CS 335: Register Allocation

Swarnendu Biswas

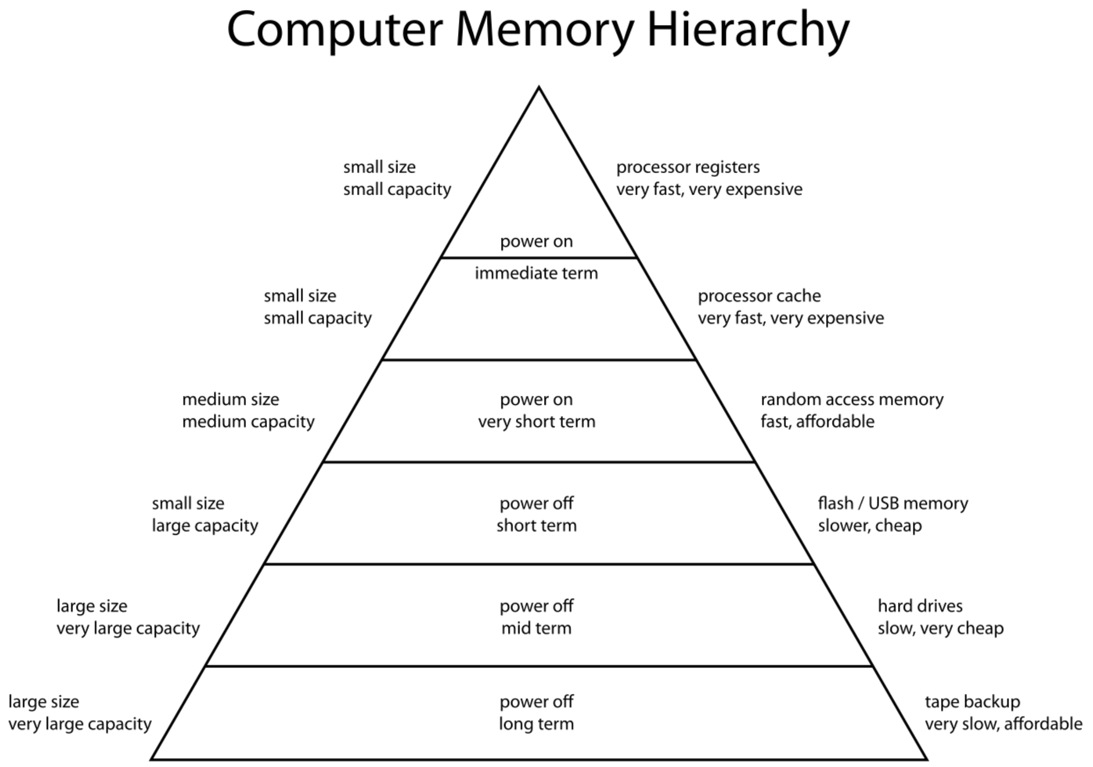
Semester 2022-2023-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

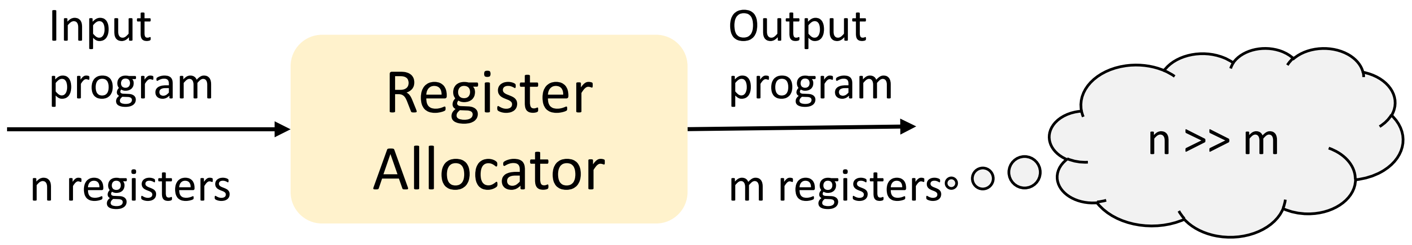
Impact of Register Operands

- Instructions involving register operands are faster than those involving memory operands
- Code is often smaller and hence is faster to fetch
- Efficient utilization of registers is important
    - Number of general-purpose registers is limited
    - E.g., ~16-32 64-bit general-purpose registers



Goals in Register Allocation

- All variables are not used (or live) at the same time
- Register allocator in a compiler helps with decision making
  - Which values will reside in registers?
  - Which register will hold each of those values?
  - At each program location, values stored in virtual registers in the IR are mapped to physical registers



Goals in Register Allocation

- Programs spend most of their time in loops
    - Natural to store values in innermost loops in registers
- **Register pressure** measures the availability of free registers
    - High register pressure implies that a large fraction of registers are in use
- When no registers are available to store a computation, the contents of one of the in-use registers must be stored into memory
    - This is called **register spilling,** requires generating load and store instructions
    - Spilling increases code size and execution time
- Goal in register allocation is thus to minimize the impact of spills especially for performance-critical code

Register Allocation and Assignment

- **Allocation**
  - Maps an unlimited name space onto the register set of the target machine
  - Register-to-register model: map virtual registers to the physical register set and spill values that do not fit in the physical register set
  - Memory-to-memory model: map a subset of memory locations to a set of physical registers
- **Assignment**
  - Assumes that allocation has already been performed
  - Maps an allocated name set to the physical registers of the target machine

Challenges in Register Allocation

- Architectures provide different register classes
  - General purpose registers, floating-point registers, predicate and branch target registers
  - General-purpose registers may be used for floating-point register spills, which implies an order for allocation
  - Registers may be aliased, cannot use a register and its constituent registers at the same time
    - x86 has 32-bit registers whose lower halves are used as 16-bit or 8-bits registers
    - Similar for vector registers like `zmm`, `ymm`, and `xmm`
  - If different register classes overlap, the compiler must allocate them together
- Architecture calling conventions place more constraints on register usage
  - E.g., PowerPC requires parameters to be passed in R3-R10 and the return is in R3

Register Allocation Problem

- General formulation of the problem is NP-Complete
  - E.g., register allocation for a set of BBs, multiple control flow paths, multiple data types, and non-uniform cost of memory access complicate the analysis
  - Optimal allocation can be done in polynomial time for very restricted versions with a single BB and with one data type

More about Register Allocation

- Types of allocation: local (over a BB), global (over a whole function), or Interprocedural (across function boundaries traversed via call-graph)
- Backend includes instruction selection, register allocation, and instruction scheduling
  - Performing allocation first restricts movement of code during scheduling, not a good idea
  - Scheduling instructions first cannot handle spill code introduced during allocation
  - Possible order: selection -> scheduling -> allocation -> scheduling

Local Register Allocation

Frequency count and Variable usage

Local Register Allocation

- Assumptions
  - Considers only a single basic block
  - Loads values from memory at the start of a BB and stores to memory at the end of the BB
  - IR makes use of many virtual registers
  - Target machine has a uniform class of k general-purpose registers

$$\begin{array}{c} \mathrm{op}_1\ vr_{1_1}, vr_{1_2} \Rightarrow vr_{1_3} \\ \mathrm{op}_2\ vr_{2_1}, vr_{2_2} \Rightarrow vr_{2_3} \\ \ldots \\ \mathrm{op}_n\ vr_{n_1}, vr_{n_2} \Rightarrow vr_{n_3} \end{array} \quad \Rightarrow \quad \begin{array}{c} \mathrm{op}_1\ ?,? \Rightarrow ? \\ \mathrm{op}_2\ ?,? \Rightarrow ? \\ \ldots \\ \mathrm{op}_n\ ?,? \Rightarrow ? \end{array}$$

Top-Down Allocation with Frequency Counts

- Idea
    - Count the **frequency of occurrence** of virtual registers
    - Map virtual registers to physical registers in descending order of frequency
    - If the BB uses fewer than k virtual registers, then mapping is trivial
    - $F \cong 2{-}4$ registers are required to execute spill code
    - Assign the top $k - F$ virtual registers to physical registers
    - Rewrite the code and replace virtual registers with physical registers
    - For unassigned virtual registers, generate code sequence to spill code using the F reserved registers

Example of Top-Down Allocation

Assume there are three physical registers.

Usage Count
$R_a = 4$
$R_b = 3$
$R_c = 2$
$R_d = 2$
$R_e = 2$
$R_f = 2$
$R_g =2$
$R_h = 2$

	Liveness Information							
LD R_a, 1028								
	R_a							
MOV R_b, R_a								
	R_a	R_b						
MUL R_c, R_a, R_b								
	R_a	R_b	R_c					
LD R_d, x	// Spill R_c based on **usage count**							
	R_a	R_b	R_c	R_d				
SUB R_e, R_d, R_b	// Reuse R_b							
	R_a		R_c		R_e			
LD R_f, z	// Reuse R_d							
	R_a		R_c		R_e	R_f		
MUL R_g, R_e, R_f	// Reuse R_e							
	R_a		R_c				R_g	
SUB R_h, R_g, R_c	// Restore R_c, reuse R_g							
	R_a							
MOV R_h, R_a								

Example of Top-Down Allocation

LD R_a, 1028
MOV R_b, R_a
MUL R_c, R_a, R_b
LD R_d, x
SUB R_e, R_d, R_b
LD R_f, z
MUL R_g, R_c, R_f
SUB R_h, R_g, R_c
MOV R_a, R_h

$\Rightarrow$

LD R_a, 1028
MOV R_b, R_a
MUL R_c, R_a, R_b
ST $off_c(R_{SP})$, R_c
LD R_d, x
SUB R_e, R_d, R_b
LD R_f, z
MUL R_g, R_e, R_f
LD R_c, $off_c(R_{SP})$
SUB R_h, R_g, R_c
MOV R_a, R_h

spill code

register assignment $\Rightarrow$

LD R_1, 1028
MOV R_2, R_1
MUL R_3, R_1, R_2
ST $off_c(R_{SP})$, R_3
LD R_3, x
SUB R_2, R_3, R_2
LD R_3, z
MUL R_2, R_2, R_3
LD R_3, $off_c(R_{SP})$
SUB R_2, R_2, R_3
MOV R_1, R_2

- Top-down local allocation allocates a physical register to one virtual register for the **entire** BB
- Allocation can be suboptimal if variables show phased behavior (e.g., R_a)
    - A variable that is heavily-used in the first half of the BB and not used in the second half still stays in the physical register

Bottom-Up Allocation Based on Variable Usage

- Iterates over the instructions in the BB and makes decisions based on variable usage (not count)
- Assumes that the physical registers are initially empty and places them on a free list
- Satisfies demand for registers from the free list until that list is exhausted
- If the free list is empty, spill a variable to memory and reuse the register
    - Spill the variable whose next use is farthest in the future

Bottom-Up Allocation

- Assume that registers are grouped in classes (e.g., general-purpose, floating-point)
  - `Size`: # of physical registers
  - `Name`: virtual register name
  - `Next`: distance to next reuse
  - `Free`: flag to indicate whether currently in use
  - `Stack`: free physical registers

```
struct RegClass {
  int Size;
  int Name[Size];
  int Next[Size];
  int Free[Size];
  int Stack[Size];
  int StackTop;
}
```

Helper Functions in the Bottom-Up Algorithm

```
Ensure($vr$, regClass)
  if $vr$ is already in regClass
    reg = $vr$'s physical register
  else
    reg = Allocate($vr$, regClass)
    emit code to move $vr$ into reg
  return reg

Free(pr, regClass)
  // 0≤i<Size
  regClass.Name[i] = null
  regClass.Next[i] = -1
  regClass.Free[i] = true
  push(regClass, pr)
```

```
Allocate($vr$, regClass)
  if regClass.StackTop >= 0
    // free register available
    i = pop(regClass)
  else
    // Check for farthest use
    i = j s.t. $\forall_{j \in Size} \max(\text{regClass.Next}[j])$
    // Emit spill code
    store contents of j
  regClass.Name[i] = $vr$
  regClass.Next[i] = -1
  regClass.Free[i] = false
  return i
```

Bottom-Up Algorithm

```
for each instruction $i = \{1 \dots n\}$ in BB
  // Instruction $i$: $\text{op}_i\ vr_{i_1}, vr_{i_2} \Rightarrow vr_{i_3}$
  $r_x$ = Ensure($vr_{i_1}$, regClass($vr_{i_1}$))
  $r_y$ = Ensure($vr_{i_2}$, regClass($vr_{i_2}$))
  if $vr_{i_1}$ is not needed after $i$
    Free($r_x$, regClass($r_x$))
  if $vr_{i_2}$ is not needed after $i$
    Free($r_y$, regClass($r_y$))
  $r_z$ = Allocate($vr_{i_3}$, regClass($vr_{i_3}$))
  rewrite $i$ as $\text{op}_i\ r_x, r_y \Rightarrow r_z$
  if $vr_{i_1}$ is needed after $i$
    regClass.next[$r_x$]= Dist($vr_{i_1}$)
  if $vr_{i_2}$ is needed after $i$
    regClass.next[$r_y$]= Dist($vr_{i_2}$)
  regClass.next[$r_z$]= Dist($vr_{i_3}$)
```

Behaviour with Bottom-Up Allocation

Assume there are three physical registers.

	Liveness Information							
`LD` R_a`, 1028`								
	R_a							
`MOV` R_b`,` R_a								
	R_a	R_b						
`MUL` R_c`,` R_a`,` R_b								
	R_a	R_b	R_c					
`LD` R_d`, x`	// Spill R_a based on **farthest use**							
	R_a	R_b	R_c	R_d				
`SUB` R_e`,` R_d`,` R_b	// Reuse R_b							
	R_a		R_c		R_e			
`LD` R_f`, z`	// Reuse R_d							
	R_a		R_c		R_e	R_f		
`MUL` R_g`,` R_e`,` R_f	// Reuse R_e							
	R_a		R_c				R_g	
`SUB` R_h`,` R_g`,` R_c	// Restore R_a, reuse R_g							
	R_a							
`MOV` R_h`,` R_a								

Challenges in Bottom-Up Allocation

- A store on a spill is unnecessary if the **data is clean**
  - Register contains a constant value or a return from a load
- It seems a spill should be stored only if the data is dirty
- Assume a two-register machine: values x_1 is clean and x_2 is dirty
- Assume reference stream for the rest of the BB is $x_3 x_1 x_2$

On spilling dirty values

```
store x₂
load x₃
load x₂
```

On spilling clean values

```
load x₃
load x₁
```

overwrite x_1

Challenges in Bottom-Up Allocation

- A store on a spill is unnecessary if the **data is clean**
  - Register contains a constant value or a return from a load
- It seems a spill should be stored only if the data is dirty
- Assume a two-register machine: values x_1 is clean and x_2 is dirty
- Assume reference stream for the rest of the BB is $x_3 x_1 x_3 x_1 x_2$

On spilling clean values

```
load $x_3$
load $x_1$
load $x_3$
load $x_1$
```

On spilling dirty values

```
store $x_2$
load $x_3$
load $x_2$
```

Global Register Allocation

Usage count, live range, and graph coloring

Global Register Allocation

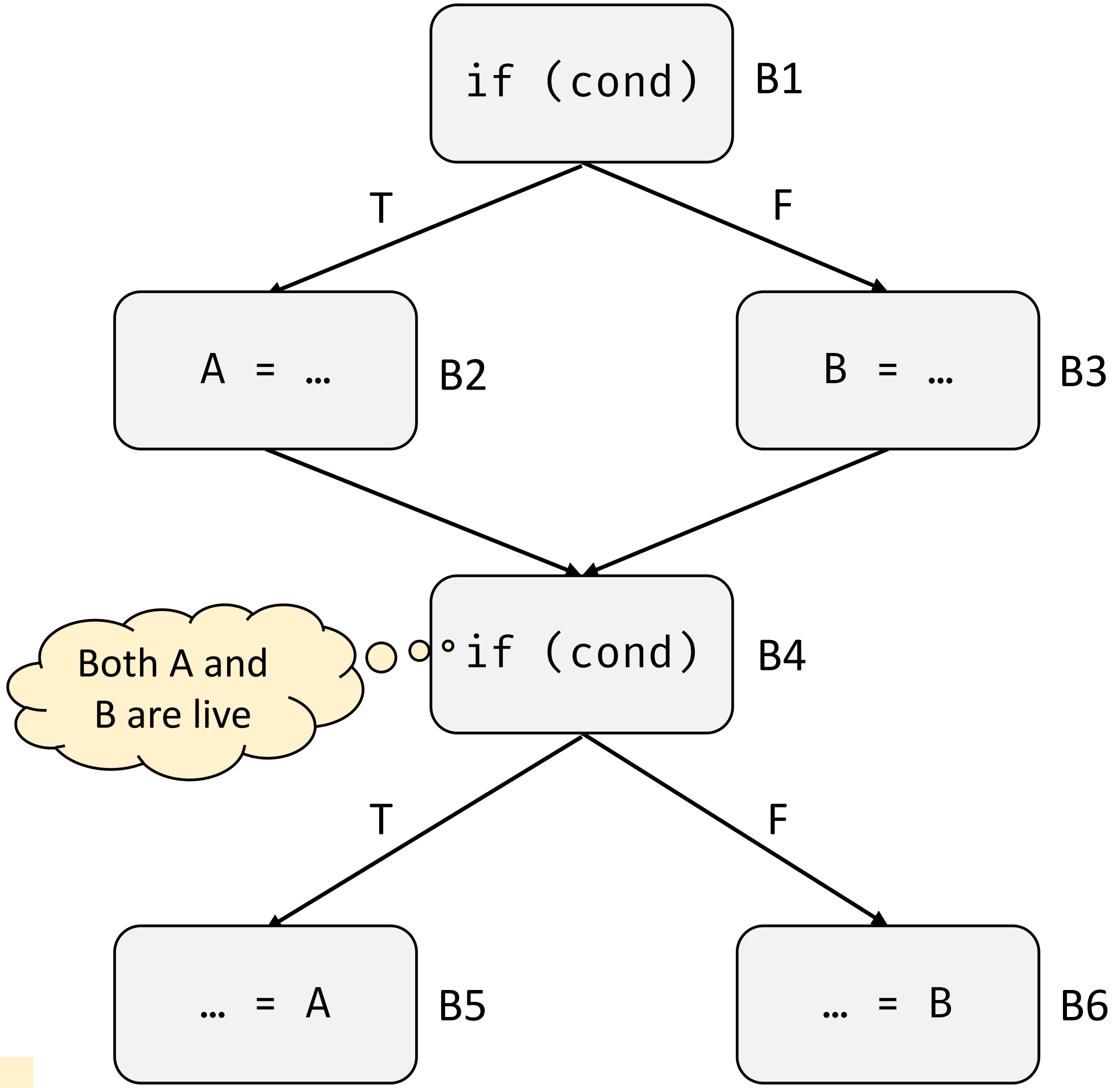
- Scope is either multiple BBs or a whole procedure
- **Decision problem**: Given an input program in IR form (e.g., CFG) and a number k, is there an assignment of registers to program variables such that no conflicting variables are assigned the same register, no extra loads and stores are introduced, and at most k registers are used?
- Fundamentally a more complex problem than local register allocation
  - Need to consider def-use across multiple blocks
  - Cost of spilling may not be uniform since it depends on the execution frequency of the block where a spill happens

Live Ranges

- A variable is **live** at a point p in the CFG if there is a use of the variable in the path from p to the end of the CFG
- A **live range** for a variable is the smallest set of program points at which it is live
  - A variable's live range starts at the point in the code where the variable receives a value and ends where that value is used for the last time
  - A single variable may be represented by many live ranges
- Two variables interfere or **conflict** if their live ranges intersect (i.e., both are live)

Example

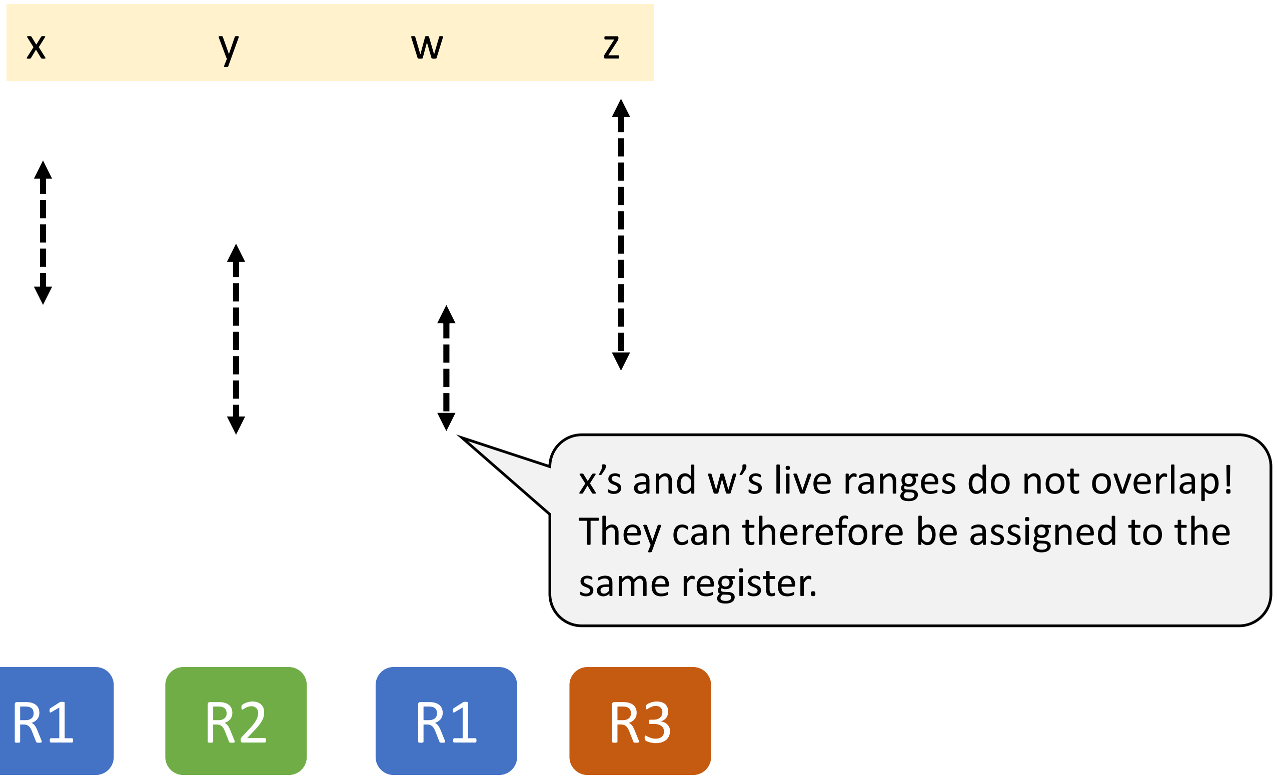
```
if (cond) {
  A = …
} else {
  B = …
}
if (cond) {
  … = A
} else {
  … = B
}
```



Live range of A: B2, B4, B5
Live range of B: B3, B4, B6

Utility of Live Ranges

```
z = 1
x = 2 * x
y = 3 * z
w = x + y
print y + z
x = y * w
```



x's and w's live ranges do not overlap! They can therefore be assigned to the same register.

Global Register Allocation Based on Usage Counts

- Heuristic method to allocate registers for most frequently-used variables
- Requires information about liveness of variables at the entry and exit of each BB in the loop body
- Once a variable is computed into a register, it stays in that register until the end of the BB

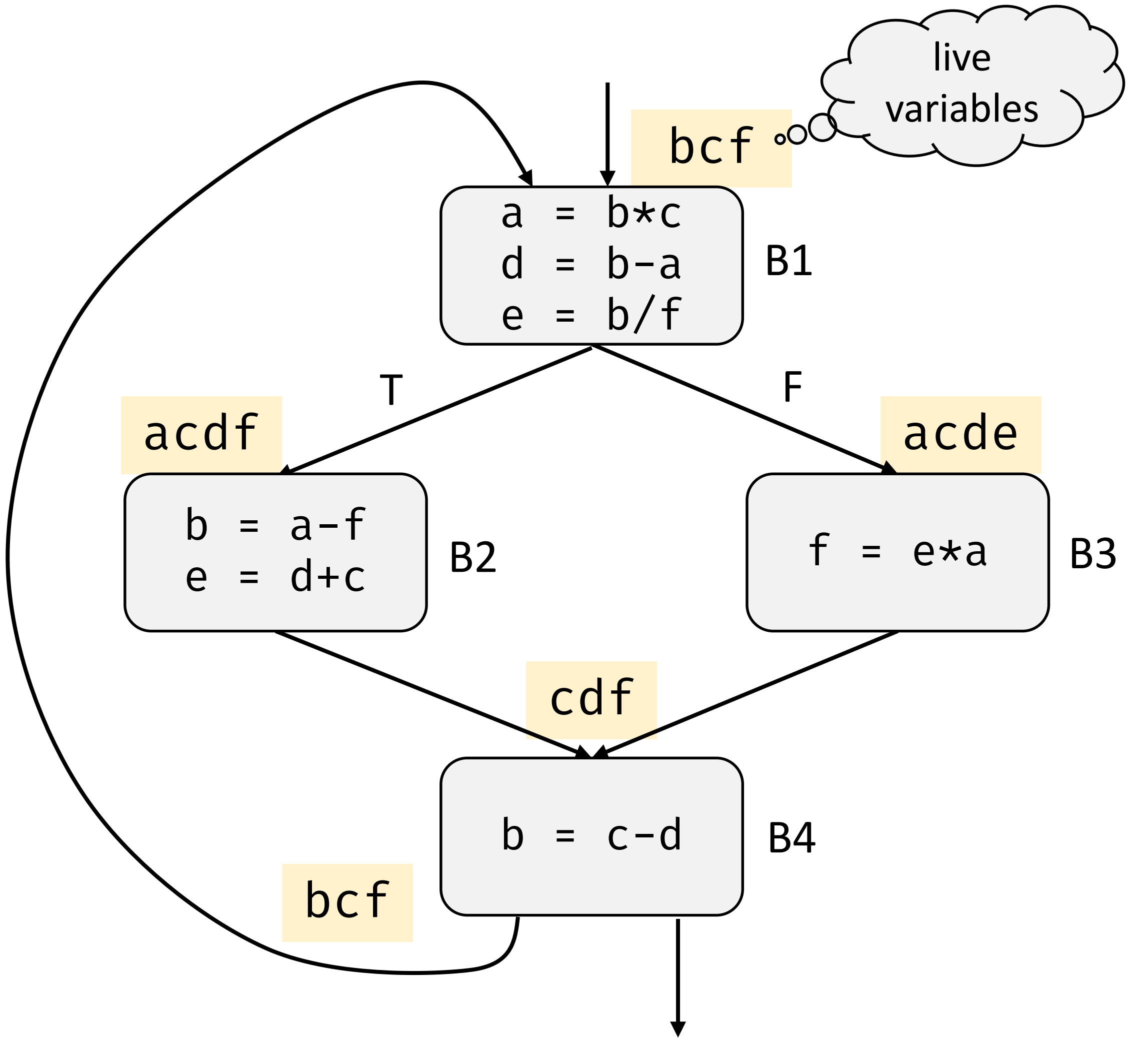
Global Register Allocation Based on Usage Counts

- For every **usage** of a variable v in a BB **until it is first defined**, do:
  - savings(v) = savings(v) + 1
- After v is defined, it stays in the register and all further references are to that register
- For every variable v **computed** in a BB, if it is live on exit from the BB,
  - Count a savings of 2 because it is not necessary to store it at the end of the BB
- Total savings per variable v is $\sum_{\text{blocks } B \text{ in } L} use(v,B) + 2 * live(v,B)$
  - $live(v,B)$ is 1 or 0 depending on whether v is live on exit from B
  - Assume load/store instructions cost two units
- Estimate of savings is approximate since all BBs are not executed with the same frequency

Example

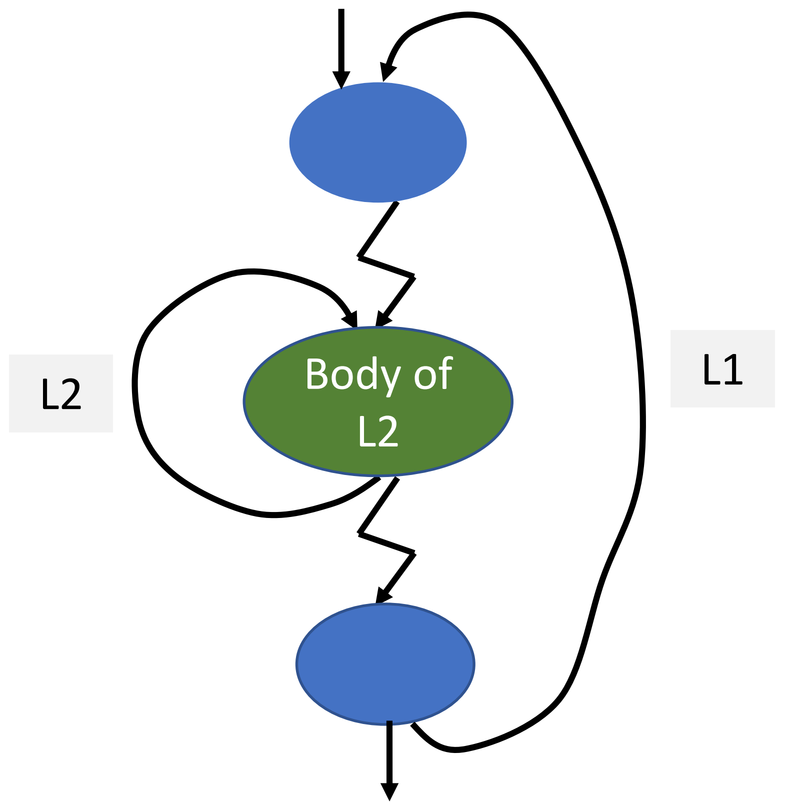
	B1	B2	B3	B4	Sum
a	(0+2)	(1+0)	(1+0)	(0+0)	4
b	(3+0)	(0+0)	(0+0)	(0+2)	5
c	(1+0)	(1+0)	(0+0)	(1+0)	3
d	(0+2)	(1+0)	(0+0)	(1+0)	4
e	(0+2)	(0+0)	(1+0)	(0+0)	3
f	(1+0)	(1+0)	(0+2)	(0+0)	4

If there are 3 registers, they will be allocated to b, and (say) a and d



Global Register Allocation Based on Usage Counts for Nested Loops

- Assign registers for inner loops first before considering outer loops
- Let loop L1 nest L2
- For variables assigned registers in L2 but not in L1,
  - Load these variables on entry to L2 and store them on exit from L2
- For variables assigned registers in L1 but not in L2,
  - Store these variables on entry to L2 and load them on exit from L2

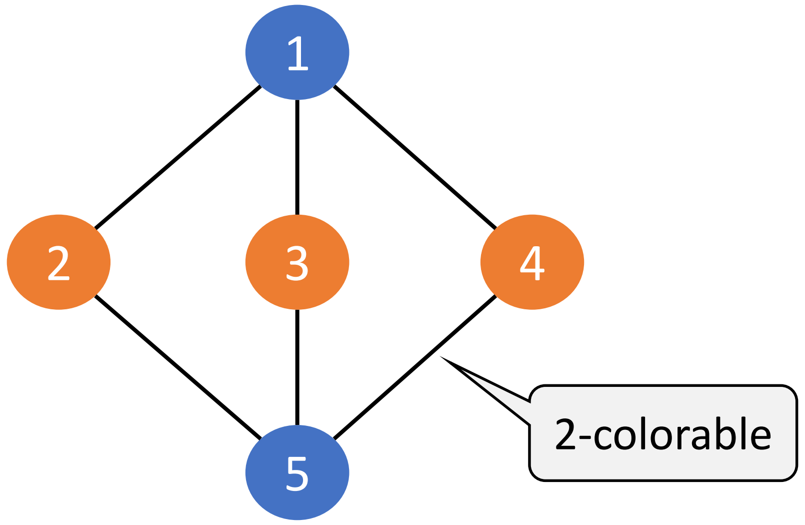


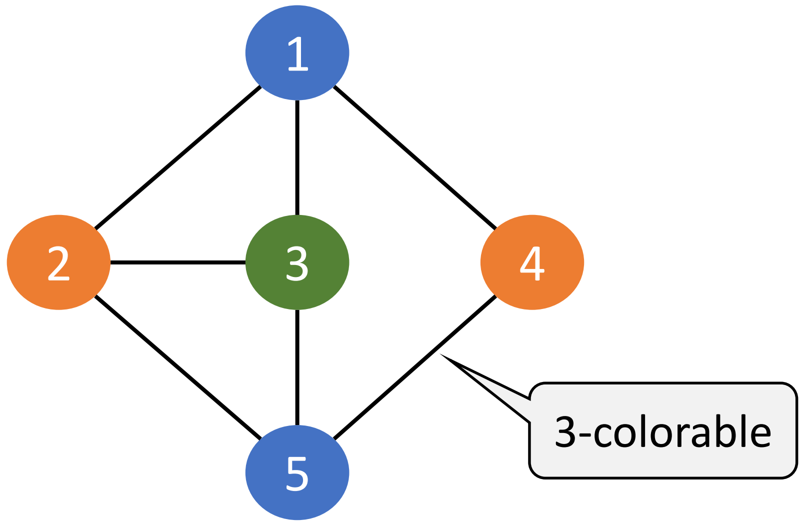
Global Register Allocation Using Graph Coloring

Chaitin's Allocator

The Graph Coloring Problem

- For an arbitrary graph G, a coloring of G assigns a color to each node in G so that no pair of adjacent nodes have the same color
- A coloring that uses k colors is termed a ***k*-coloring**
- The smallest possible k for a given graph is called the graph's **chromatic number**
- Finding the chromatic number of a graph is NP-complete
- Determining if a graph is k-colorable, for some fixed k, is NP-complete





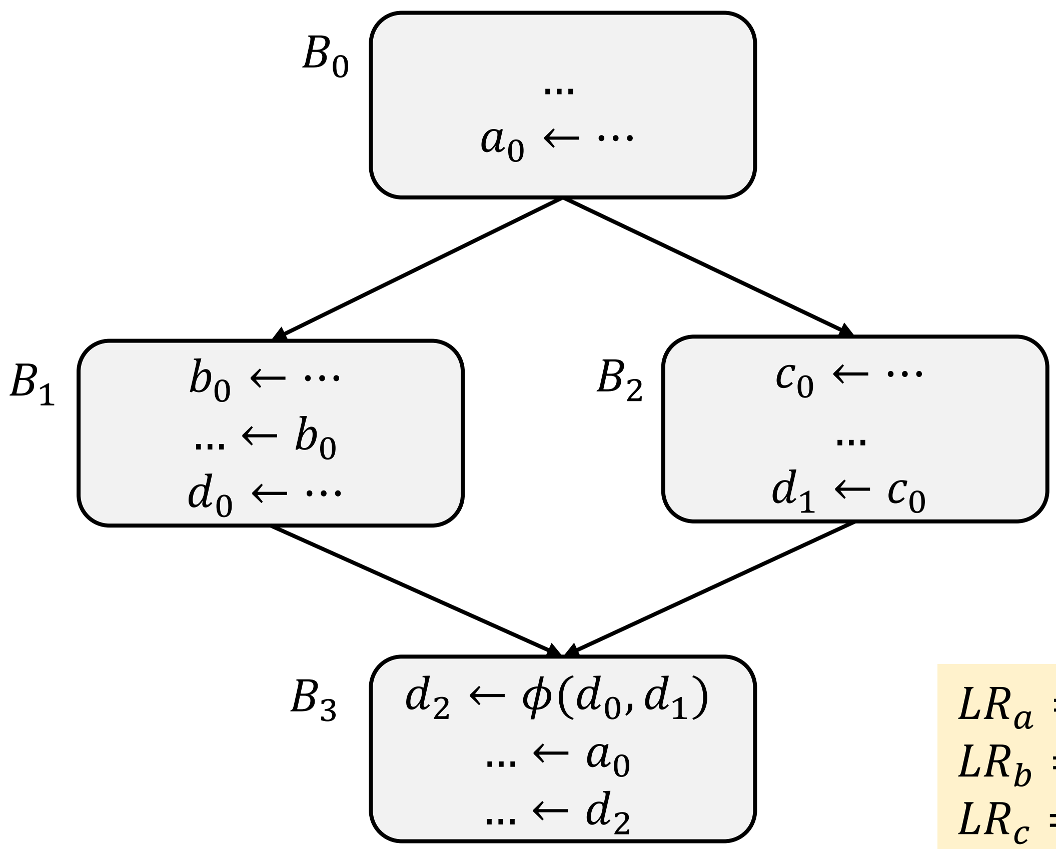
Global Register Allocation with Graph Coloring

- Compilers model register allocation through k-coloring on an interference graph where k is the number of physical registers available to the allocator
  - Each color represents an available register
  - Spill some values to memory and retry if k-coloring is not possible
- An **interference graph** models conflicts in live regions
  - If variables a and b are active (live) at the same point, they cannot be assigned to the same register
  - Nodes in an interference graph represent live ranges (LR) for a variable, and an edge (i, j) indicates LR_i and LR_j cannot share a register

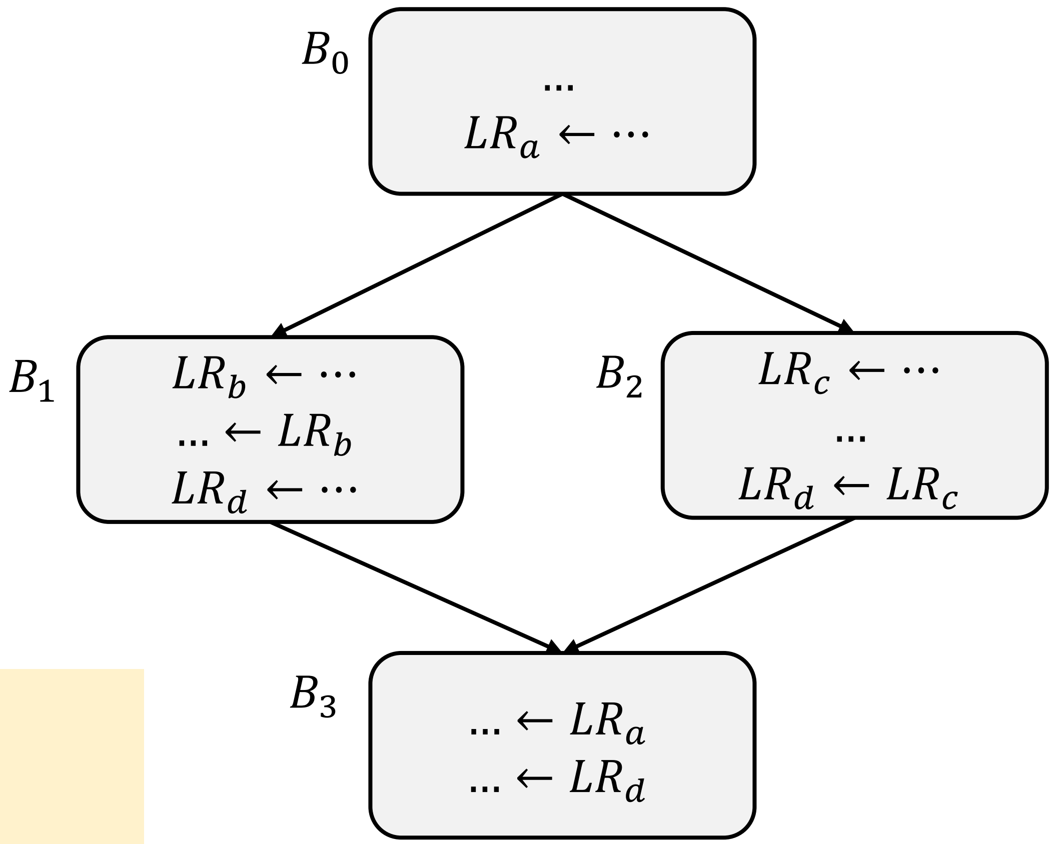
Identifying Global Live Ranges

- Requirement:
  - Group all definitions that reach a single use
  - Group all uses that a single definition can reach
- Assumption: Register allocation operates on the SSA form
  - In SSA, each name is defined once, and each use refers to one definition
  - ϕ functions are used at control flow merge points

Discovering Live Ranges Using SSA Form

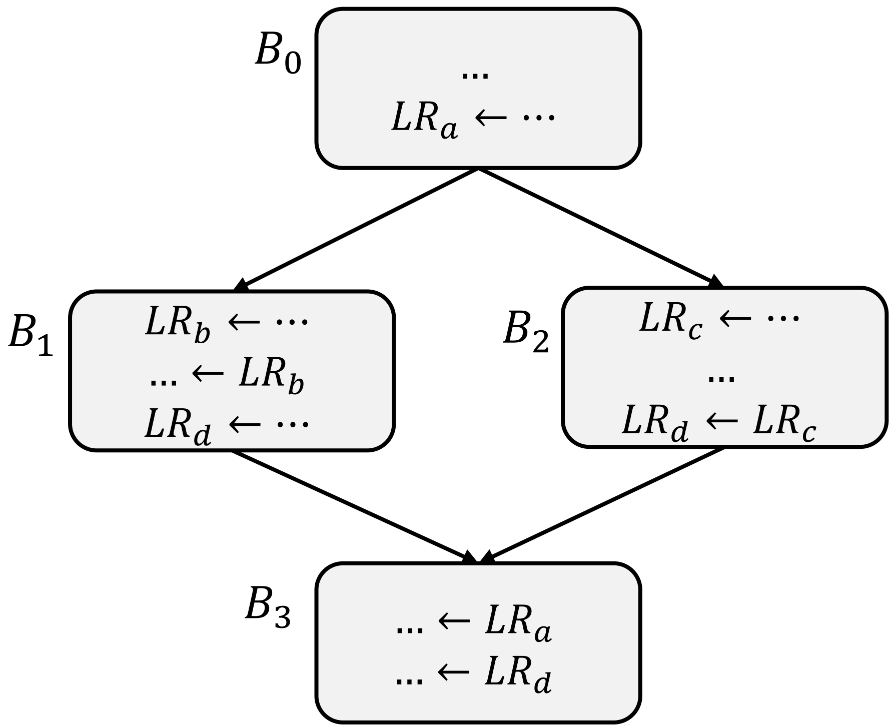


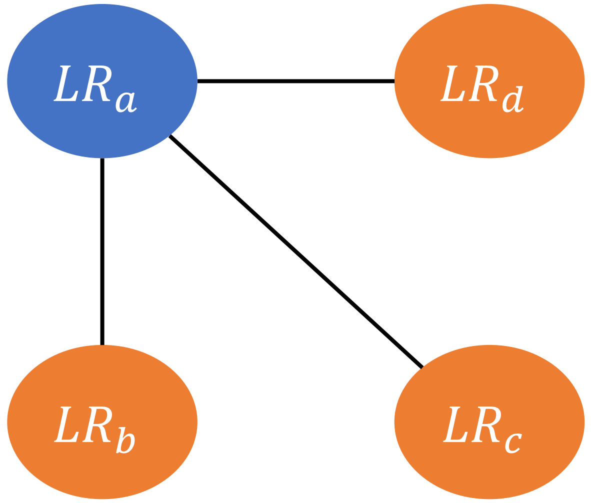
$LR_a = \{a_0\}$
$LR_b = \{b_0\}$
$LR_c = \{c_0\}$
$LR_d = \{d_0, d_1, d_2\}$



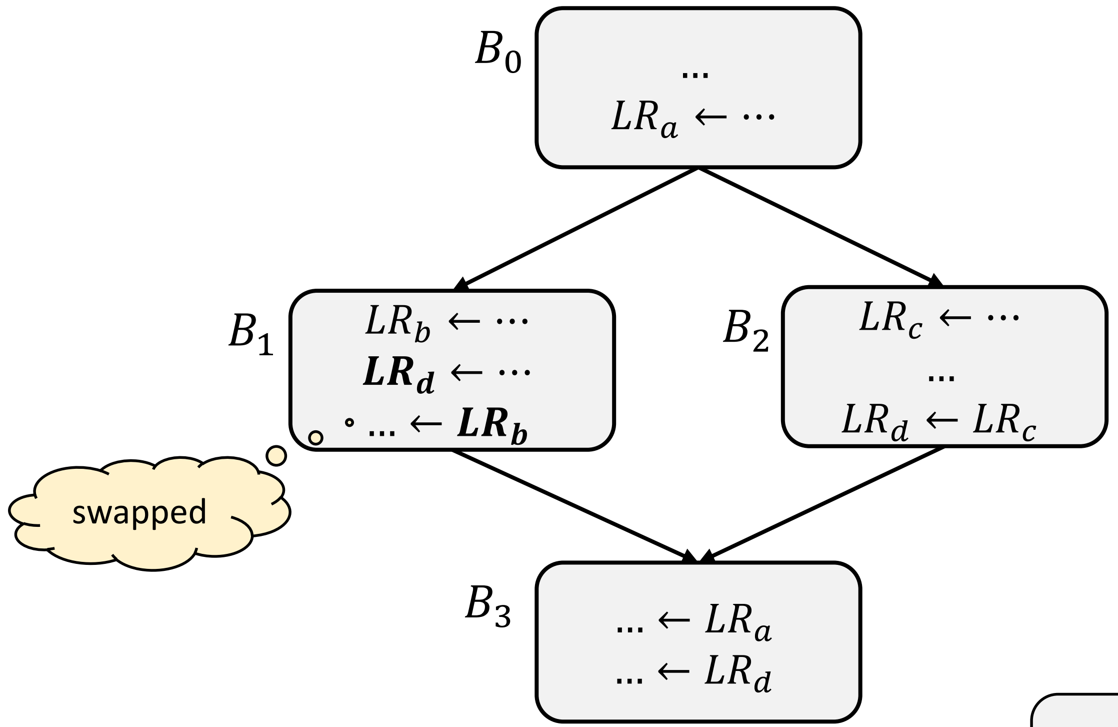
Interferences and the Interference Graph

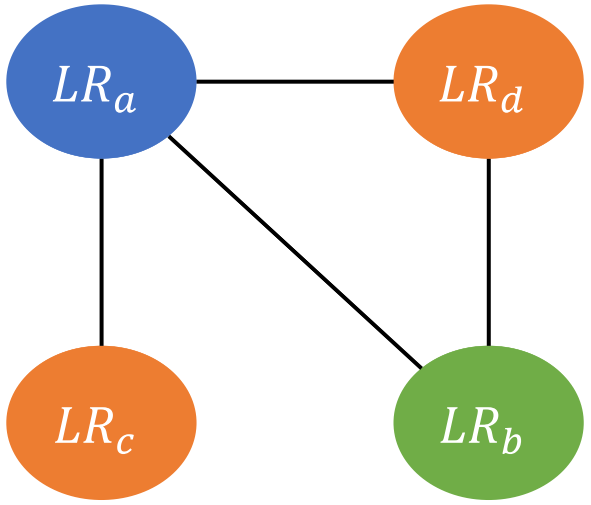
- If there is an operation during which both LR$_i$ and LR$_j$ are live, they cannot reside in the same register
- Two live ranges LR$_i$ and LR$_j$ **interfere** if one is live at the definition of the other and they have different values





Interferences and the Interference Graph





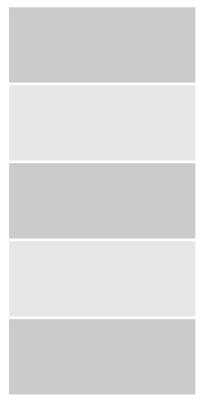
The allocator will need to spill either LR_a or LR_b if only two registers are available.

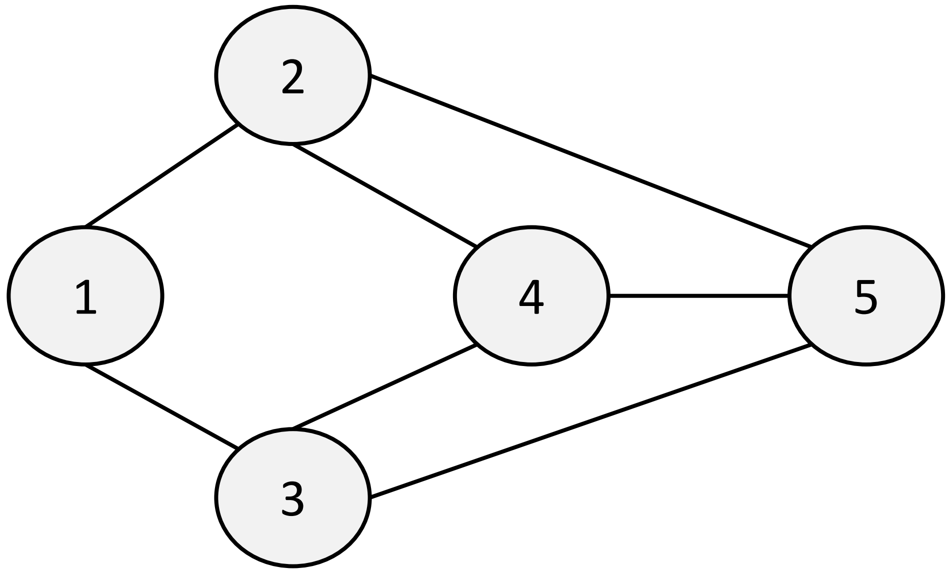
Chaitin’s Algorithm: High-level Idea

- Choose an arbitrary node of degree less than k and put it on the stack
  - Remove that vertex and all its edges from the graph
  - This may decrease the degree of some other nodes and cause some more nodes to have degree less than k
- At some point, if all remaining vertices have degree greater than or equal to k, some node has to be spilled
- If no vertex needs to be spilled, successively pop vertices off stack and color them in a color not used by neighbors (reuse colors as far as possible)

Chaitin's Algorithm: Example 1

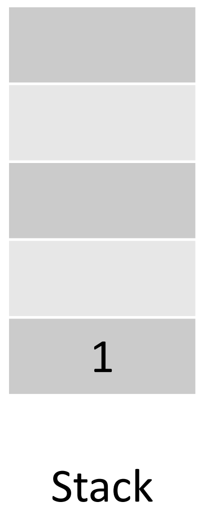
Assume three physical registers are available

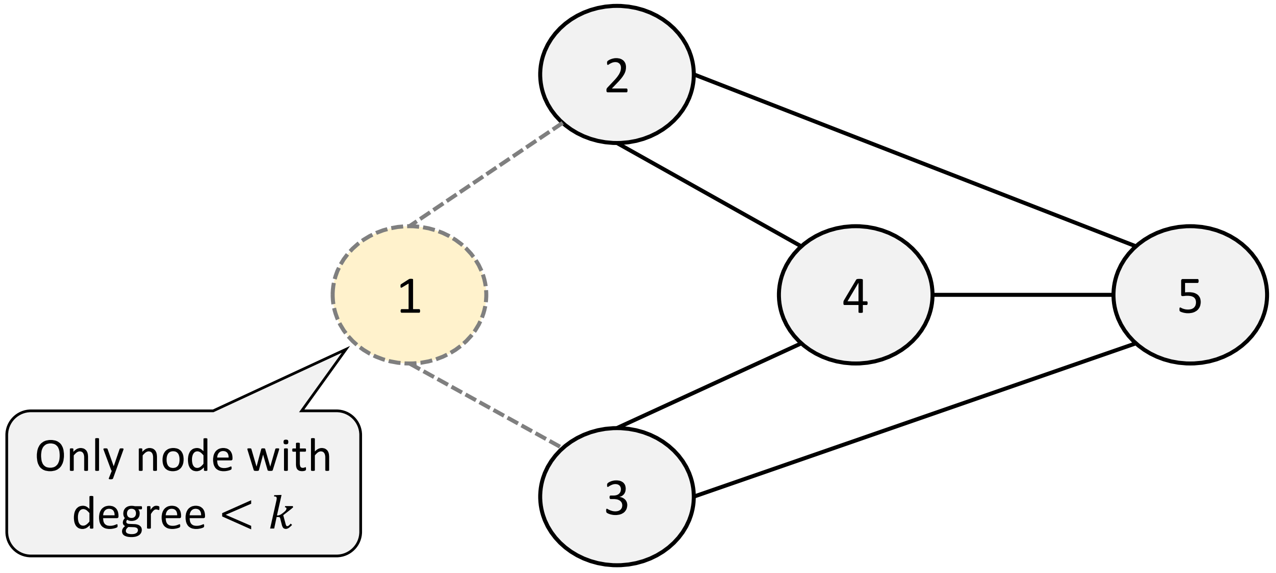




Chaitin’s Algorithm: Example 1

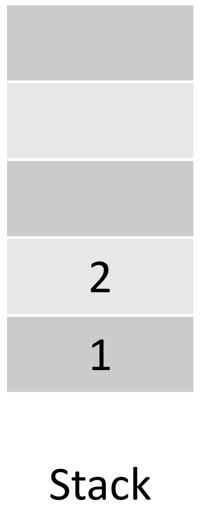
Assume three physical registers are available

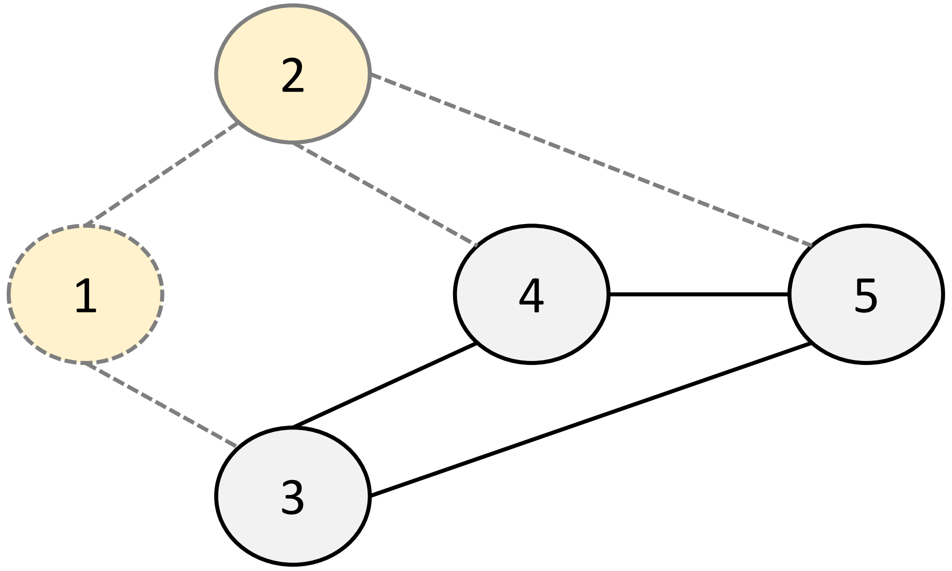




Chaitin's Algorithm: Example 1

Assume three physical registers are available



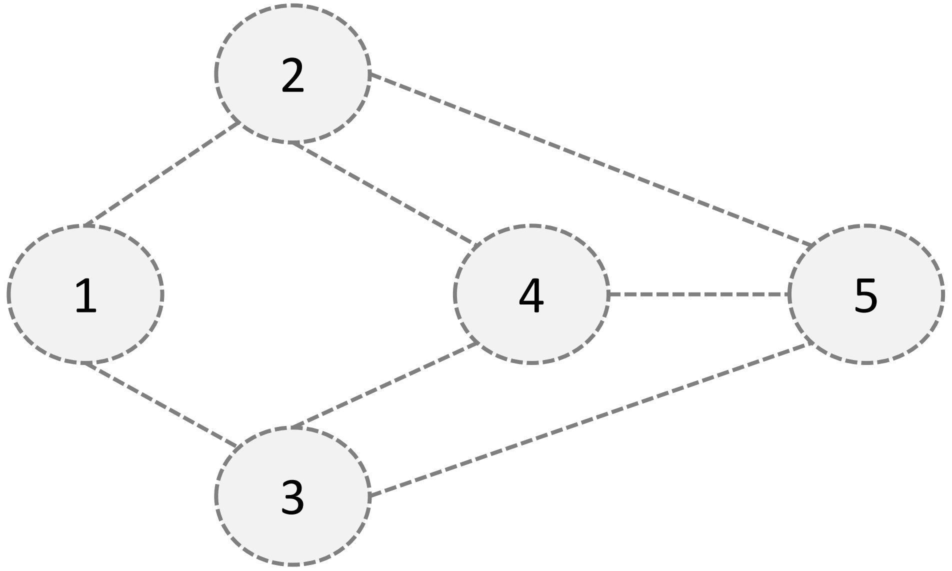


Chaitin’s Algorithm: Example 1

Assume three physical registers are available

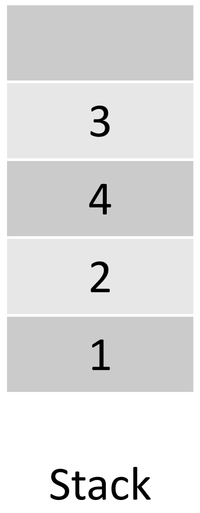
5
3
4
2
1

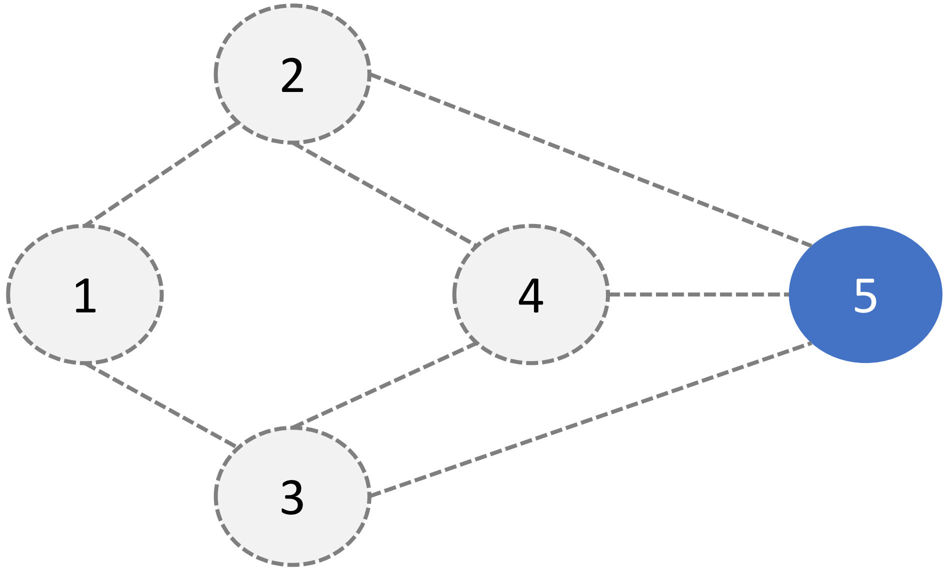
Stack



Chaitin's Algorithm: Example 1

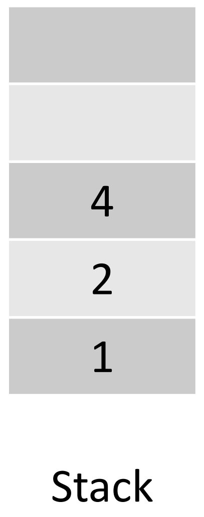
Assume three physical registers are available

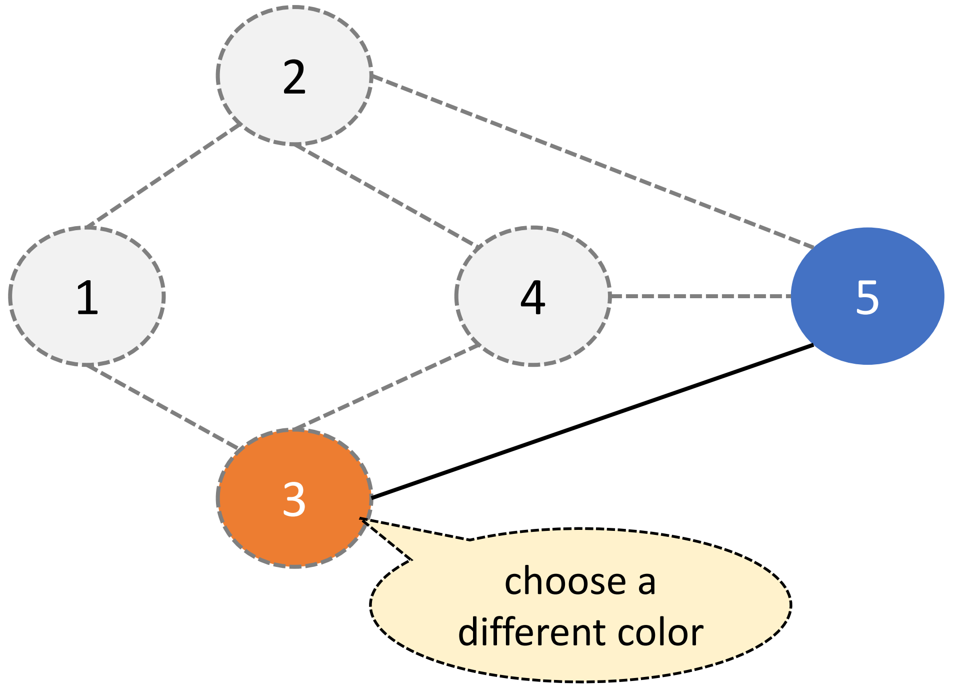




Chaitin’s Algorithm: Example 1

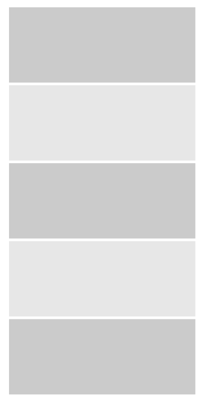
Assume three physical registers are available

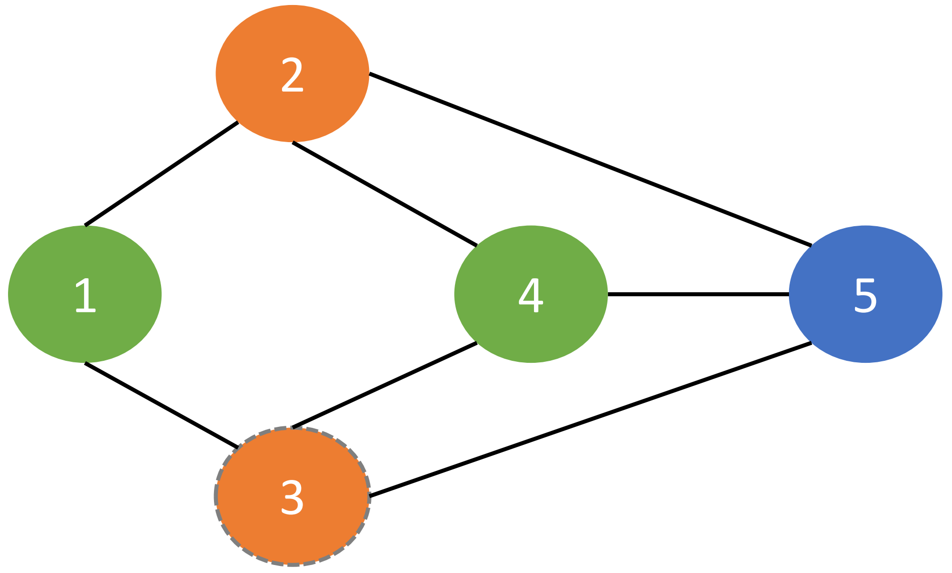




Chaitin's Algorithm: Example 1

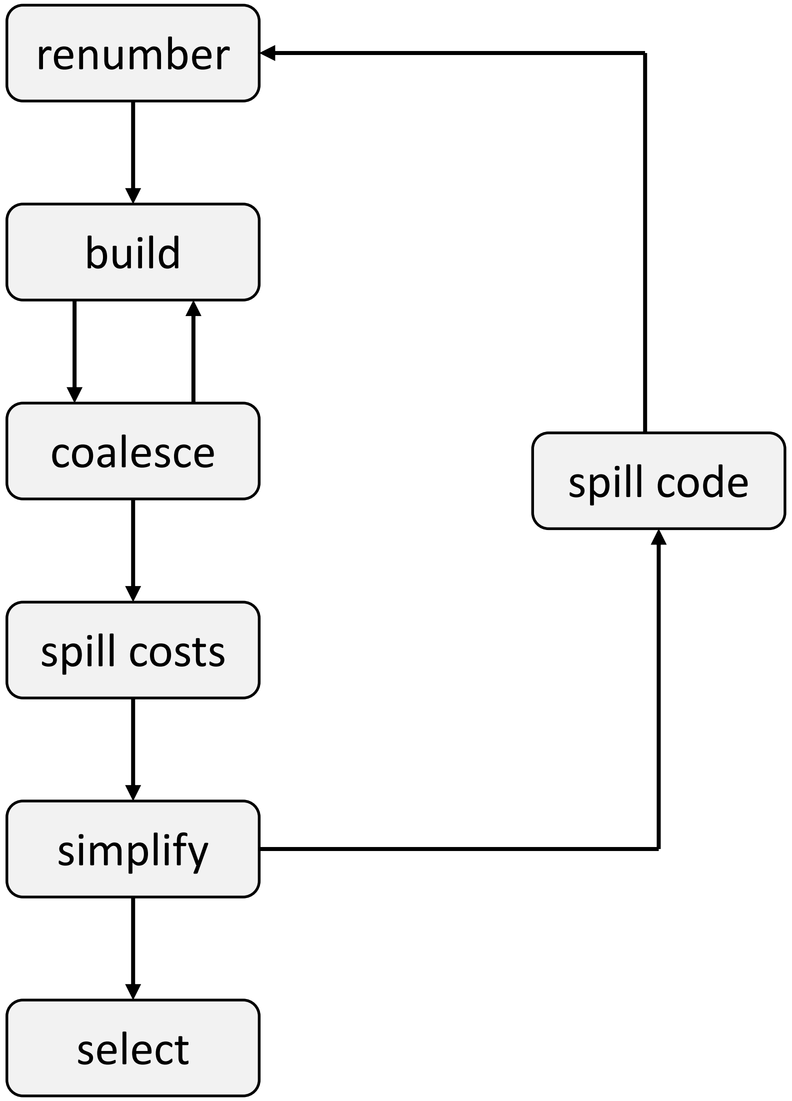
Assume three physical registers are available



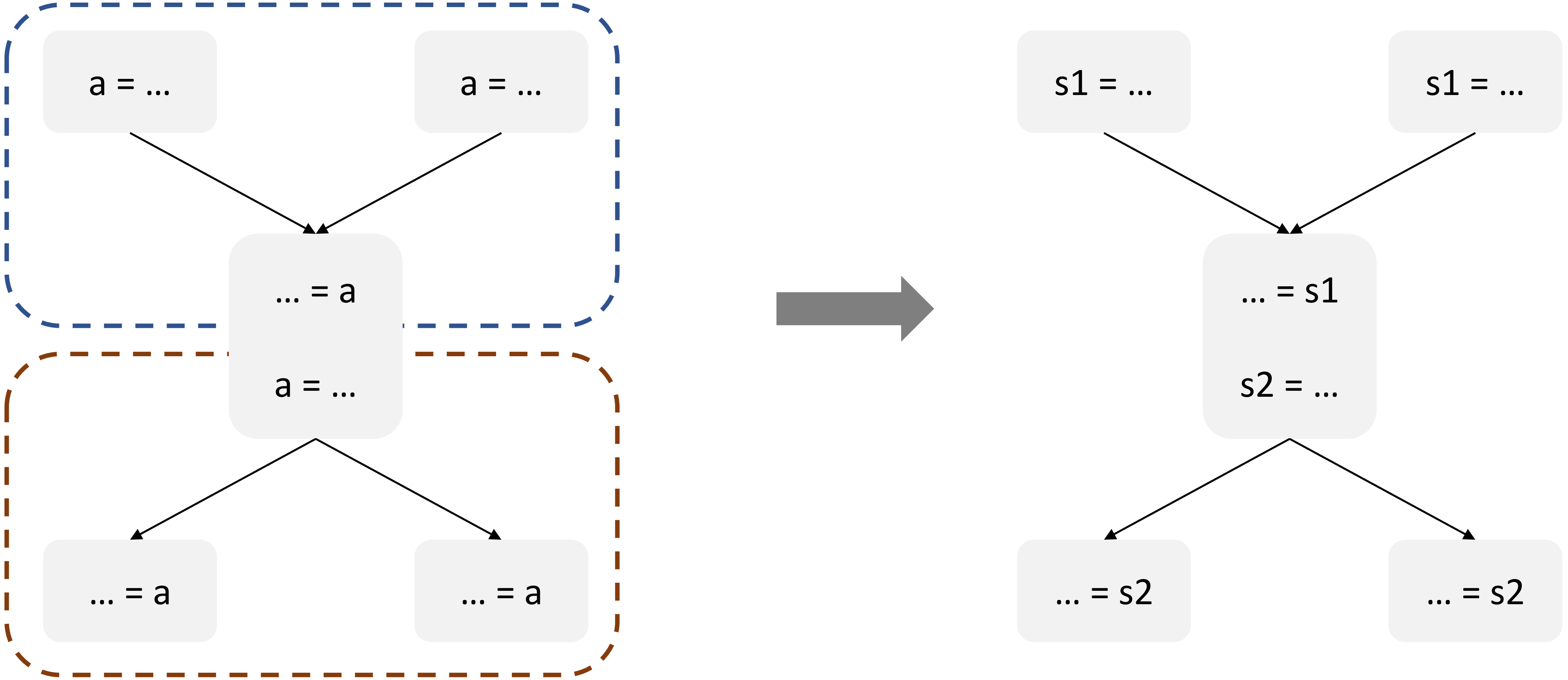


Detailed Steps in Chaitin's Algorithm

i. Renaming – find **all distinct** live ranges and number them uniquely
ii. Build – construct the interference graph
iii. Coalesce – remove unnecessary copy or move operations for non-interfering variables
iv. Spill costs – estimate the dynamic cost of spilling each live range
v. Simplify – construct an ordering of the nodes in the interference graph
vi. Spill code – insert spill operations
vii. Select – assign a register to each variable



Example of Renaming Live Ranges



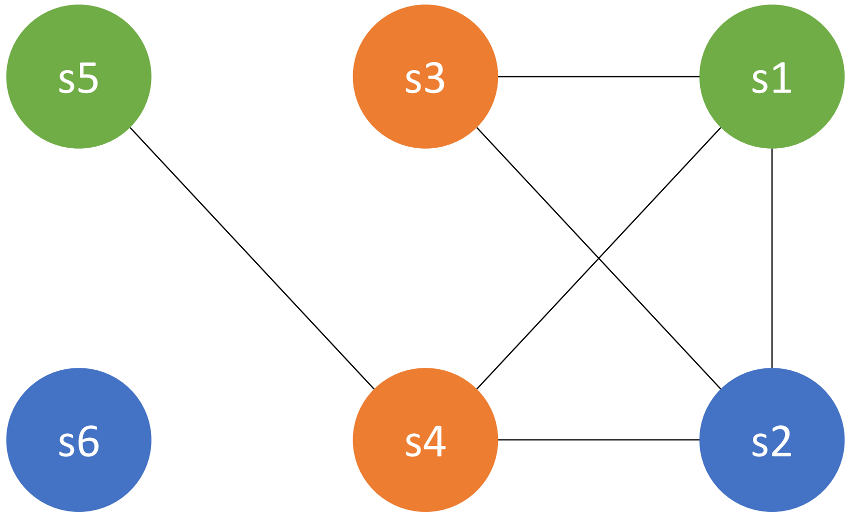
Example of Renaming

```
1. x = 2
2. y = 4
3. w = x + y
4. z = x + 1
5. u = x * y
6. x = z * 2
```

original code

```
s1 = 2       // live range: 1-5
s2 = 4       // live range: 2-5
s3 = s1 + s2 // live range: 3-4
s4 = s1 + 1  // live range: 4-6
s5 = s1 * s2 // live range: 5-6
s6 = s4 * 2  // live range: 6-…
```

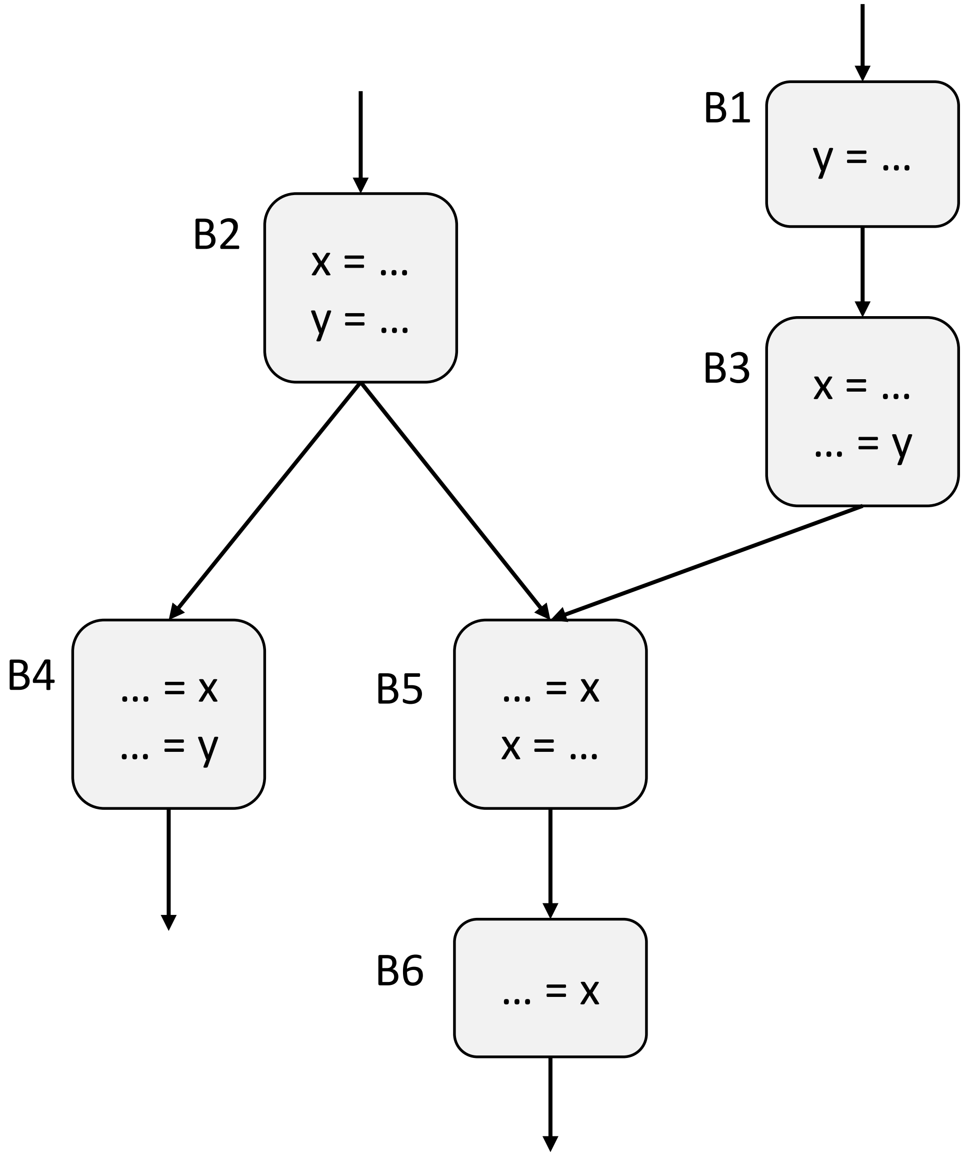
after renaming



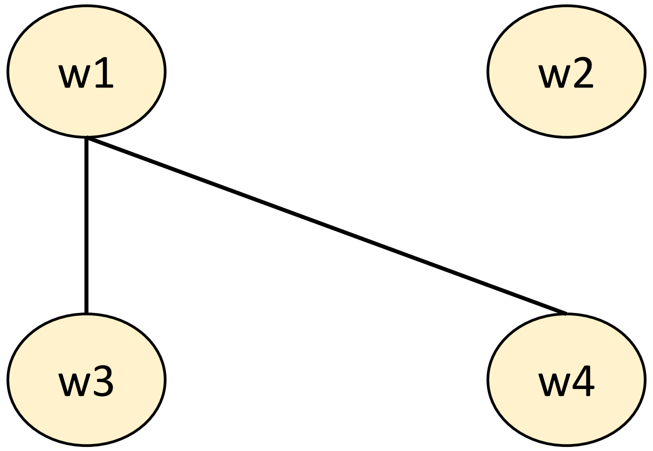
How to model constraints on register usage? For example, say s4 cannot live in register R1.

- Create nodes corresponding to the physical registers in the interference graph
- Add edges between interfering nodes (e.g., s4 and R1)

Live Range and Interference Graph



w1: def of x in B2, def of x in B3, use of x in B4, use of x in B5
w2: def of x in B5, use of x in B6
w3: def of y in B2, use of y in B4
w4: def of y in B1, use of y in B3

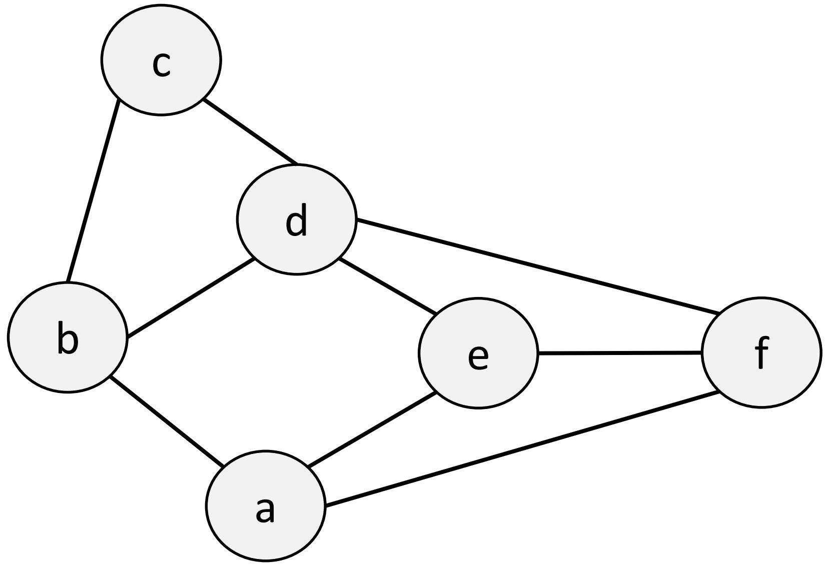


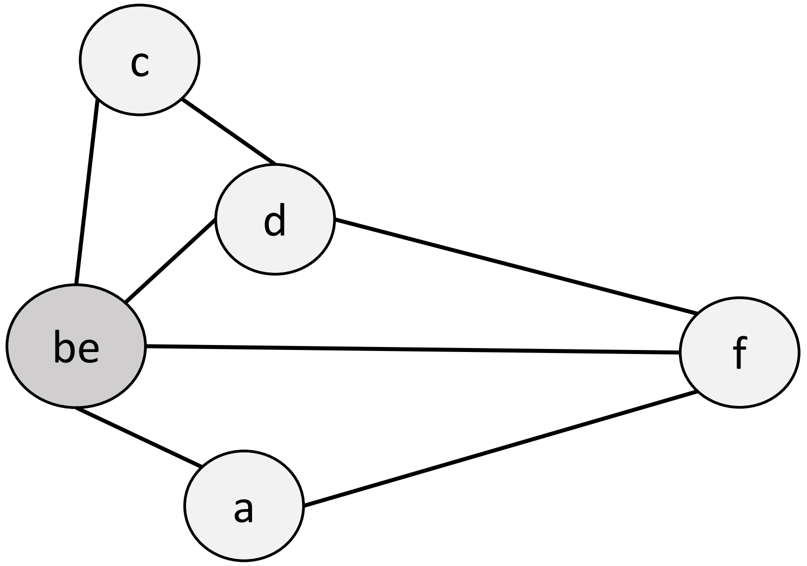
Coalescing

- Consider a copy instruction

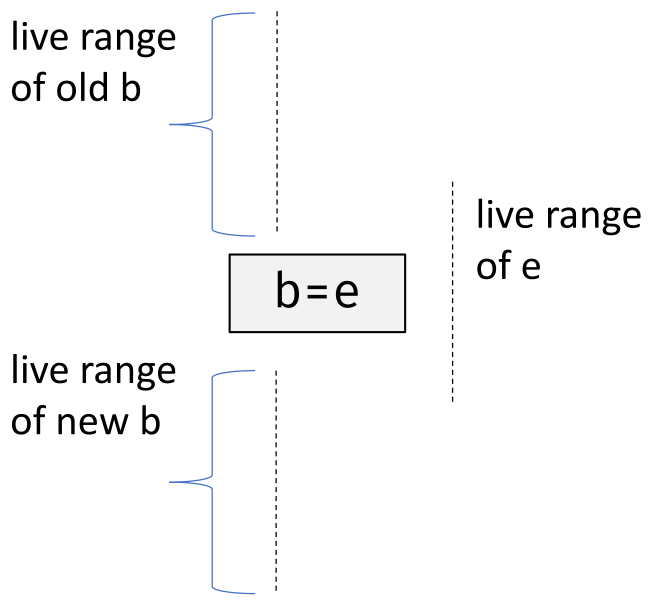
b := e

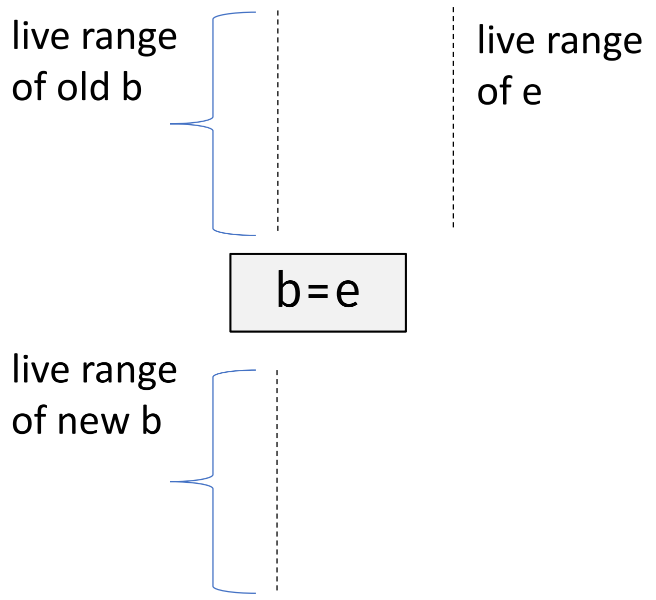
- If the live ranges of b and e do not overlap, then b and e can be given the same register (i.e., color)
    - The copy instruction can then be removed from the final program
- Coalesce by merging b and e into one node that contains edges of both nodes





Copy Subsumption or Coalescing

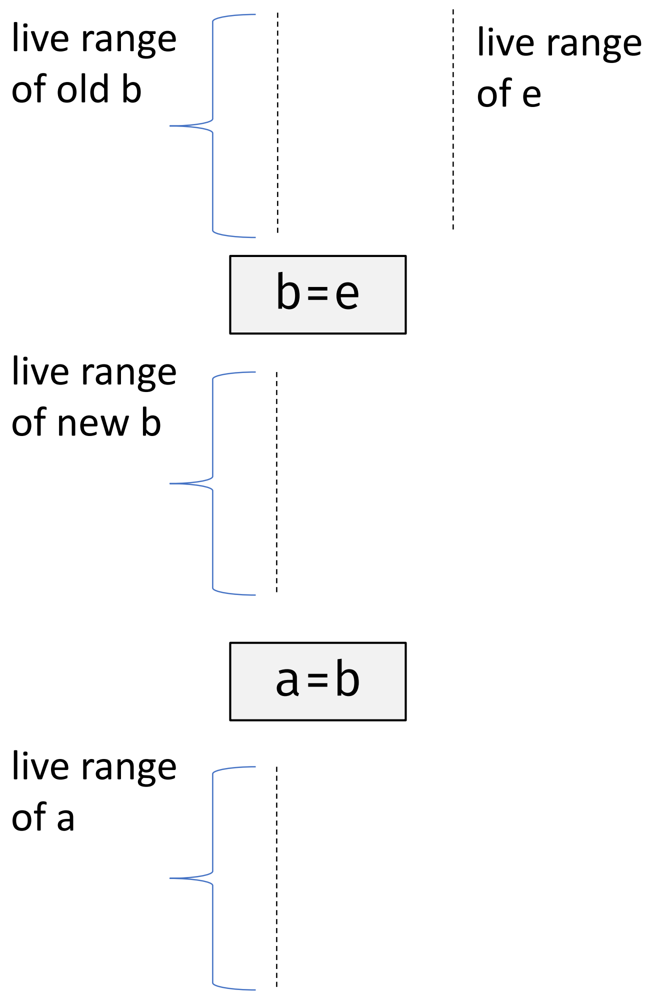




Copy subsumption is not possible, live ranges of e and new def of b interfere

Copy subsumption is possible, live ranges of e and new def of b do not interfere

Repeated Application of Copy Subsumption



Copy subsumption happens twice, once between b and e, and second between a and b. e, b, and a can all be given the same register.

Coalescing

- Coalesce all possible copy instructions
- Rebuild the interference graph
  - May offer further opportunities for coalescing
  - Build-coalesce phase is repeated till no further coalescing is possible
- Coalescing reduces the size of the interference graph and possibly reduces spilling

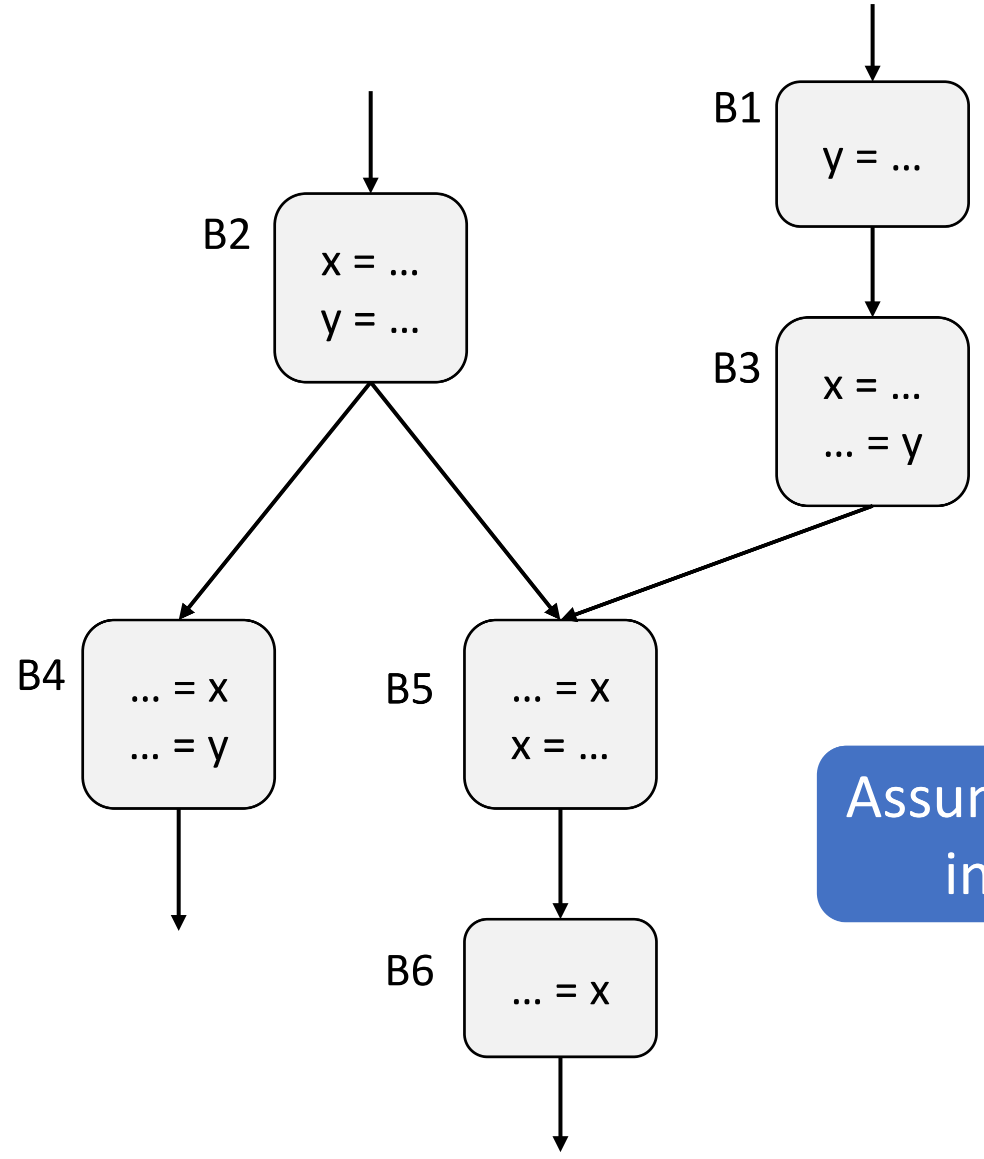
Simplification and Estimating Spill Cost

- If a node n in the interference graph has degree less than k, remove n and all its edges from the graph and place n on a stack
- We need to spill a node when no such nodes are available
  - Implies loading a variable x into a register before every use of x and storing x from register into memory at every definition of x
- Estimate of spill cost for a live range v: $cost(v) = \sum_{\substack{\text{all store or load} \\ \text{operations in } v}} c \times 10^d$

where c is the cost of the operation, d is the loop nesting depth, and 10 is the average number of loop iterations (assumption)

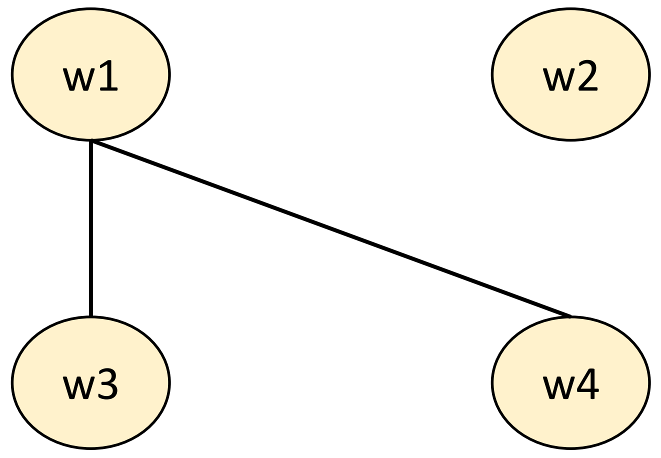
- The node to be spilled is the one with $\min(cost(v)/\deg(v))$

Effect of Spilling on Live Range

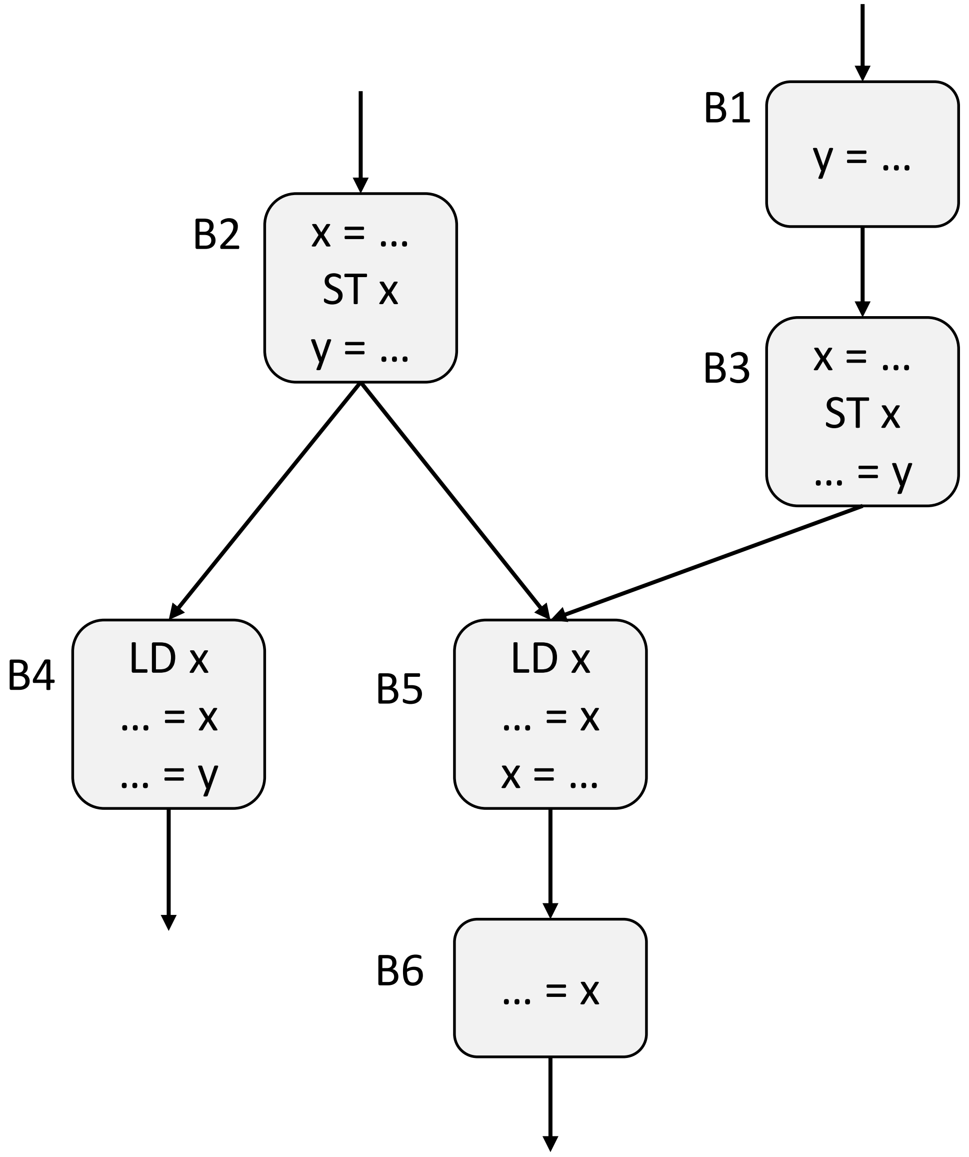


w1: def of x in B2, def of x in B3, use of x in B4, use of x in B5
w2: def of x in B5, use of x in B6
w3: def of y in B2, use of y in B4
w4: def of y in B1, use of y in B3

Assume that x is spilled in live range w1

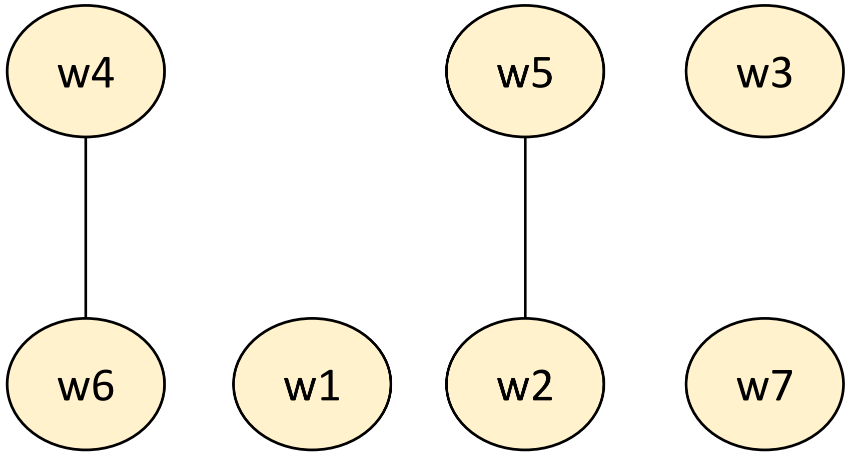


Effect of Spilling on Live Range



w1: def of x in B2, ST of x in B2
w2: def of x in B3, ST of x in B3
w3: def of x in B5, use of x in B6
w4: def of y in B2, use of y in B4
w5: def of y in B1, use of y in B3
w6: LD of x in B4, use of x in B4
w7: LD of x in B5, use of x in B5

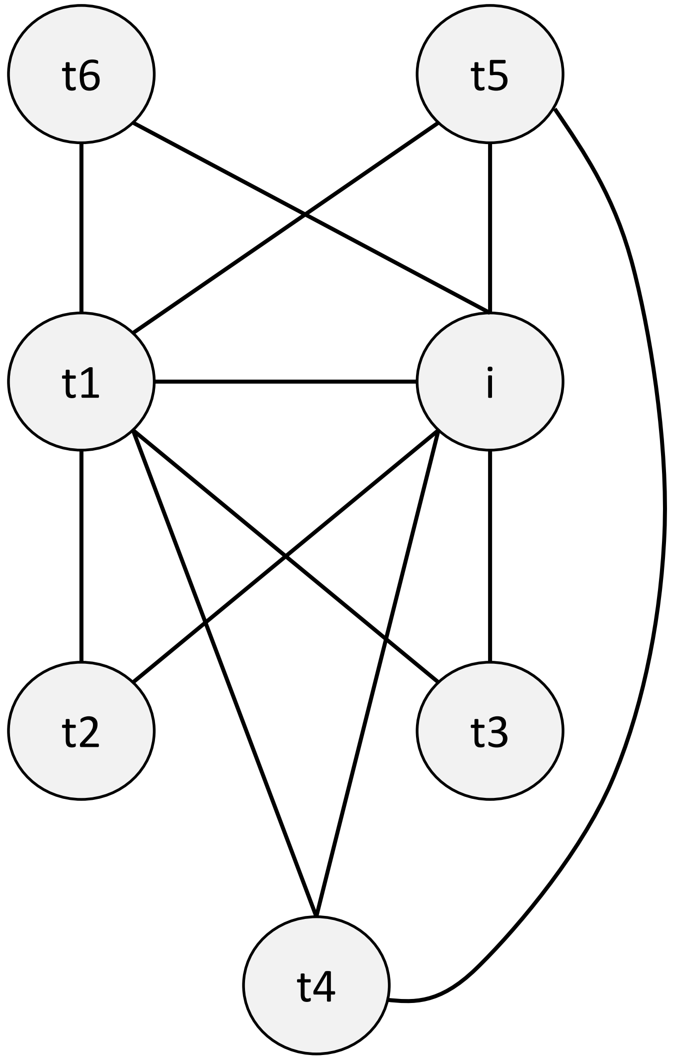
creates many short live ranges



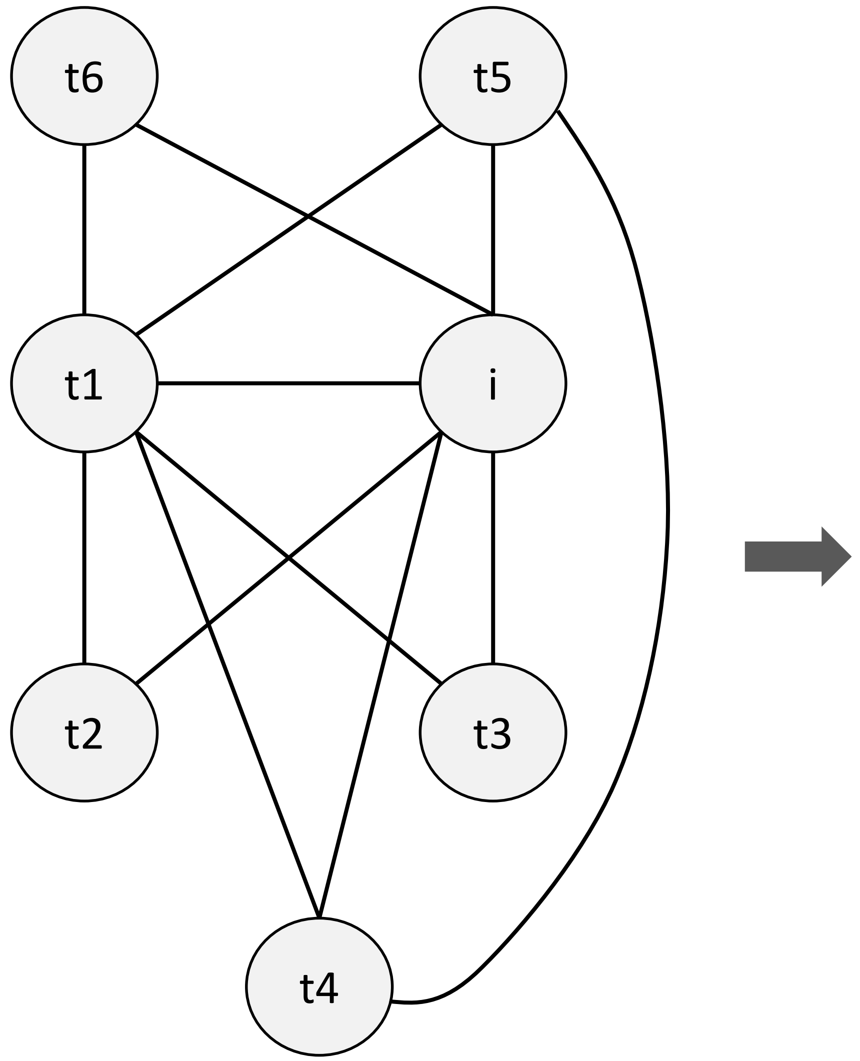
Chaitin’s Algorithm: Example 2

```
1.       t1=202
2.       i=1
3.  L1:  t2=i>100
4.       if t2 goto L2
5.       t1=t1-2
6.       t3=addr(a)
7.       t4=t3-4
8.       t5=4*i
9.       t6=t4+t5
10.       *t6=t1
11.       i=i+1
12.       goto L1
13. L2:
```

Variable	Live Range
t1	1-10
i	2-11
t2	3-4
t3	6-7
t4	7-9
t5	8-9
t6	9-10

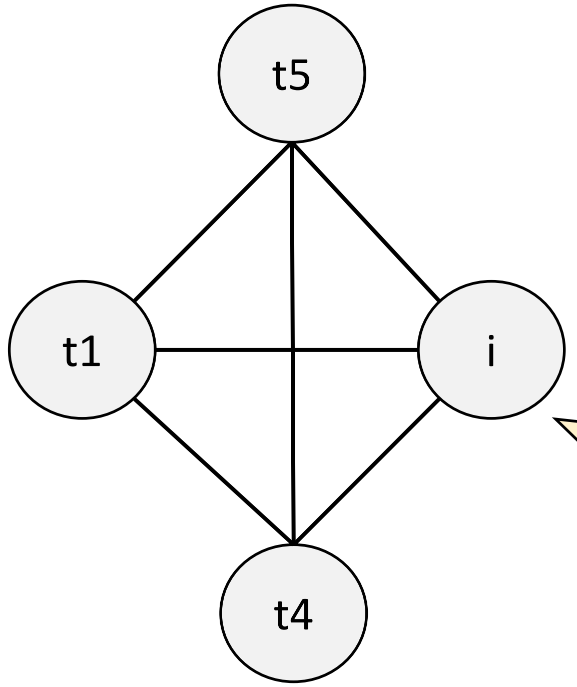


Chaitin's Algorithm: Example 2



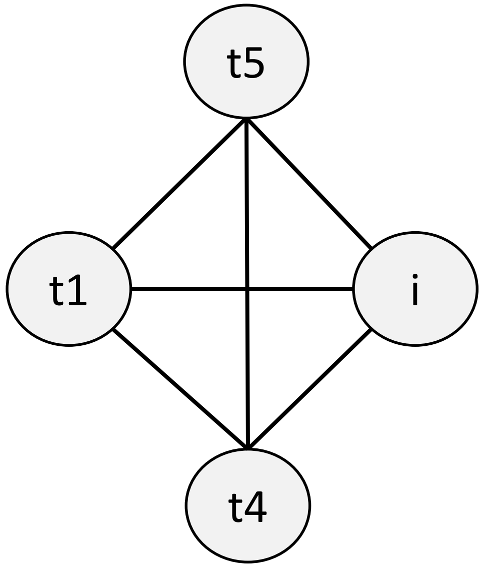
Assume there are three physical registers.

Nodes t6, t2, and t3 are pushed on to the stack during graph reduction.



cannot be reduced further, spilling is necessary

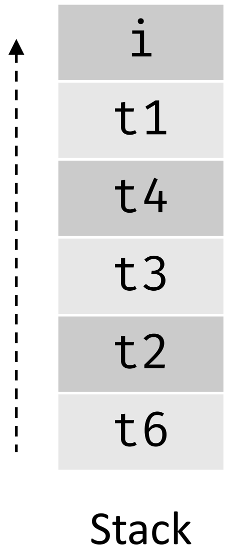
Chaitin’s Algorithm: Example 2

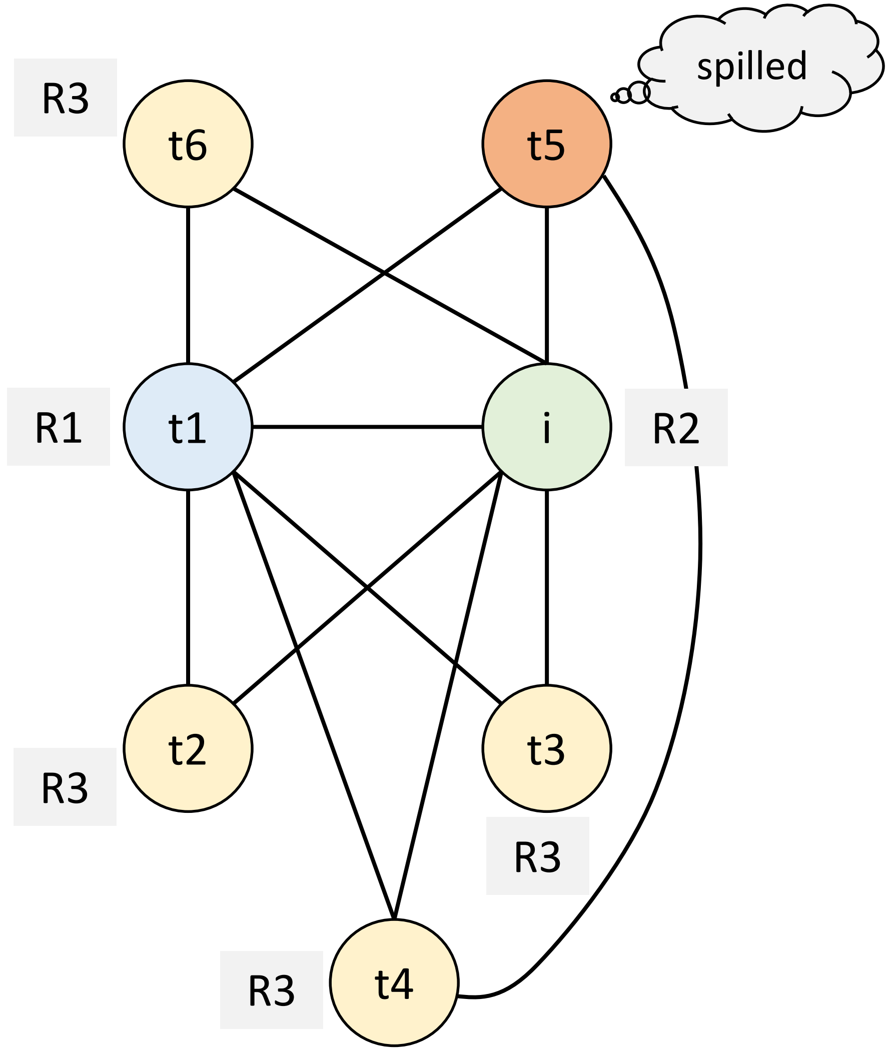


Node v	Cost(v)	deg(v)	Cost(v) /deg(v)
t1	1+(1+1+1)*10=31	3	10
i	1+(1+1+1+1)*10=41	3	14
t4	(1+1)*10=20	3	7
t5	(1+1)*10=20	3	7

Let us pick t5 for spilling

Chaitin's Algorithm: Example 2



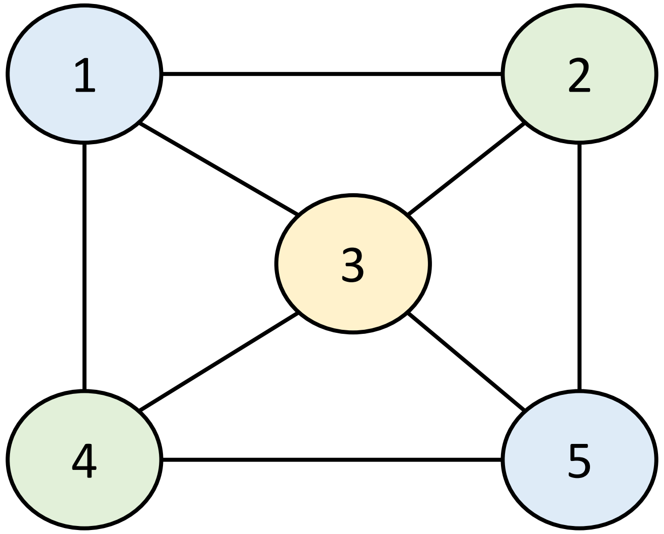


```
1.        R1=202
2.        R2=1
3.  L1:  R3=R2>100
4.        if R3 goto L2
5.        R1=R1-2
6.        R3=addr(a)
7.        R3=R3-4
8.        t5=4*R2
9.        R3=R3+t5
10.       *R3=R1
11.       R2=R2+1
12.       goto L1
13. L2:
```

Problem with Chaitin's Algorithm

- Constructing and modifying the interference graphs is very costly

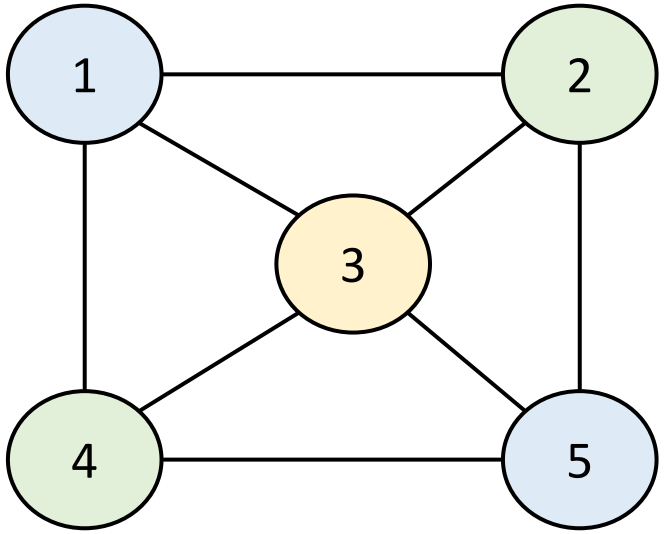
- Careful coalescing
  - Do not coalesce if coalescing increases the degree of a node to $> k$

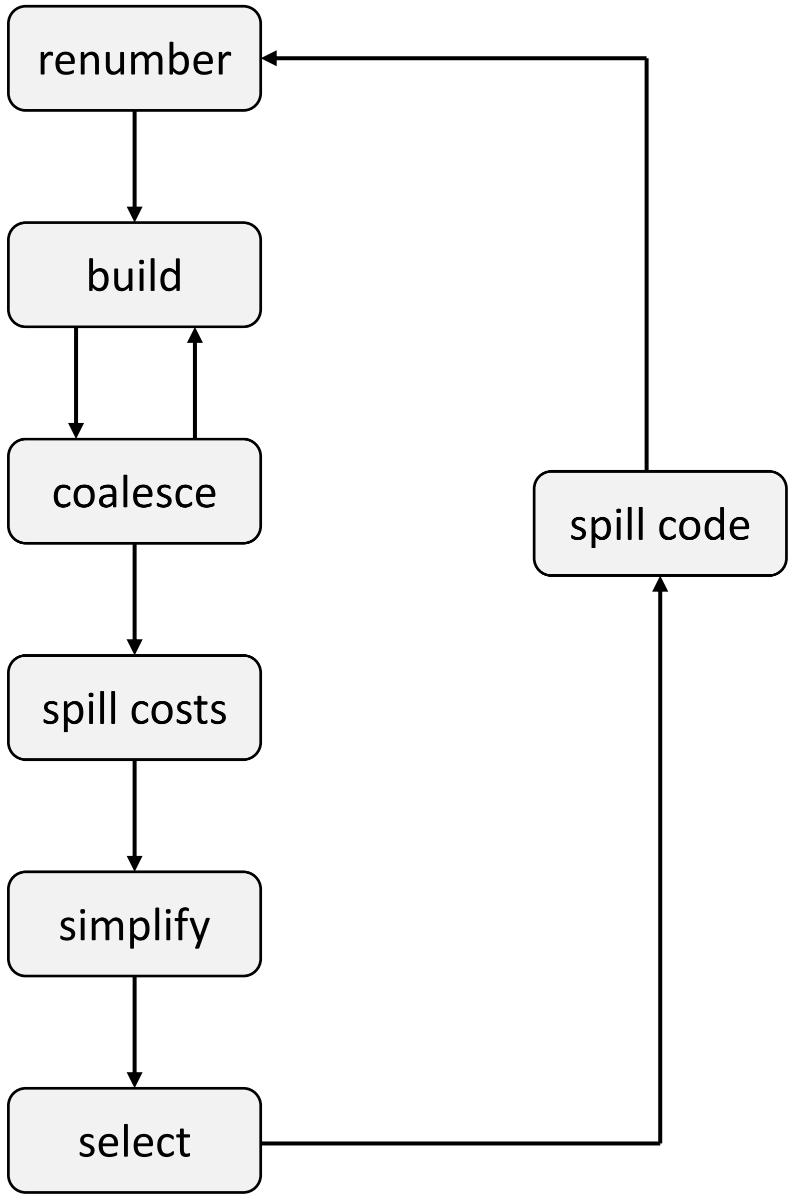


There is a 3-coloring, but Chaitin's heuristic does not find it

Brigg's Optimistic Allocator

- Instead of spilling, Briggs pushes the spill candidate on the stack, hoping there will be a color available
    - Spill a node only when it is popped and there are no colors available





References

- A. Aho et al. Compilers: Principles, Techniques, and Tools, 1st edition, Chapter 9.7.
- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, Chapter 8.8.
- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, Chapter 13.
- Register Allocation, Wikipedia. https://en.wikipedia.org/wiki/Register_allocation
- Christian Collberg. CSc 553: Register Allocation, University of Arizona.
- Y. N. Srikanth. NPTEL Course on Compiler Design: Global Register Allocation, IISc.
- Hugh Leather. Compiler Optimization: Register Allocation, University of Edinburgh.
- Preston Briggs. Register Allocation, University of Washington.