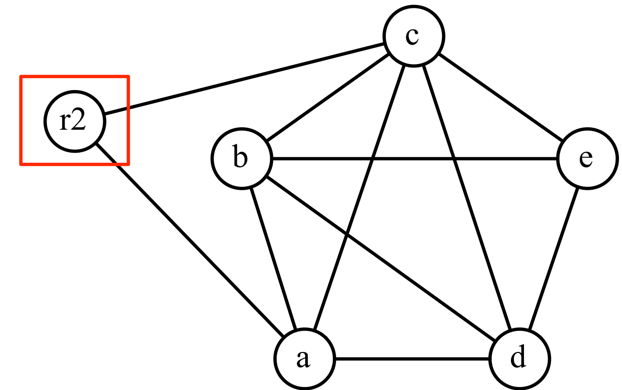
This heuristics is known as Kempe's heuristics, after a paper by A. Kempe, "On the geographical problem of the four colors" (1879)

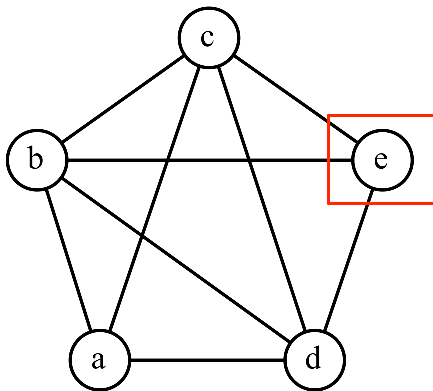
Kempe's Simplification

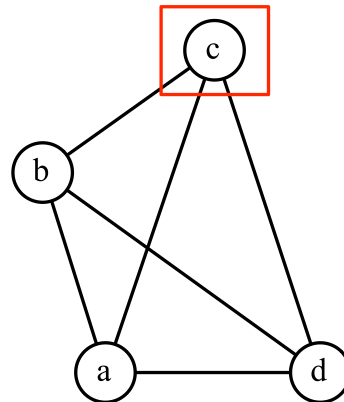
Ok, the graph is four colorable. How can we color it?

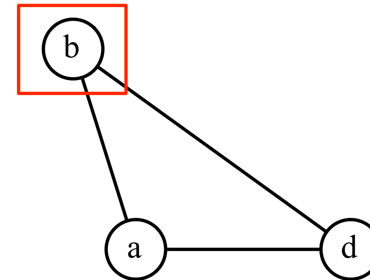


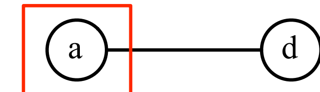


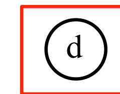






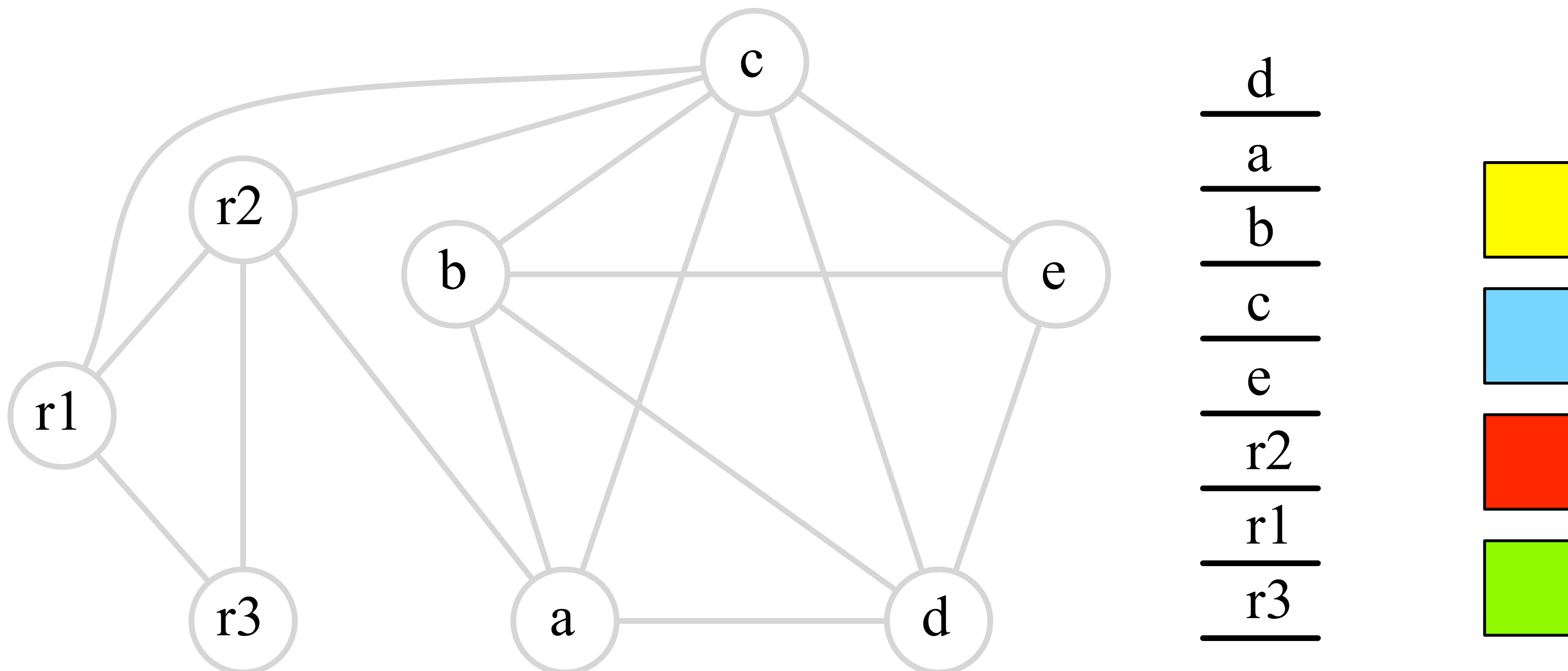






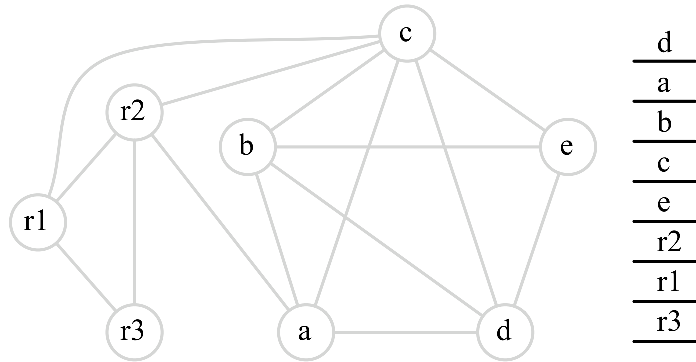
Greedy Coloring

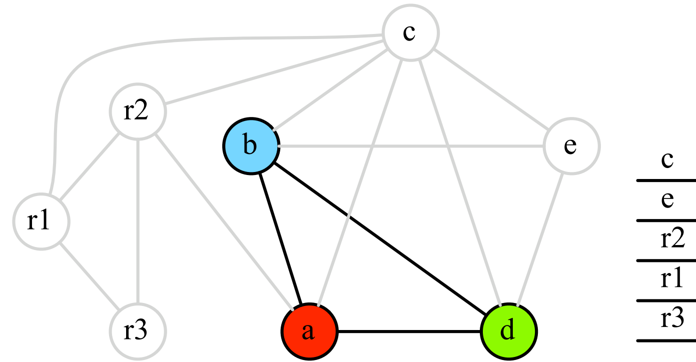
- We can assign colors to the nodes greedily, in the reverse order in which nodes are removed from the graph.
- The color of the next node is the first color that is available, i.e., is not used by any neighbor.



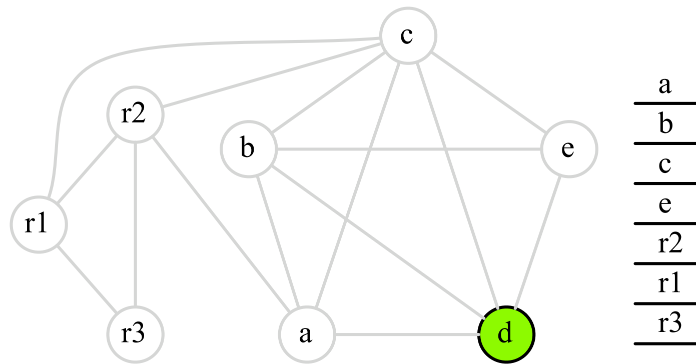
PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO
EM CIÊNCIA DA COMPUTAÇÃO
DCC–UFMG

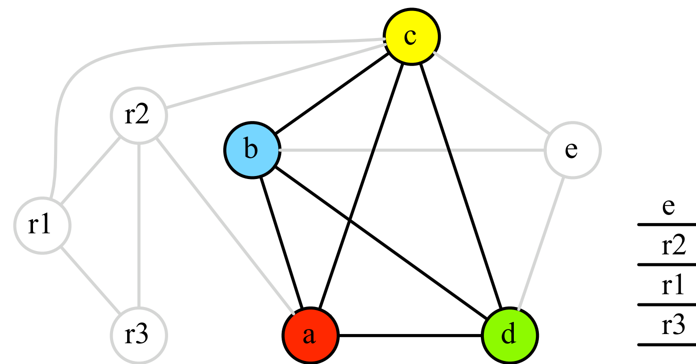
Greedy Coloring

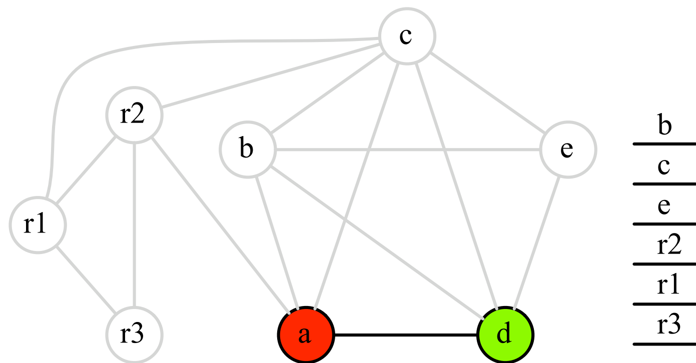


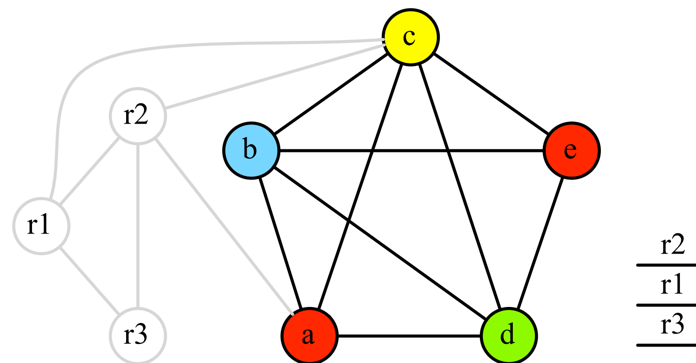


1
2
3
4









PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DCC-UFMG

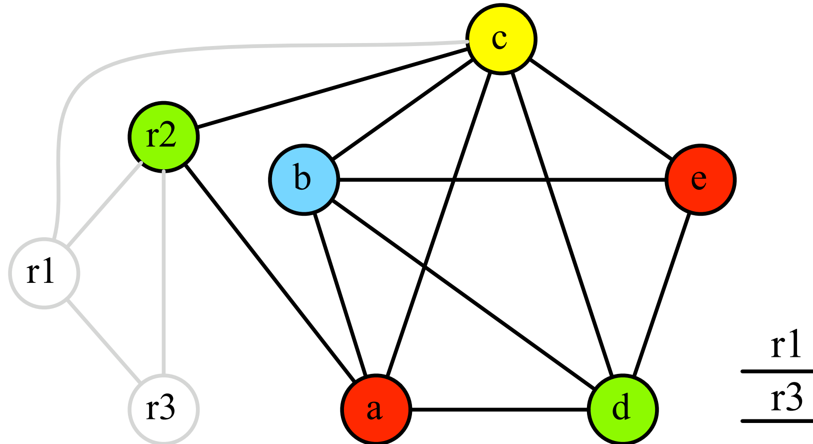
Greedy Coloring

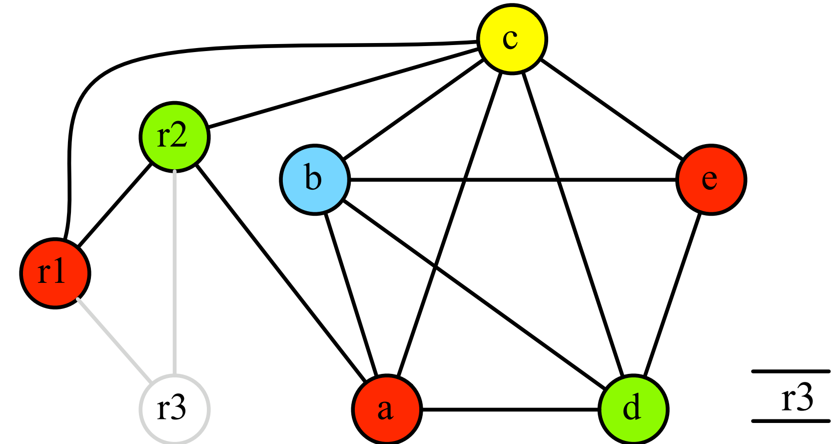
1

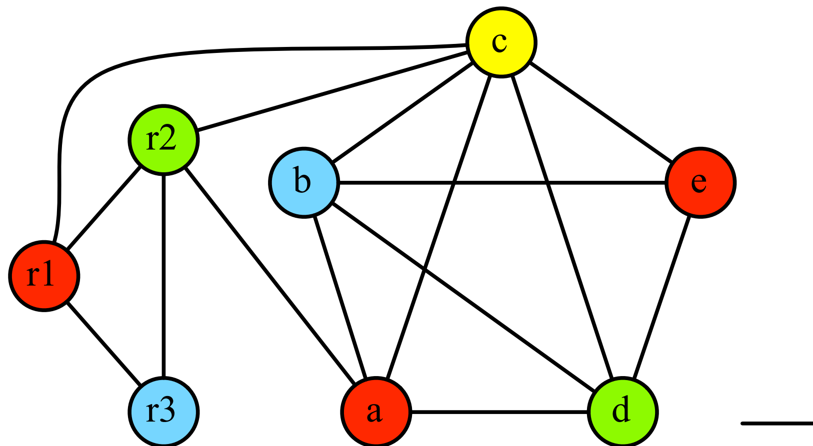
2

3

4







Notice that this heuristics does not try to find the minimum number of colors (chromatic number of a graph). It only wants to find if the graph is K colorable.

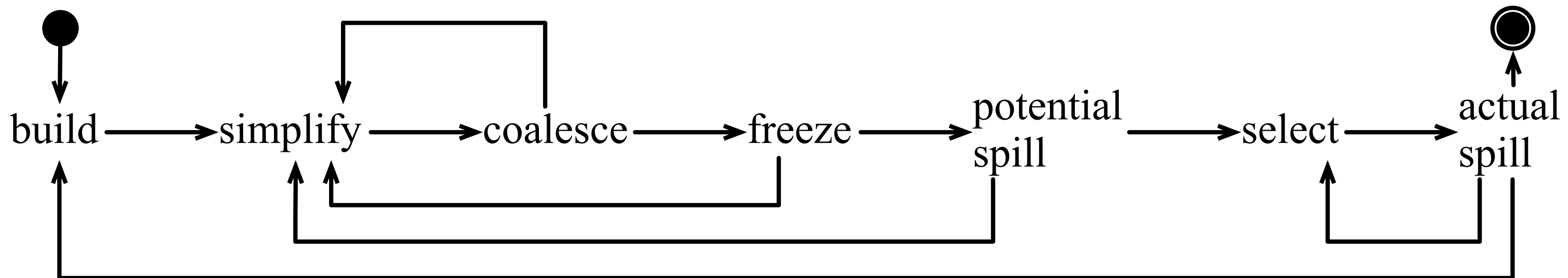
By the way, what is the complexity of Kempe's heuristics?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Iterated Register Coalescing

- Now we have a heuristics to color the graphs, but we need to deal with the other aspects of register allocation:
  - Coalescing
  - Spilling
- Henceforth, we will be using an algorithm called Iterated Register Coalescing◊.

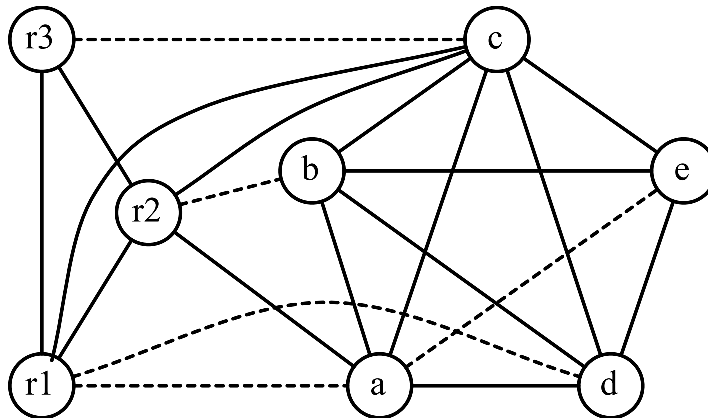
What do you think each of these phases is all about?

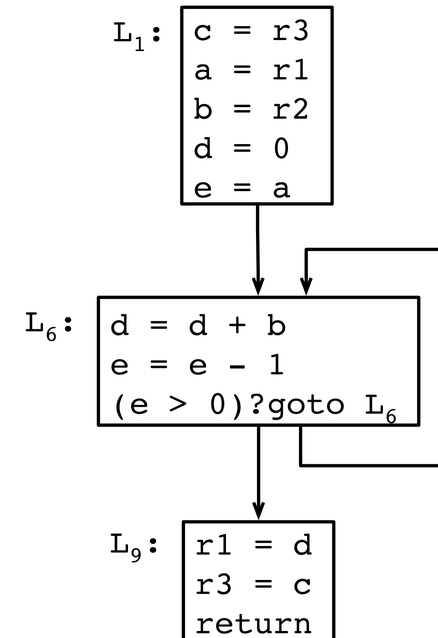


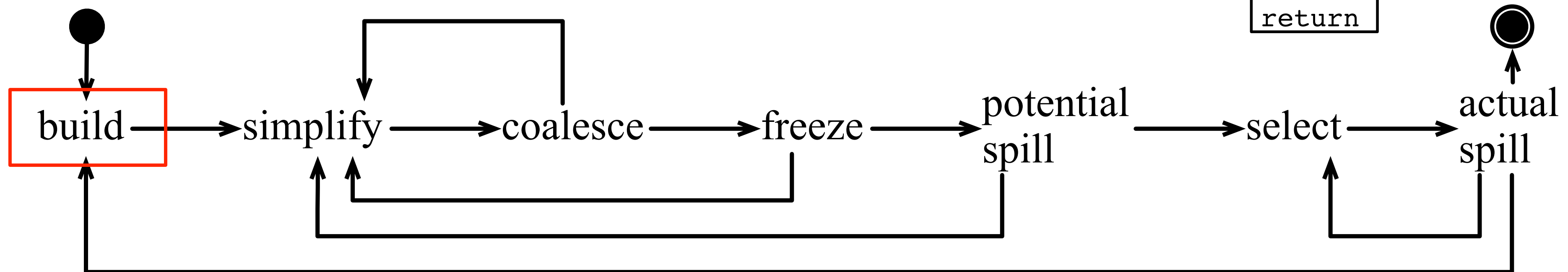
◊: Iterated Register Coalescing, TOPLAS (1996)

Build

- In the build phase we produce an interference graph out of liveness analysis.



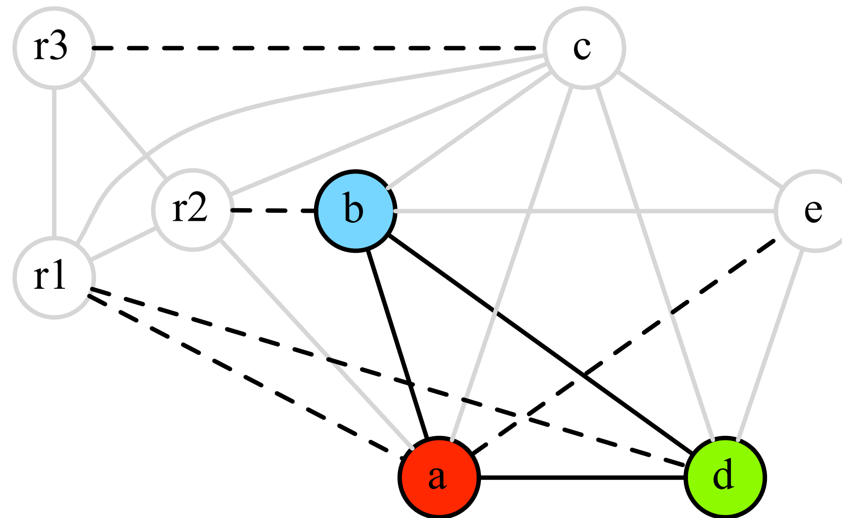


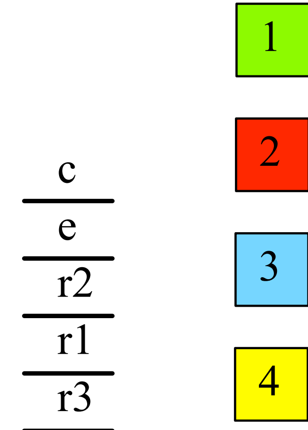


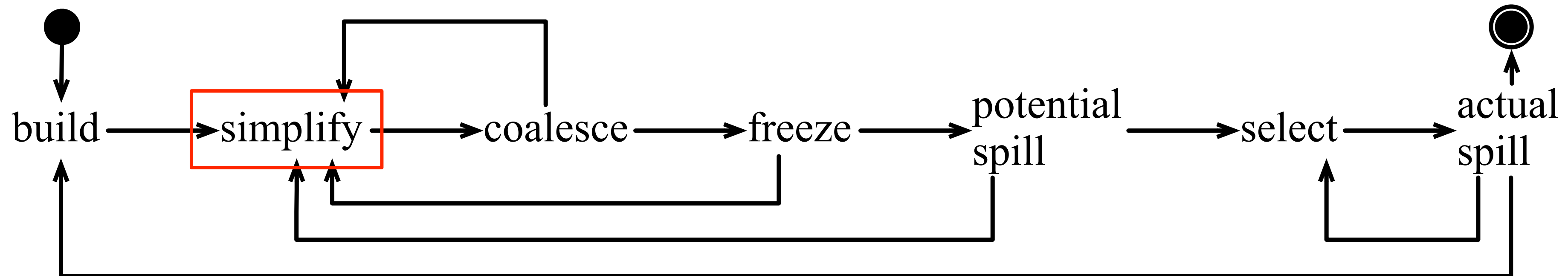
Simplify

- In the Simplify phase, we use Kempe's heuristics to try to remove non-move related nodes from the graph.

1) Why we should not simplify move related nodes?
2) Can we simplify any node in this graph?



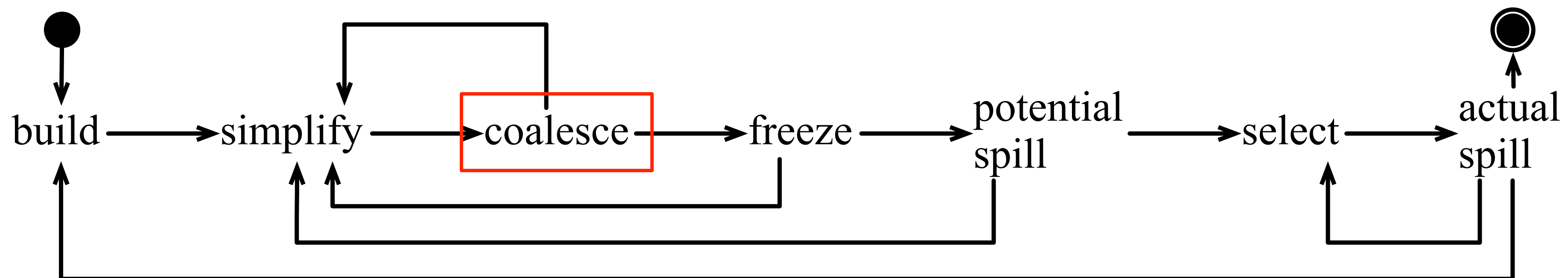




Coalesce

- After we simplify a node, we may have more opportunities for coalescing: *the resulting graph is a bit smaller*.
- We can alternate simplification and coalescing until only nodes of high degree or related by moves remain in the graph.

Why *this* implication holds will be clear soon (hopefully)

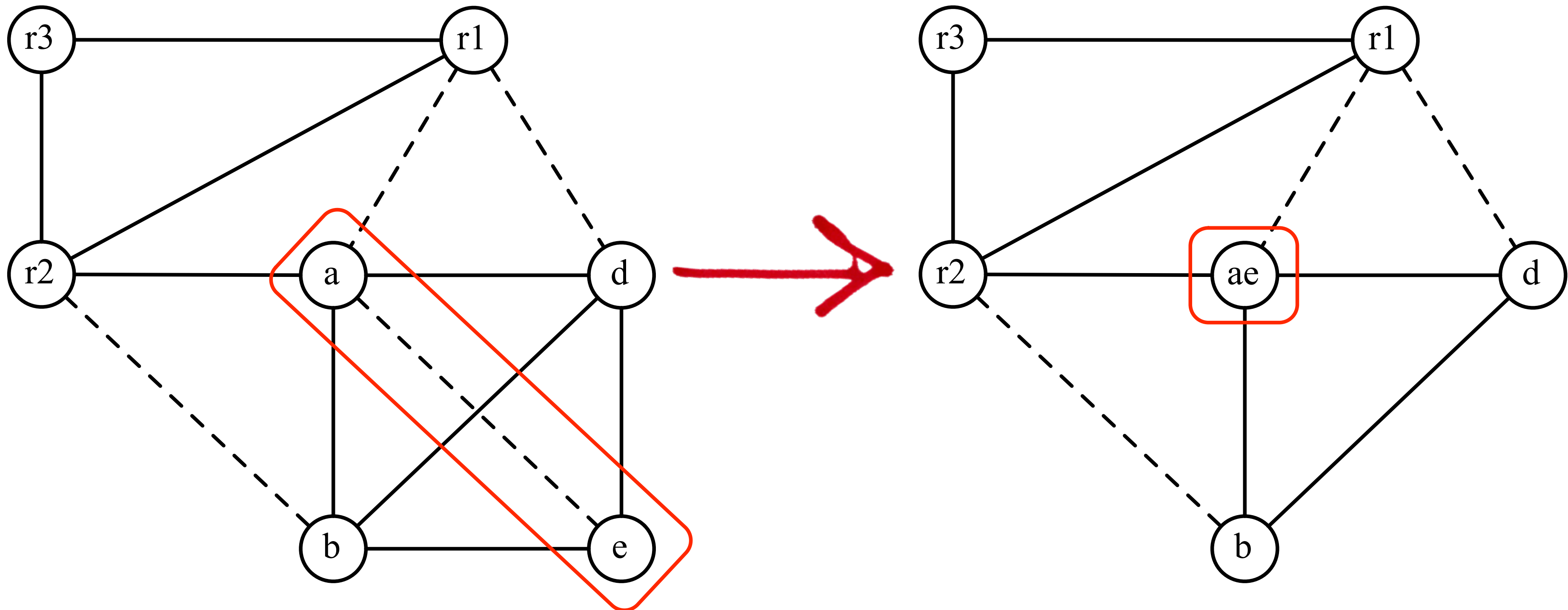


Coalescing

- Coalescing consists of collapsing two move related nodes together.

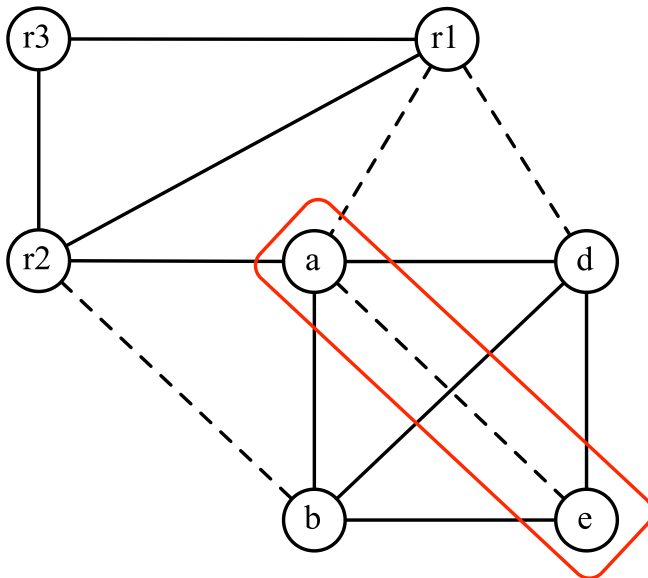
1) When can coalescing cause spills?

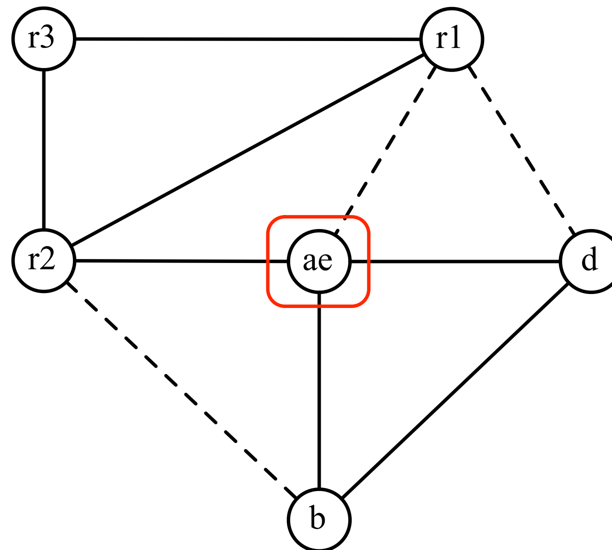
2) Can you find a rule to always perform safe coalescing?



Conservative Coalescing

- **Briggs**: Nodes *a* and *b* can be coalesced if the resulting node *ab* will have fewer than K neighbors of high degree (i.e., degree ≥ K edges)
- **George**: Nodes *a* and *b* can be coalesced if, for every neighbor *t* of *a*, either *t* already interferes with *b*, or *t* is of low degree.



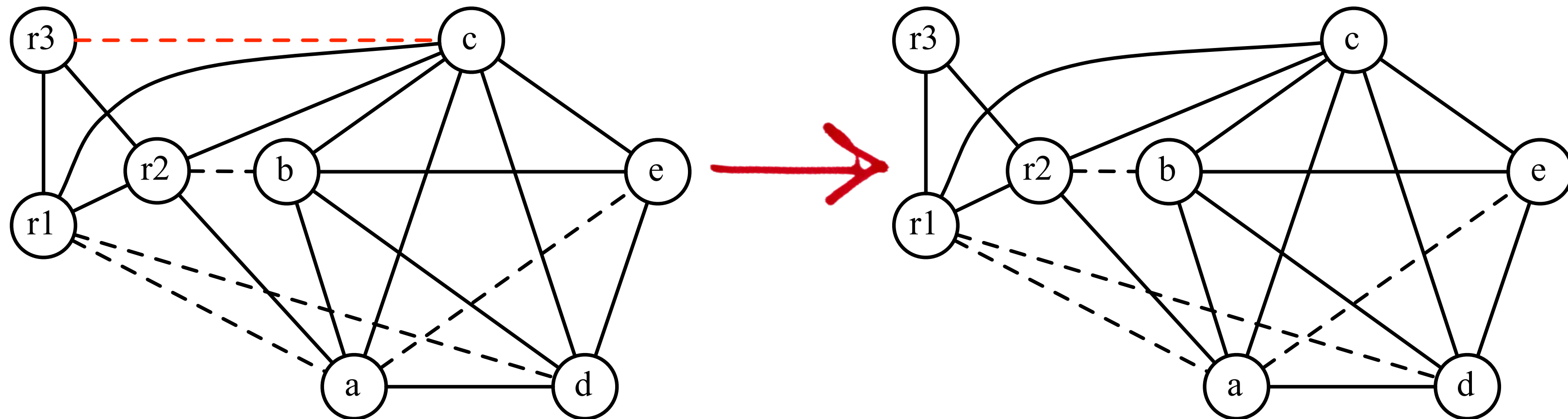


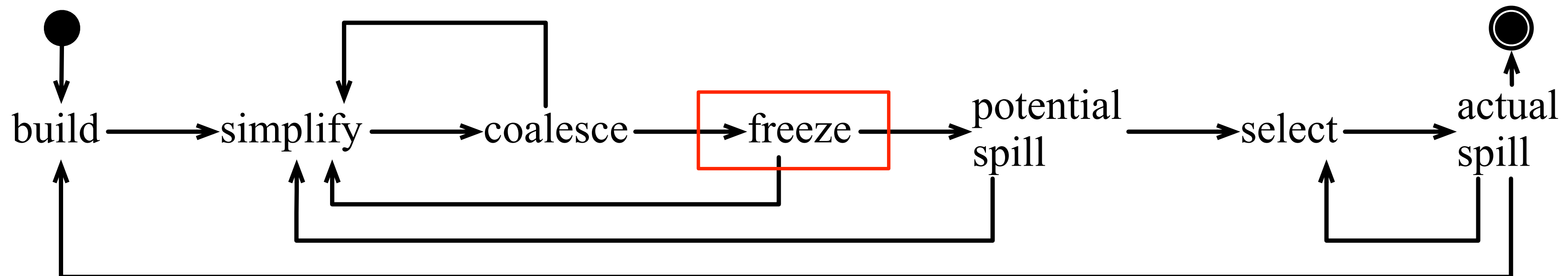
Can you explain why each of these heuristics will never cause further spilling?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC - UFMG

Freeze

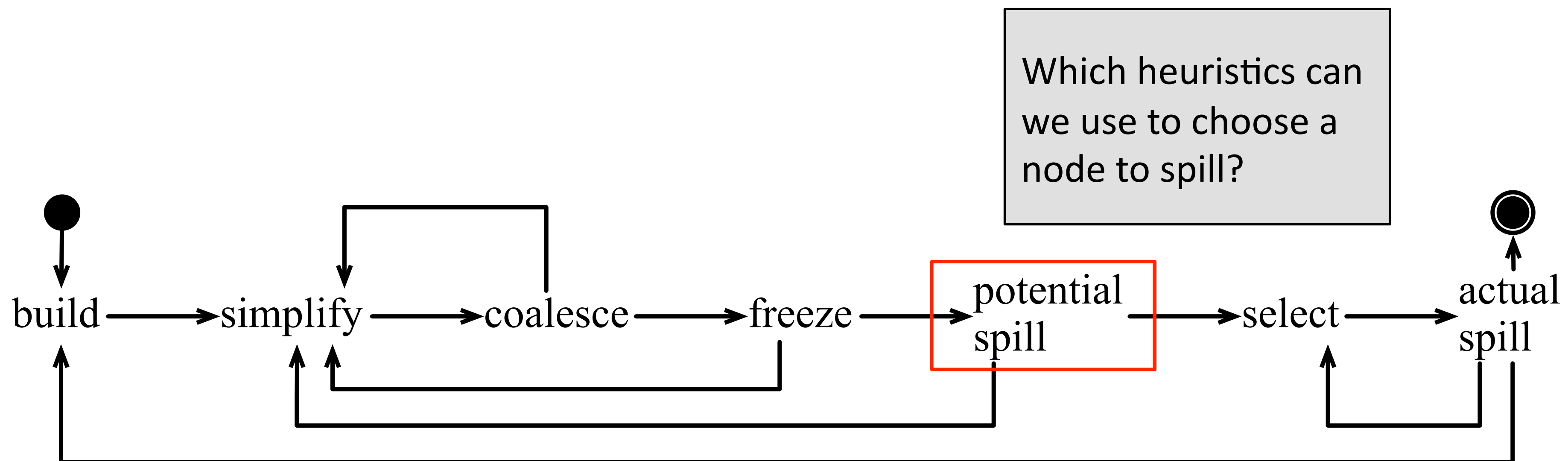
- If neither simplify nor coalesce applies, we look for a move related edge, and remove it from the graph.
- We are giving up the opportunity of coalescing these nodes.





Potential Spill

- If we cannot find any low-degree node, then we select a high-degree node for potential spill, and push it onto the stack of simplified nodes.
- **Important**: we are not spilling yet. It may be the case that once we start coloring the nodes on the stack, we end up finding a color to this node marked for spilling.



PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
DCC - UFMG

Spilling Heuristics

- That is one of the hardest parts of register allocation.
- We want to map onto memory the nodes that are less likely to be used in a dynamic run of the program.
- But we are compiling it statically. Unless we use profiling (and even profiling is not super reliable), we will have to guess which variables are likely to be more used or less used.
- We can try to remove the variable that has the lowest spilling cost, given by the formula:

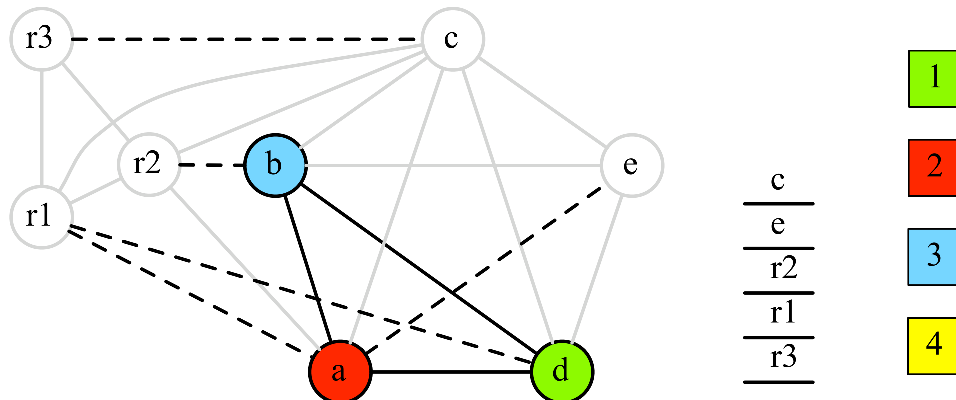
```
Spill_Cost(v)
  cost = 0
  foreach definition at block B, or use at block B
    cost += 10^N/D, where
      N is B's loop nesting factor
      D is v's degree in the interference graph
```

1) Why does it make sense to multiply the spilling cost by 10^N?

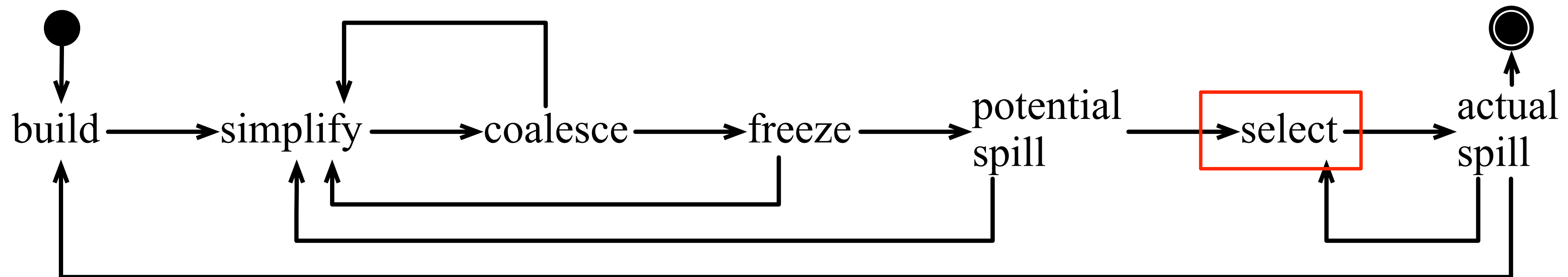
2) Why does it make sense to divide this cost by D?

Select

- Now we pop the entire stack, assigning colors to the nodes in a greedy fashion.



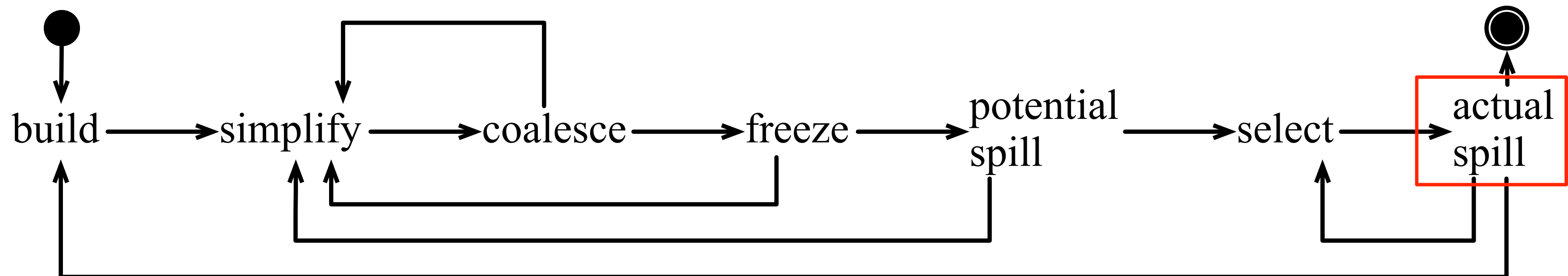
We can still find a color for a node marked for spilling. How is that possible?



Spilling

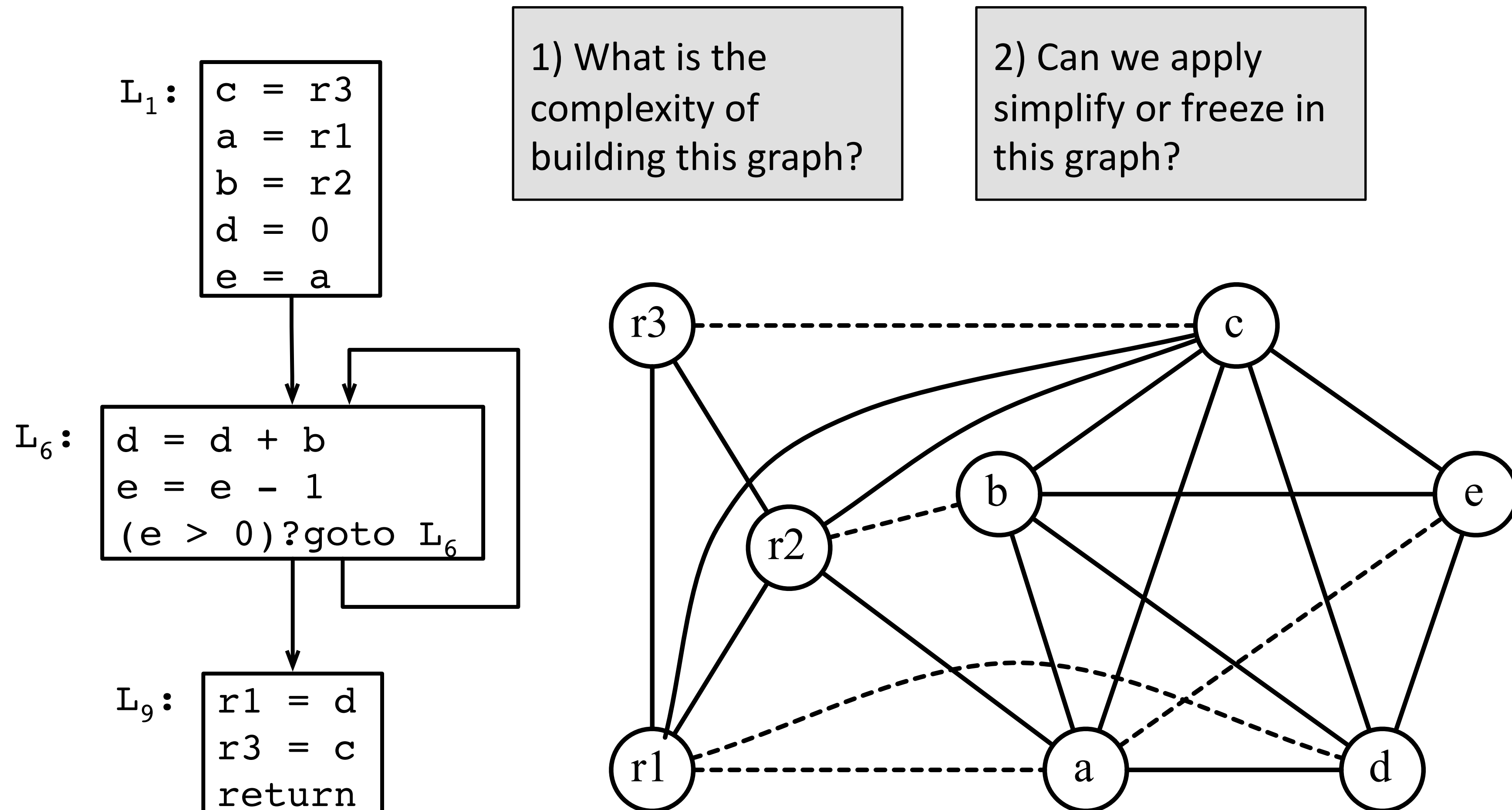
- If spilling is necessary, we may replace a variable by a sequence of loads and stores.
- Build and simplify must be repeated on the entire function.

What should we do about coalescing? Should we undo every coalescing?



Example

- We will show how to perform iterated register coalescing on our running example, assuming three physical registers.



Choosing a Node to Spill

Node	Formula	Spilling Cost
a	(2 + 10 * 0) / 4	0.50
b	(1 + 10 * 1) / 4	2.75
c	(2 + 10 * 0) / 6	0.33
d	(2 + 10 * 2) / 4	5.50
e	(1 + 10 * 3) / 3	10.33

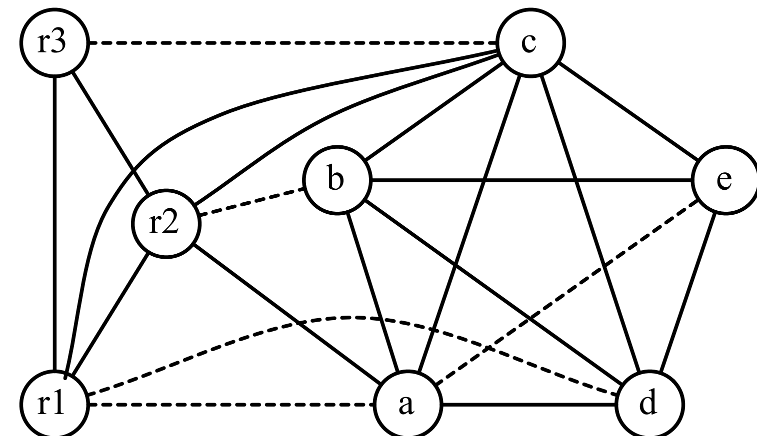
Should we spill the variable with the high spilling cost, or the low spilling cost?

$Spill_Cost(v) = (\Sigma (S_B \times 10^N))/D$, where

S_B is the number of uses and defs at B

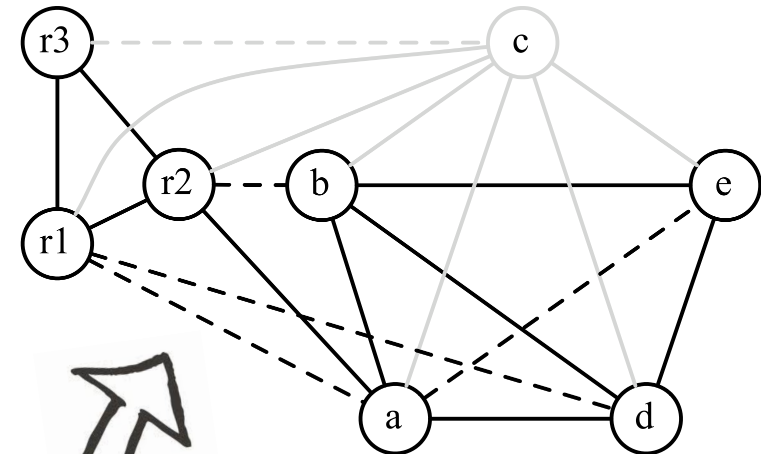
N is B's loop nesting factor

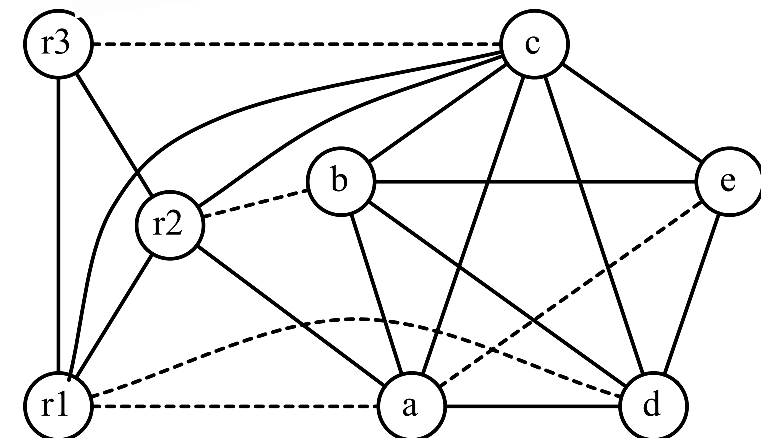
D is v's degree in the interference graph



Choosing a Node to Spill

Node	Formula	Spilling Cost
a	(2 + 10 * 0) / 4	0.50
b	(1 + 10 * 1) / 4	2.75
c	(2 + 10 * 0) / 6	0.33
d	(2 + 10 * 2)/4	5.50
e	(1 + 10 * 3) / 3	10.33





Spill_Cost(v) = $(\Sigma (S_B \times 10^N))/D$, where

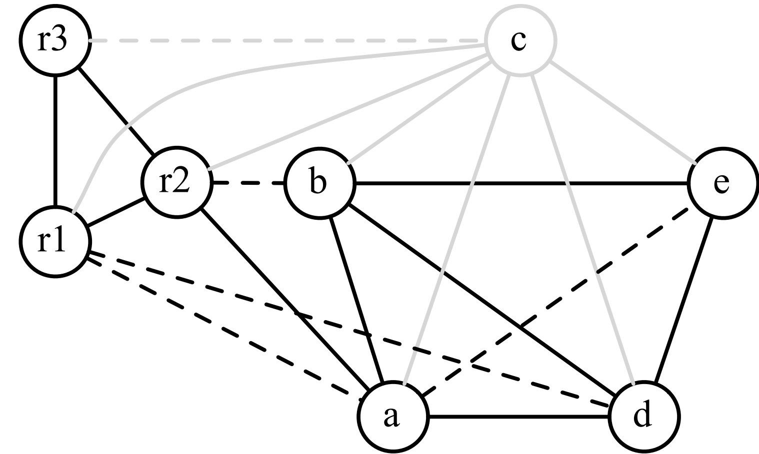
S_B is the number of uses and defs at B

N is B's loop nesting factor

D is v's degree in the interference graph

First Round of Coalescing

- **Briggs**: Nodes *a* and *b* can be coalesced if the resulting node *ab* will have fewer than K neighbors of high degree (i.e., degree ≥ K edges)
- **George**: Nodes *a* and *b* can be coalesced if, for every neighbor *t* of *a*, either *t* already interferes with *b*, or *t* is of low degree.

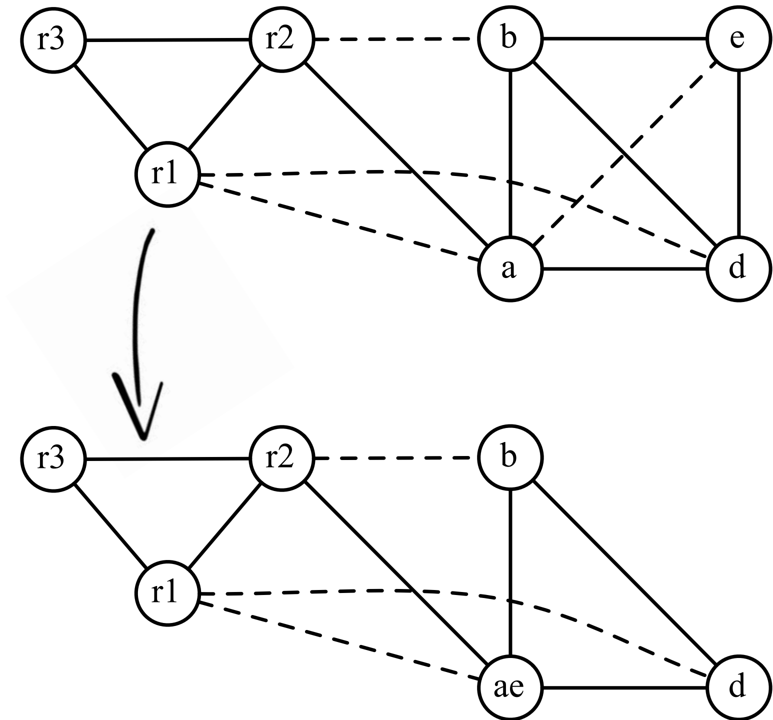


Which node can we coalesce, using each heuristics?

First Round of Coalescing

- **Briggs**: Nodes *a* and *b* can be coalesced if the resulting node *ab* will have fewer than K neighbors of high degree (i.e., degree ≥ K edges)

- **George**: Nodes *a* and *b* can be coalesced if, for every neighbor *t* of *a*, either *t* already interferes with *b*, or *t* is of low degree.



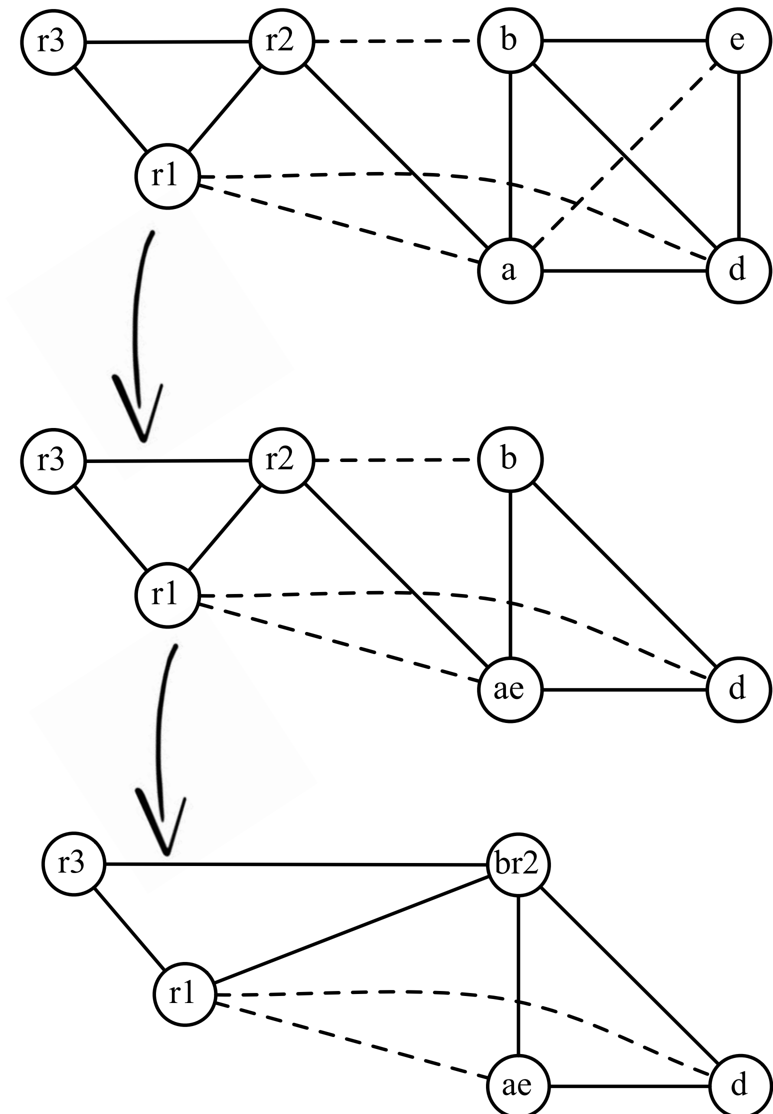
Can we coalesce more nodes?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO
EM CIÊNCIA DA COMPUTAÇÃO
DCC-UFMG

Second Round of Coalescing

- **Briggs**: Nodes *a* and *b* can be coalesced if the resulting node *ab* will have fewer than K neighbors of high degree (i.e., degree ≥ K edges)

- **George**: Nodes *a* and *b* can be coalesced if, for every neighbor *t* of *a*, either *t* already interferes with *b*, or *t* is of low degree.



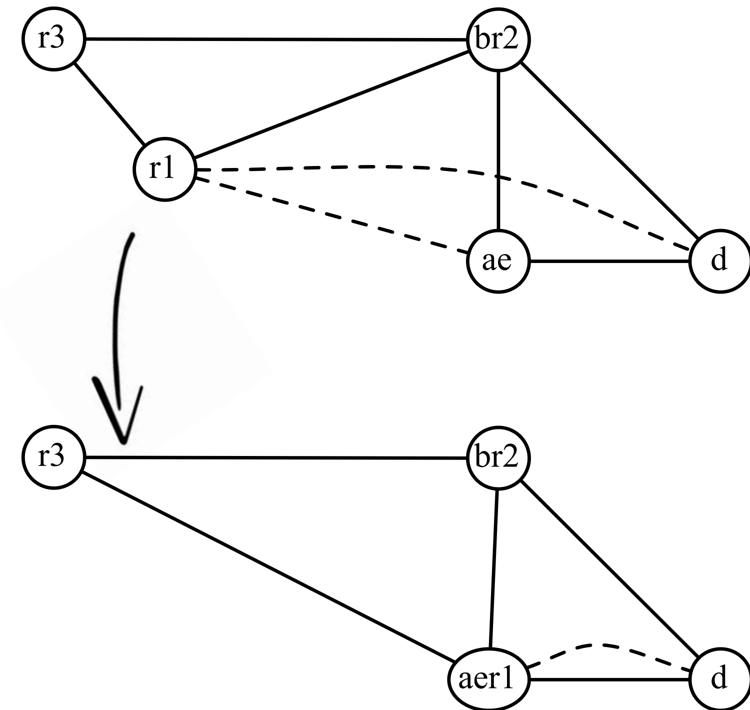
Are there still nodes to coalesce?

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

And Some More Coalescing

- **Briggs**: Nodes *a* and *b* can be coalesced if the resulting node *ab* will have fewer than K neighbors of high degree (i.e., degree ≥ K edges)

- **George**: Nodes *a* and *b* can be coalesced if, for every neighbor *t* of *a*, either *t* already interferes with *b*, or *t* is of low degree.

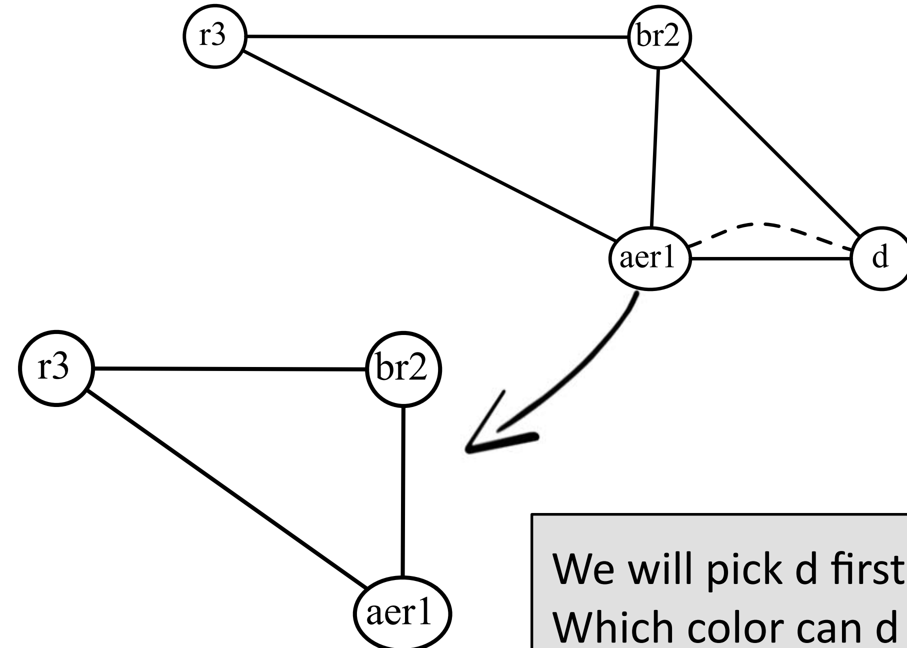


1) We cannot coalesce the nodes {a, e, r1} and d. Why is that so?

2) Given that we cannot coalesce them, what can we do?

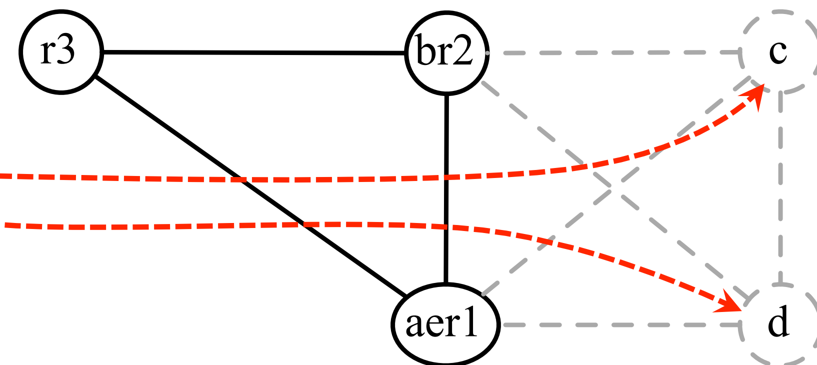
Simplification

We cannot coalesce {r1, a, e} and d because these two nodes interfere. We can only coalesce nodes that do not interfere. The only thing that we can do is to freeze and simplify d.



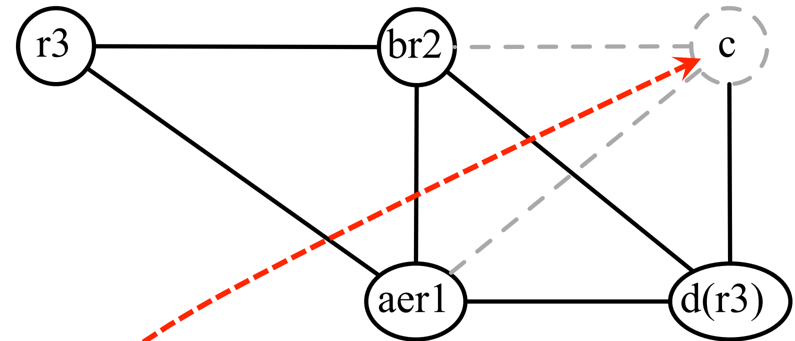
We will pick d first. Which color can d be assigned?

We have reached a configuration that only has pre-colored nodes. We do not need to do any further simplification. Instead, we can start popping **nodes** from the stack, assigning them colors in a greedy fashion.



Greedy Color assignment

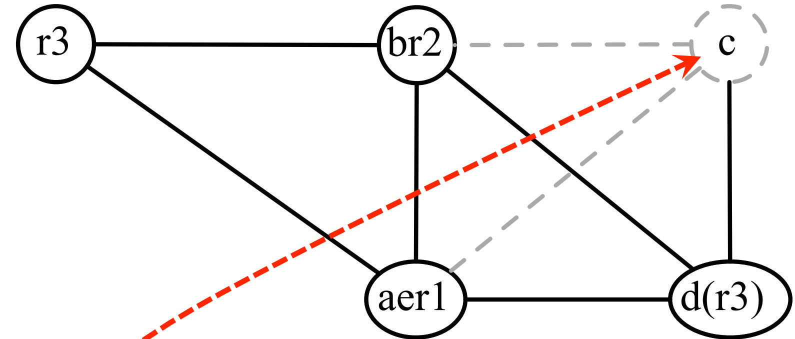
We can assign d color r3, which is the only color available for it.



1) And what will happen once we try to assign a color to c?

Potential Spill → Actual Spill

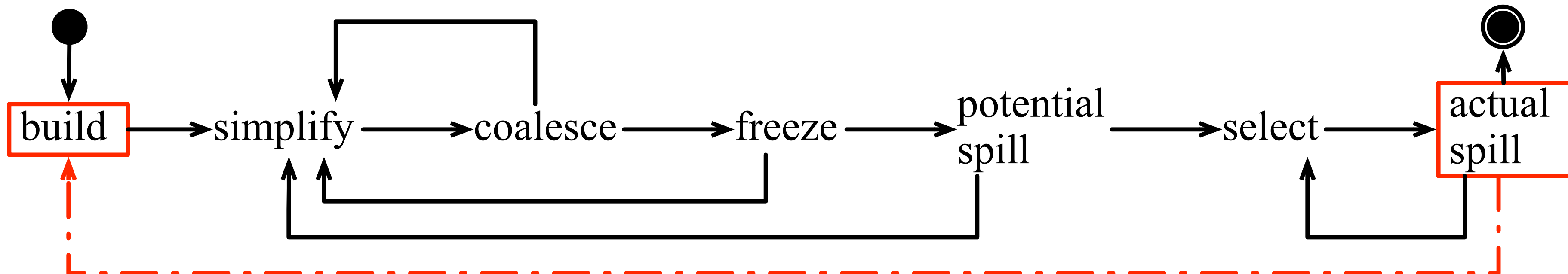
We can assign d color r3, which is the only color available for it.



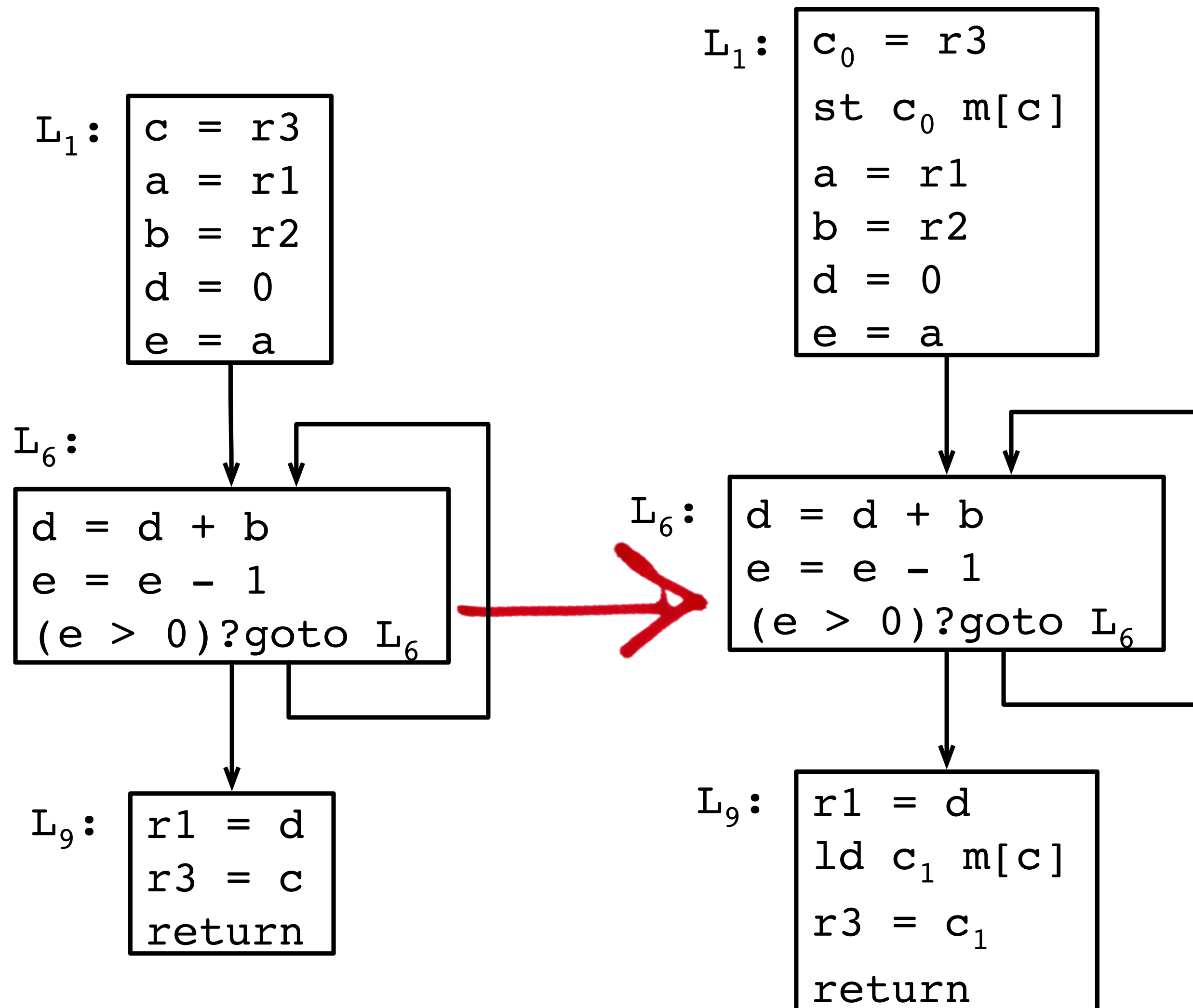
1) And what will happen once we try to assign a color to c?

2) What do we do once we find a potential spill?

We have no color left for c. If you remember, c was removed as a *potential spill*. But now we have found out that c is an *actual spill*.



A new round of "Build"



The potential spill causes us to insert loads and one store in target programs. We are splitting the variable in several new definitions.

1) How many stores do we have to insert in the target program?
2) How many loads do we have to insert?
3) How many new variable names do we have for each spilled variable?

The New Interference Graph

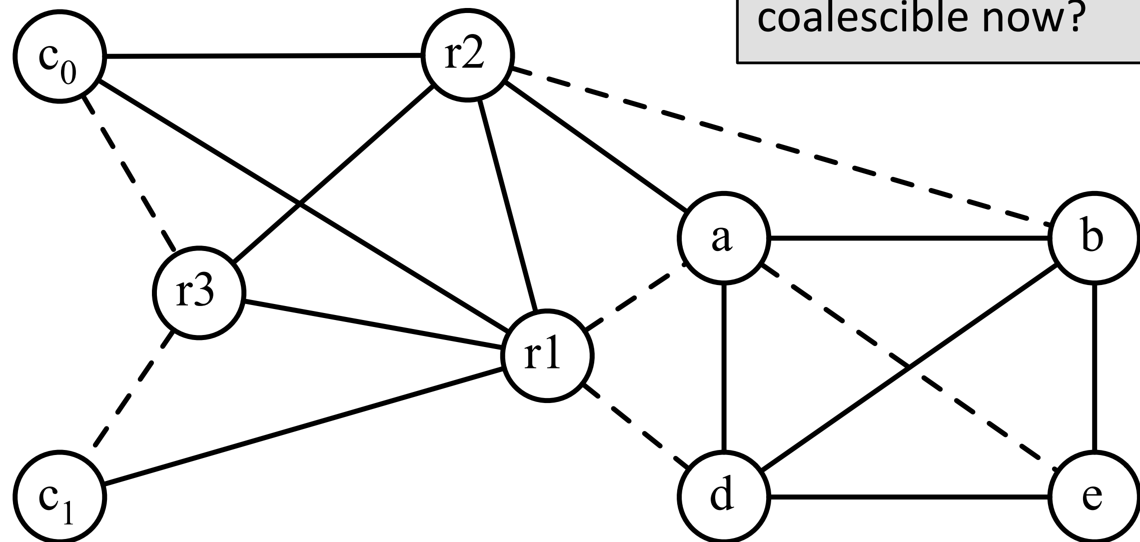
```
L1: c0 = r3
    st c0 m[c]
    a = r1
    b = r2
    d = 0
    e = a

L6: d = d + b
    e = e - 1
    (e > 0)?goto L6

L9: r1 = d
    ld c1 m[c]
    r3 = c1
    return
```

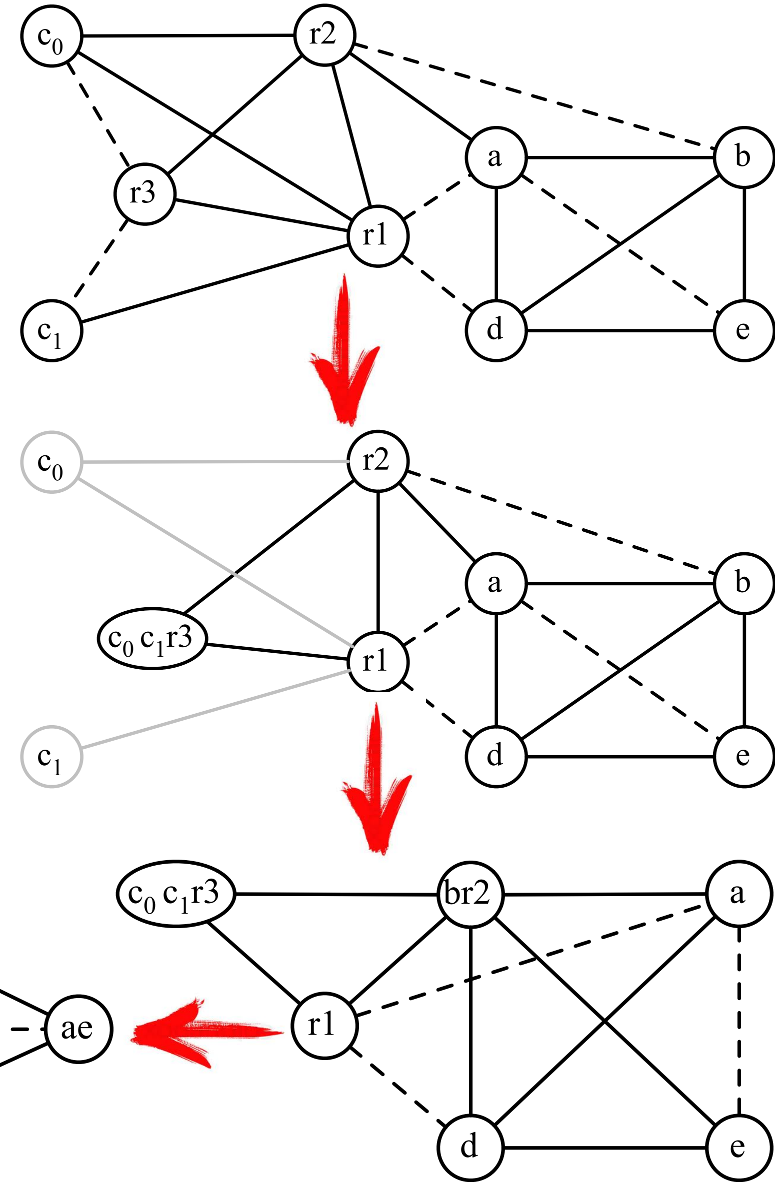
This new graph has more variables, i.e., two new names for variable c. Nevertheless, it is easy to color, because node c was interfering with two many nodes before.

Can you guess which nodes are coalescible now?

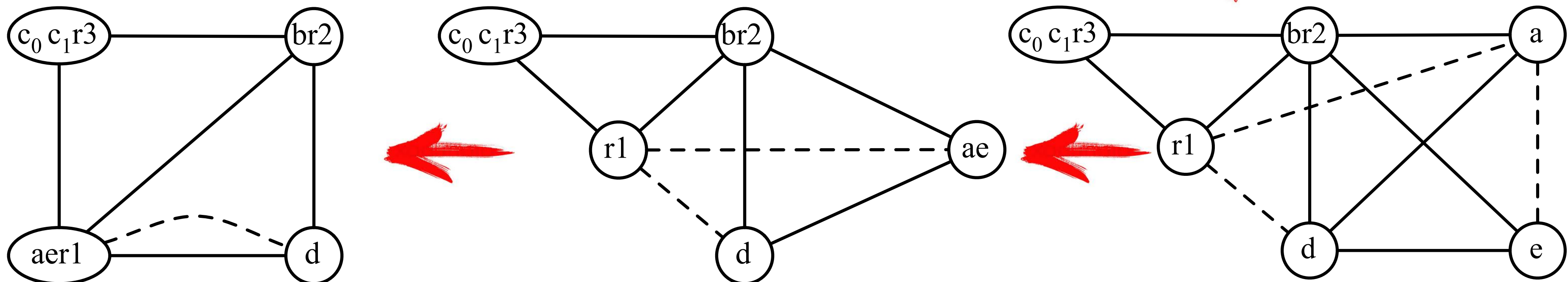


Fresh New Rounds of Coalescing

- **Briggs**: Nodes *a* and *b* can be coalesced if the resulting node *ab* will have fewer than K neighbors of high degree (i.e., degree ≥ K edges)
- **George**: Nodes *a* and *b* can be coalesced if, for every neighbor *t* of *a*, either *t* already interferes with *b*, or *t* is of low degree.

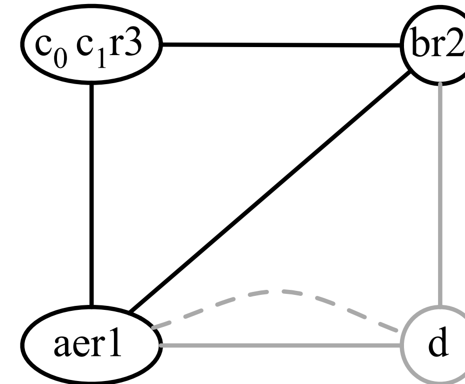


What is the next action to be taken?



Simplification

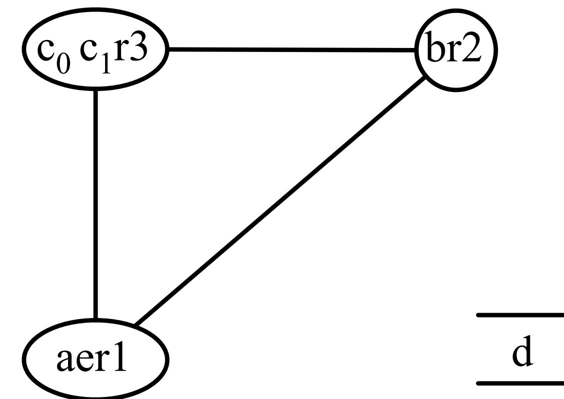
There is no more coalescing that could be performed; hence, we simplify d.



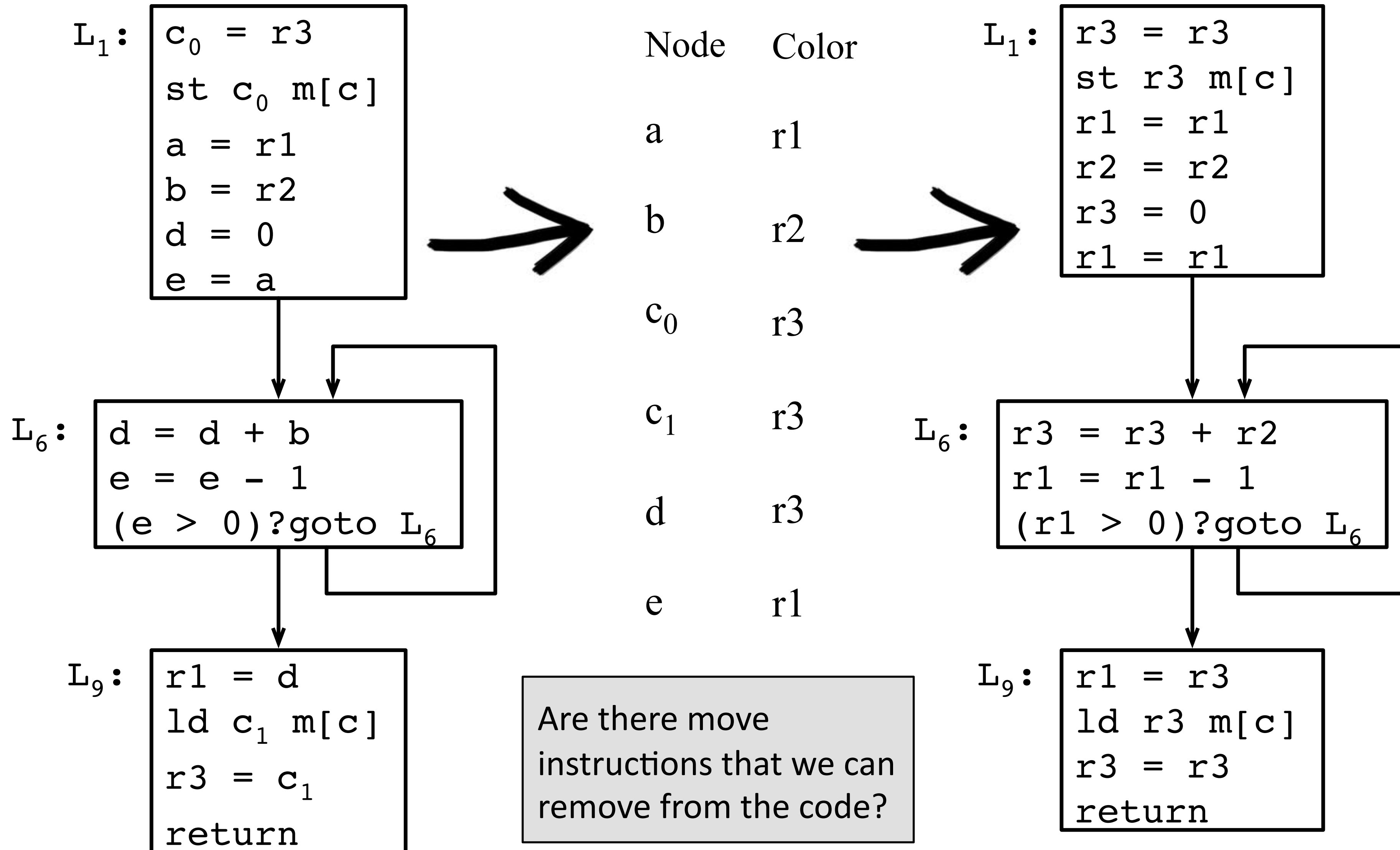
Once we simplify d, we only have pre-colored nodes in the graph. We must find a color for d.

1) What is the only color available for d?

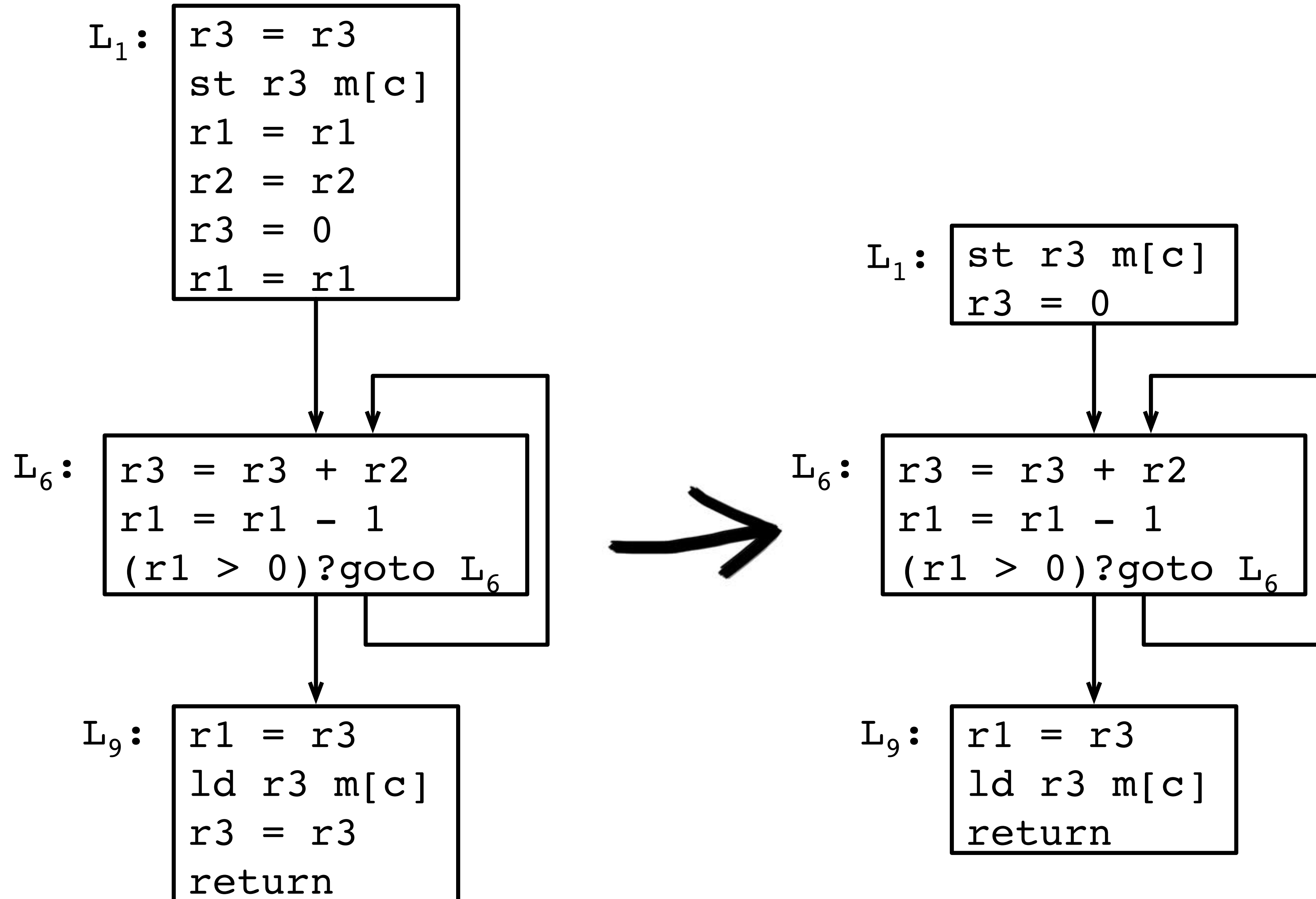
2) Why is it not necessary to find colors for the other nodes?



Program Re-Writing



Copy Elimination



PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

A Bit of History

- Register allocation by graph coloring was introduced by Chaitin et al. in 1981.
- We have used the iterated register coalescing as an example of graph coloring allocator. This was a creation of George and Appel
- Linear Scan was an invention of Poletto and Sarkar.

- Chaitin, G., Auslander, M., Chandra, A., Cocke, J., Hopkins, M., and Markstein, P. "Register allocation via coloring", Computer Languages, p 47-57 (1981)
- George, L., and Appel, A., "Iterated Register Coalescing", North Holland, TOPLAS, p 300-324 (1996)
- Poletto, M., and Sarkar, V., "Linear Scan Register Allocation", TOPLAS, p 895-913 (1999)