编译原理

11. 寄存器分配

rainoftime.github.io

浙江大学
计算机科学与技术学院

Content

Outline

1. Introduction
2. Register Allocation via Graph Coloring: Overview
3. Coloring by Simplifications

1. Introduction

Register Allocation

- Speed: Registers > Memory
  - Registers are 2x – 7x faster than cache
- Physical machines have limited number of registers
- **Register allocation**
  - ∞ virtual registers ➞ k physical registers

Register Allocation

- Speed: Registers > Memory
  - Registers are 2x – 7x faster than cache
- Physical machines have limited number of registers
- **Register allocation**
  - ∞ virtual registers $\rightarrow$ k physical registers
- **Requirement**
  - Produce correct code using k or fewer registers

Register Allocation

- Speed: Registers > Memory
  - Registers are 2x – 7x faster than cache
- Physical machines have limited number of registers
- **Register allocation**
  - ∞ virtual registers $\rightarrow$ k physical registers
- **Requirement**
  - Produce correct code using k or fewer registers
  - Minimize loads, stores, and space to hold spilled values

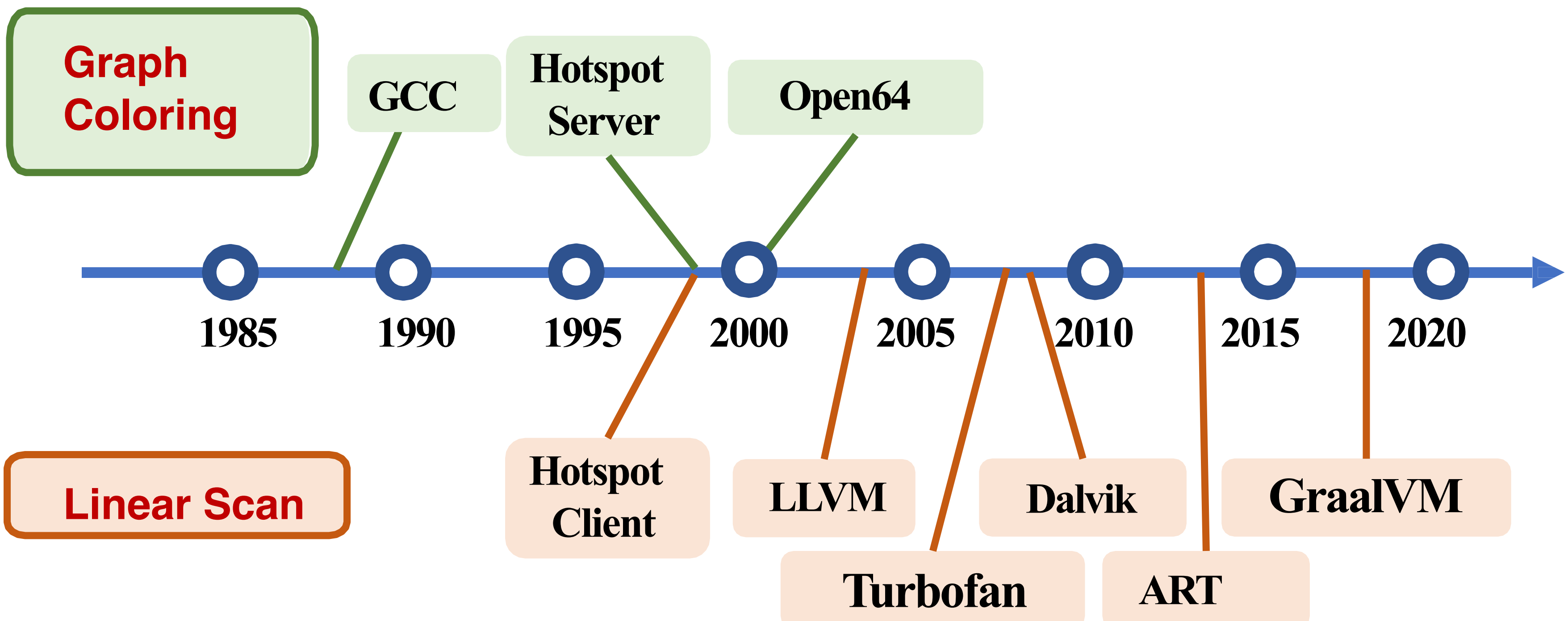
Register Allocation

- Speed: Registers > Memory
  - Registers are 2x – 7x faster than cache
- Physical machines have limited number of registers
- **Register allocation**
  - ∞ virtual registers $\rightarrow$ k physical registers
- **Requirement**
  - Produce correct code using k or fewer registers
  - Minimize loads, stores, and space to hold spilled values
  - Efficient register allocation

    (typically, O(n) or O(nlogn))

Example: Register Allocation Algorithms

分配效果好、但运行时间长、常见于传统编译器



算法运行时间短，分配效果接近图着色、常见于现代编译器

Example: Registers Allocation in LLVM

- **Basic**：线性扫描算法的改进，使用启发式的顺序对寄存器 进行生存期赋值
- **Fast**：顺序扫描每条指令，对其中的变量进行寄存器分配， 当没有寄存器可以分配时，选择溢出代价最小的寄存器进 行溢出操作
- **Greedy**：线性扫描算法的改进，Basic分配器的高度优化的实现，合并了全局生存期分割，努力最小化溢出代码的成本
- **PBQP**：基于分区布尔二次编程（PBQP）的寄存器分配器. 构造一个表示寄存器分配问题的PBQP问题，使用PBQP求解器解决该问题，并将该解决方案映射回寄存器分配

Register Allocation

- **"Naïve" register allocation**
- **Local register allocation**
  - Basic block level
  - Does not capture reuse of values across multiple basic blocks
- **Global register allocation**
  - Function-level
  - Often uses the **graph-coloring** paradigm

Register Allocation via Graph Coloring

- Global register allocation often uses the **graph-coloring** paradigm
  1. Build a **conflict/interference graph**
  2. Find a **k-coloring** for the graph, or change the code to a nearby

2. Register Allocation Via Graph Coloring: Overview

- Interference Graph
- Register Allocation

Interference

- We have a set of temporaries (virtual registers) *a*, *b*, *c*, ... and machine registers r1, ..., rk. How to assign registers to temporaries?
- A condition that prevents *a* and *b* from being allocated to the same register is called an interference.

Interference

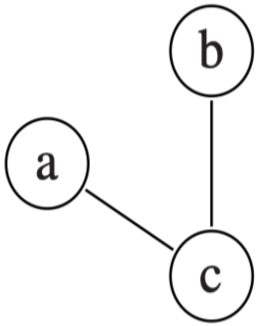
- We have a set of temporaries (virtual registers) *a*, *b*, *c*, ... and machine registers r1, ..., rk. How to assign registers to temporaries?
- A condition that prevents *a* and *b* from being allocated to the same register is called an interference.
- **Two types of interferences:**
  – Overlapping live ranges
  – When *a* must be generated by an instruction that cannot address register r1, then *a* and r1 interfere

Interference Graph

- **Interference Graph**
  - Nodes of the graph = virtual registers
  - Edges connect virtual registers that interfere with one another

	a	b	c
a			**x**
b			**x**
c	**x**	**x**	

(a) Matrix



(b) Graph

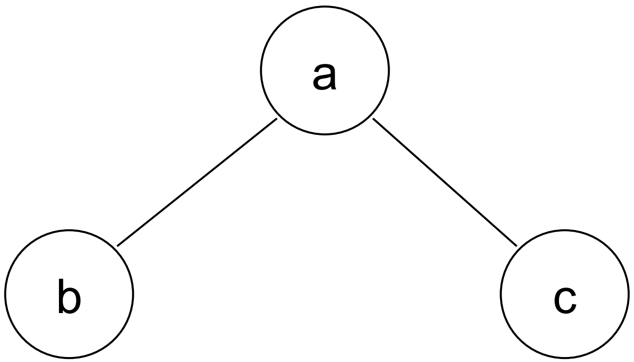
a matrix: x marking interferences

undirected graph

Example: Interference Graph

- Two values CANNOT be mapped to the same register wherever they are both live
- Two variables can be allocated to same register if no edge connects them

Instructions	**Live vars**
	a
1: b = a + 2	
	a, b
2: c = b * b	
	a, c
3: b = c + 1	
	a, b
4: return b * a	



Special Treatment of MOVE instructions

- Do not create artificial interferences between the source and destination of a MOVE. Consider:

```
t  := s              (copy)
...
x := ... s ...       (use of s)
...
y := ... t ..        (use of t)
```

  - Normally, we would make an interference edge (s, t).
  - But we do not need separate registers for s and t, since they contain the same value.
  - **Solution**: not to add an interference edge (s, t) in this case.

Special Treatment of MOVE instructions

- Do not create artificial interferences between the source and destination of a MOVE. Consider:

```
t  := s              (copy)
...
x := ... s ...       (use of s)
...
y := ... t ..        (use of t)
```

```
t  := s              (copy)
t := ...
x := ... s ...       (use of s)
...
y := ... t ...       (use of t)
```

- Normally, we would make an interference edge (s, t).
- But we do not need separate registers for s and t, since they contain the same value.
- **Solution**: not to add an interference edge (*s*, *t*) in this case.
- However, if there is a later (nonmove) definition of t while s is still live, we will create the inference edge (t, s)

Interference Graphs

Therefore, the way to add interference edges for each new definition is as follows:

1. At any nonmove instruction n that defines a variable a, where out[n] = {b1, ..., bj}
   - add interference edges (a, b1), ..., (a, bj).
2. At a move instruction a := c, where {b1, ..., bk} are the live-out set
   - add interference edges (a, b1), ..., (a, bk) for any bi that is not the same as c.

2. Register Allocation Via Graph Coloring: Overview

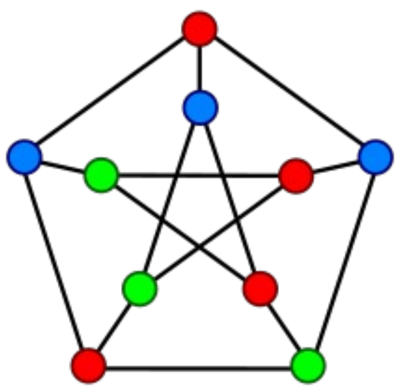
- Interference Graph
- Register Allocation

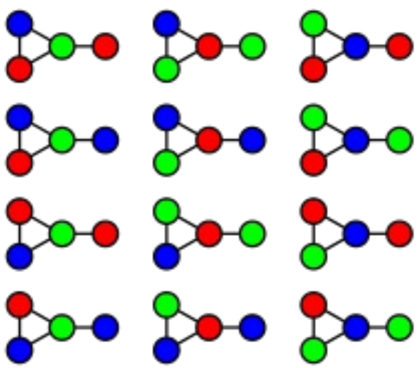
Graph Coloring

- **Vertex Coloring**: assign a color to each vertex such that no edge connects vertices with the same color.

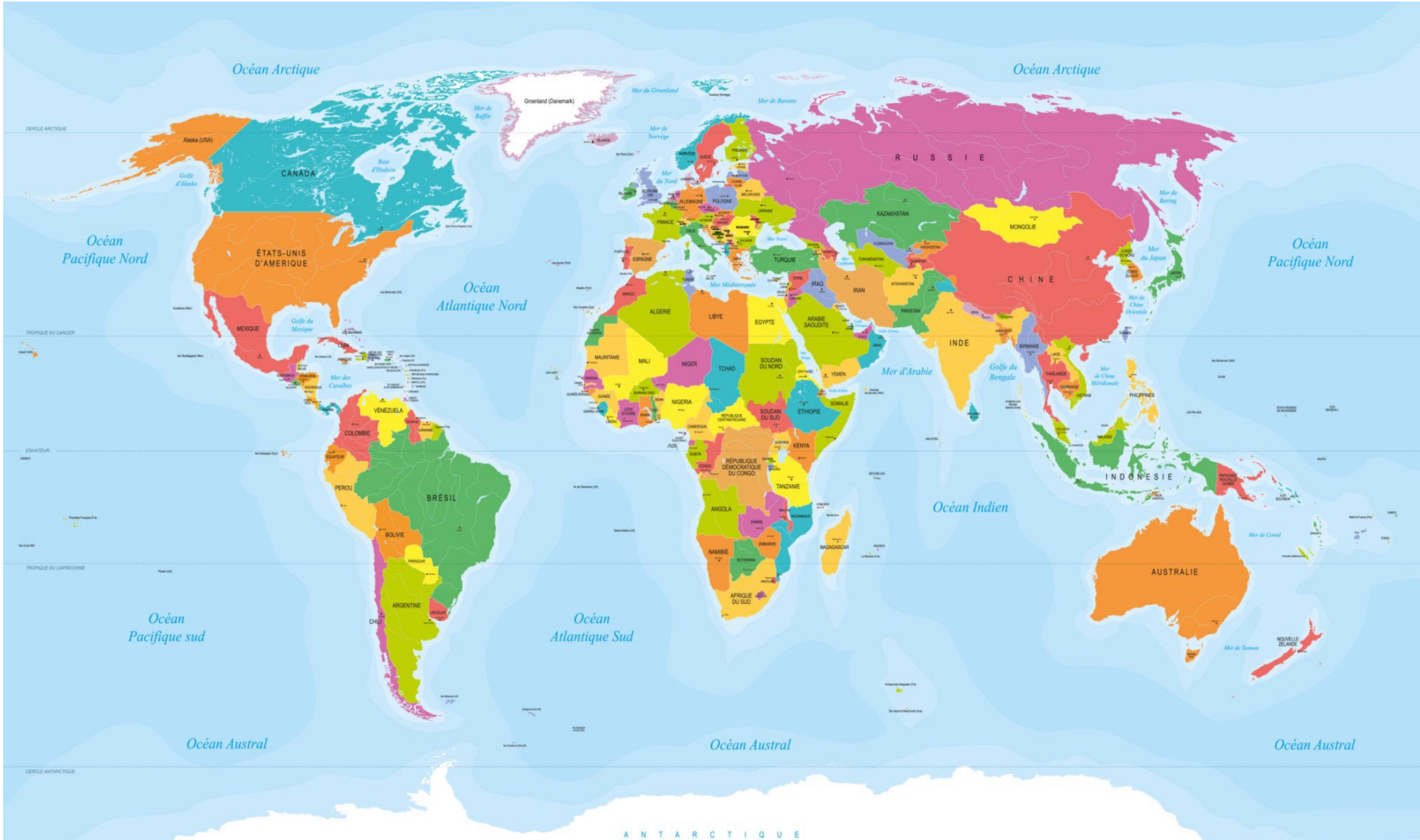
Graph Coloring

- **Vertex Coloring**: assign a color to each vertex such that no edge connects vertices with the same color.
- **K-Coloring**: a coloring using at most k colors





3-coloring in 12 ways


Océan Arctique
Océan Arctique
Groenland (Danemark)
Alaska (USA)
CANADA
ÉTATS-UNIS D'AMERIQUE
MEXIQUE
Océan Pacifique Nord
Océan Pacifique Nord
Océan Atlantique Nord
RUSSIE
KAZAKHSTAN
MONGOLIE
CHINE
TURQUIE
IRAN
ARABIE SAOUDITE
INDE
Mer Méditerranée
ALGERIE
LIBYE
EGYPTE
MALI
NIGER
TCHAD
SOUDAN DU NORD
NIGERIA
ETHIOPIE
KENYA
RÉPUBLIQUE DÉMOCRATIQUE DU CONGO
TANZANIE
ANGOLA
NAMIBIE
AFRIQUE DU SUD
MADAGASCAR
Mer d'Arabie
Golfe du Bengale
INDONESIE
Océan Indien
AUSTRALIE
VÉNÉZUELA
COLOMBIE
PEROU
BRÉSIL
BOLIVIE
ARGENTINE
Océan Pacifique sud
Océan Atlantique Sud
Océan Austral
Océan Austral
Océan Austral
ANTARCTIQUE

K-Coloring for Register Allocation

- **Map graph vertices onto virtual registers**
- **Map colors onto physical registers**

K-Coloring for Register Allocation

- **Map graph vertices onto virtual registers**
- **Map colors onto physical registers**

1. From live ranges construct an **interference graph**

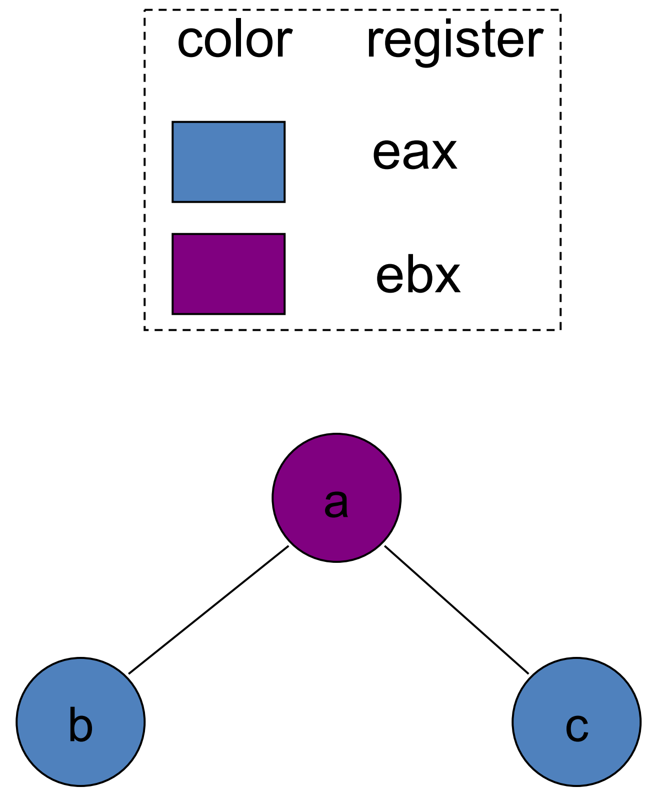
K-Coloring for Register Allocation

- **Map graph vertices onto virtual registers**
- **Map colors onto physical registers**

1. From live ranges construct an **interference graph**
2. Color the graph so that **no two neighbors have the same color**

Example: K-Coloring for Register Allocation

Instructions	Live vars
	a
b = a + 2	
	a,b
c = b * b	
	a,c
b = c + 1	
	a,b
return b * a	



Interference Graph

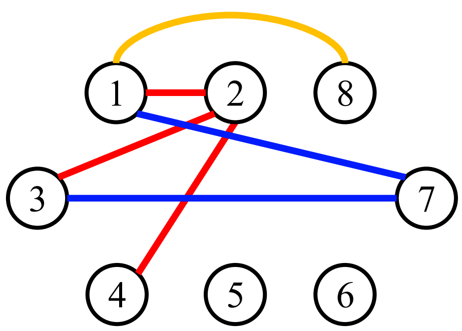
K-Coloring for Register Allocation

- **Map graph vertices onto virtual registers**
- **Map colors onto physical registers**

1. From live ranges construct an **interference graph**
2. Color the graph so that **no two neighbors have the same color**
3. If graph needs more than ***k*** colors – **Spilling**

Example: Spilling

- If we can use k, e.g., 4, colors, to color the graph, the 8 virtual registers can be replaced by 4 physical registers.
- If we have to use 5 colors to color the graph, but only k = 4 physical registers are available, spilling is necessary.



How Difficult is Graph Coloring

Consider the following two different problems:

1. **Find the least k such that the graph is k-colorable**
   – NP-hard
   – How about using "approximation algorithm"?
2. **K-coloring: Given a constant k, decide whether the graph is k-colorable**
   – **NP-complete** (the problem we usually deal with in register allocation)
   – So, heuristics are needed

Coloring By Simplification

- We will introduce a linear-time approximation algorithm that gives good results
- The algorithm has four principal ingredients
   1. **Build**
   2. **Simplify**
   3. **Spill**
   4. **Select**

Ingredient I: Build (Interference Graphs)

- Use liveness analysis to construct the interference graphs:
  - Each node represents a temporary value
  - An edge (t1, t2) indicates a pair of temporaries that cannot be assigned to the same register.
  - Analyze for all program points
- The *most common* reason for an interference edge is that t1 and t2 are live at the same time

3. Coloring By Simplifications

- Coloring by Simplifications
  - Simplification & Select
  - Spillling
- Coalescing
- Precolored Nodes

Ingredient II: Simplify

Color the graph using a simple heuristic:

- Suppose the graph G contains a node m with fewer than K neighbors (K: the number of machine registers)
- Let G' be the graph G – {m} obtained by removing m
- If G' can be colored, then so can G **(Why?)**

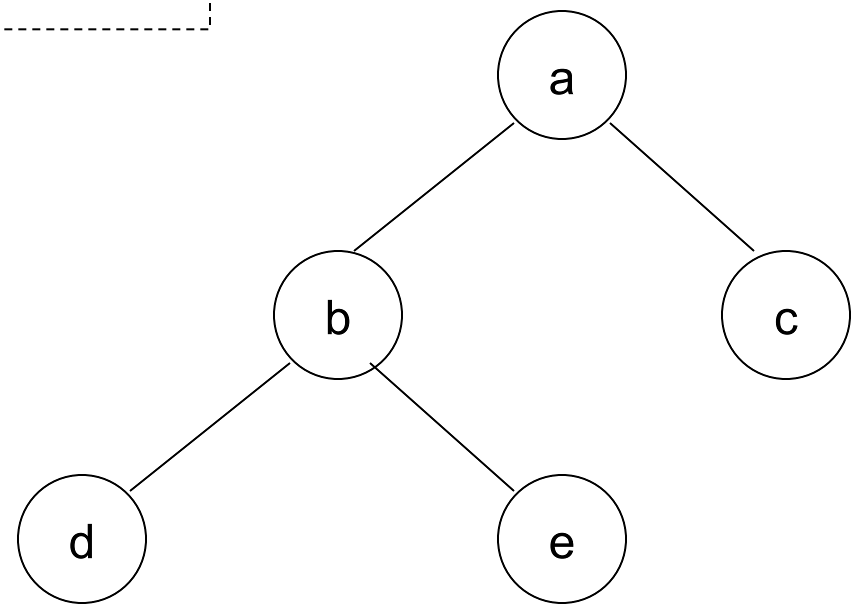
Ingredient II: Simplify

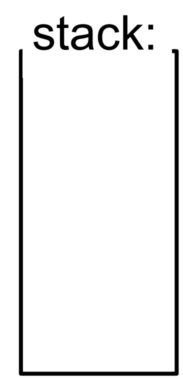
Color the graph using a simple heuristic:

- Suppose the graph G contains a node m with fewer than K neighbors (K: the number of machine registers)
- Let G' be the graph G – {m} obtained by removing m
- If G' can be colored, then so can G **(Why?)**
- This lead naturally to a ***stack-based*** algorithm for coloring
  - Repeatedly remove (and push on a stack) nodes of degree less than K.
  - Each such simplification will decrease the degrees of other nodes, leading to more opportunity for simplification

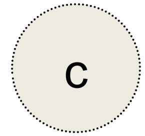
Example: Simplification (Assume K = 2)

- A vertex such that its degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty

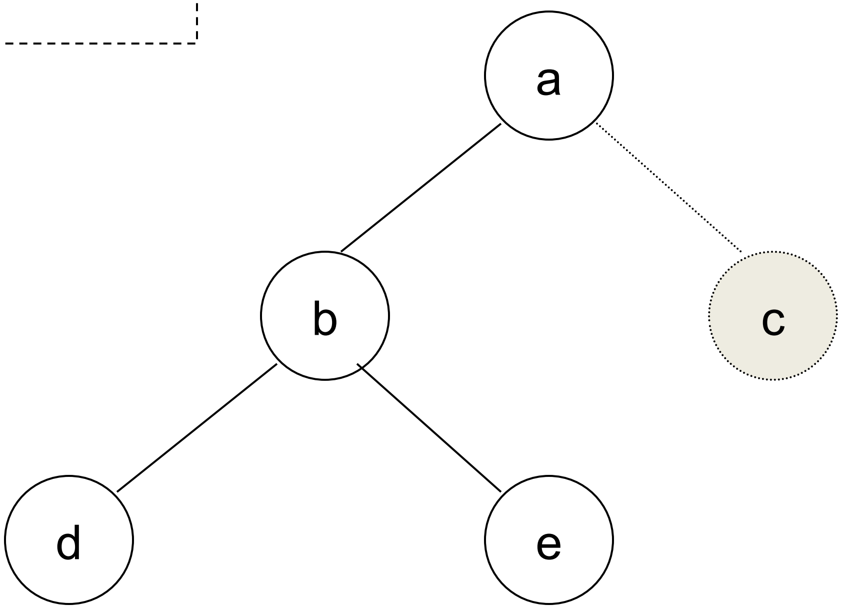


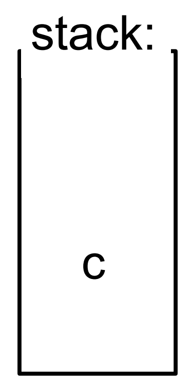


Example: Simplification (Assume K = 2)

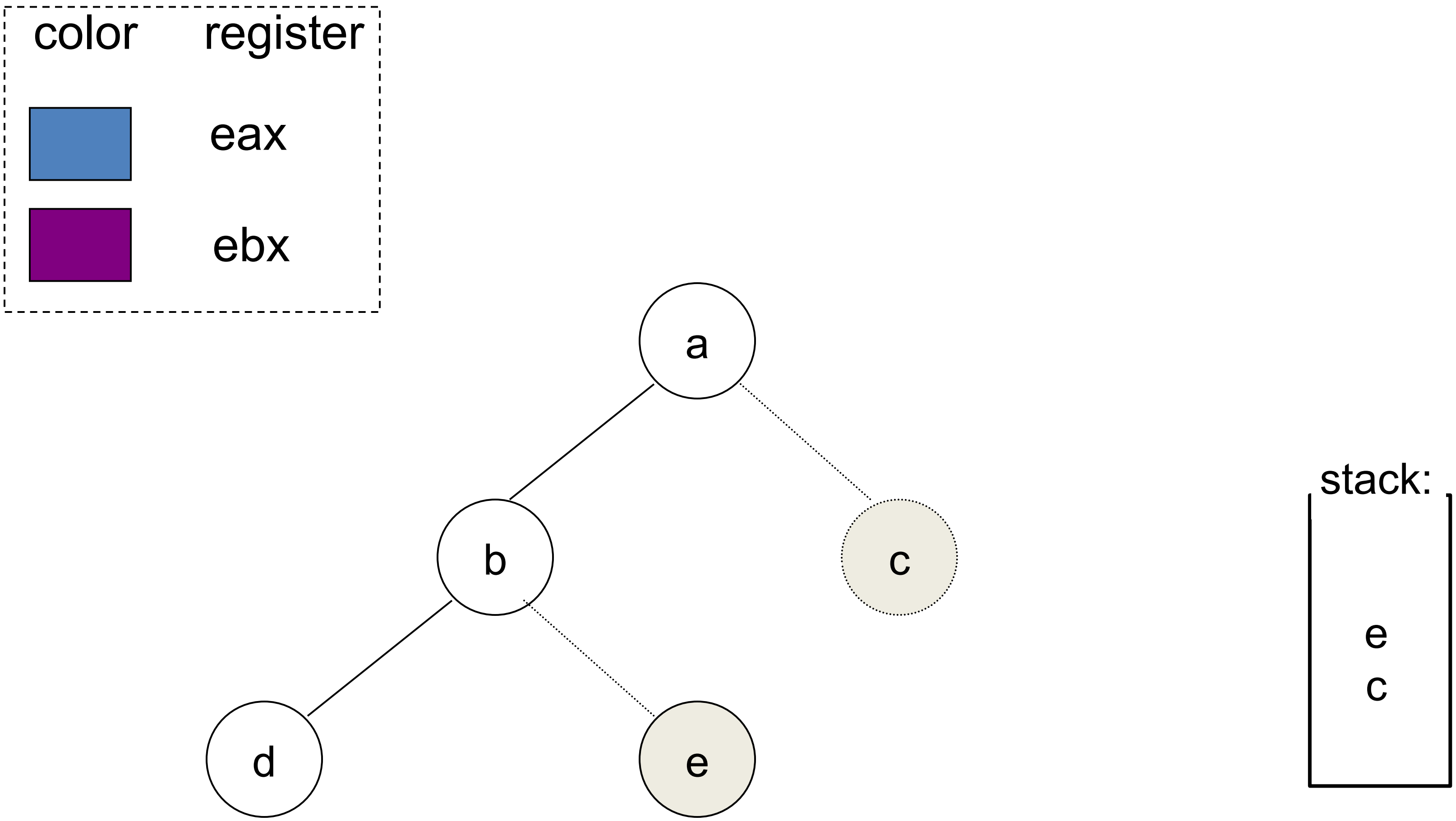


- The vertice c is removed
- The edge is removed

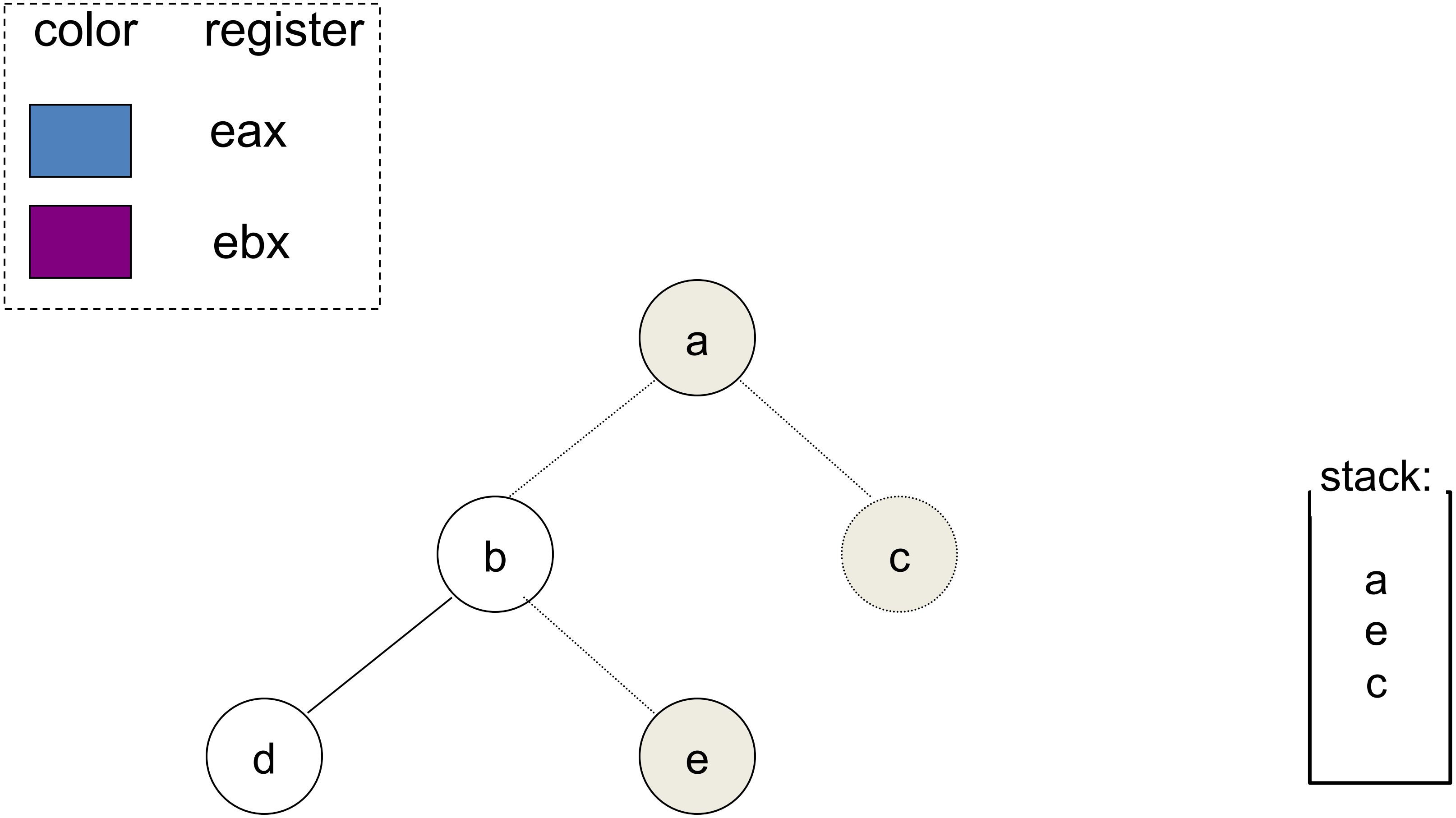




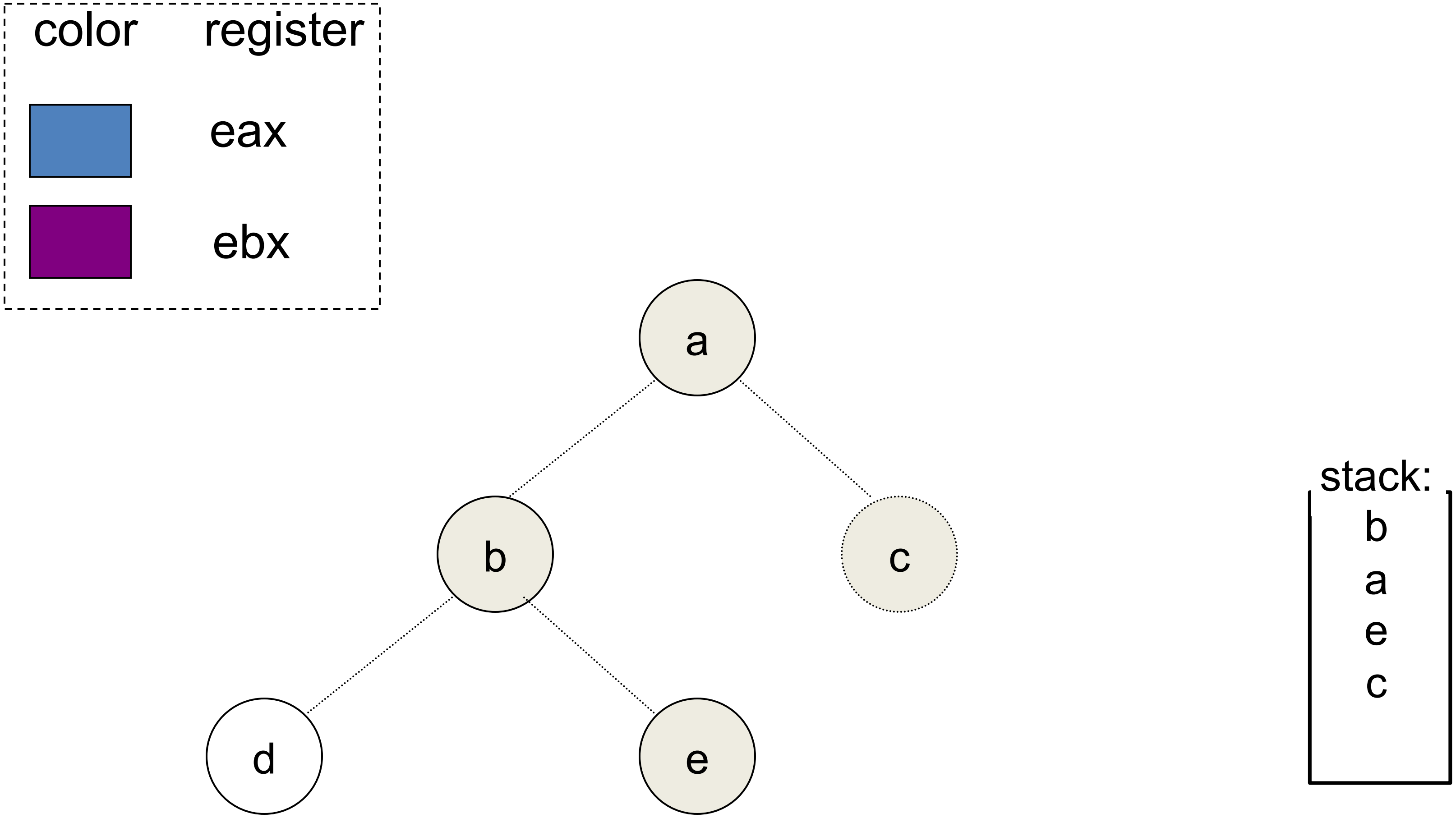
Example: Simplification (Assume K = 2)



Example: Simplification (Assume K = 2)

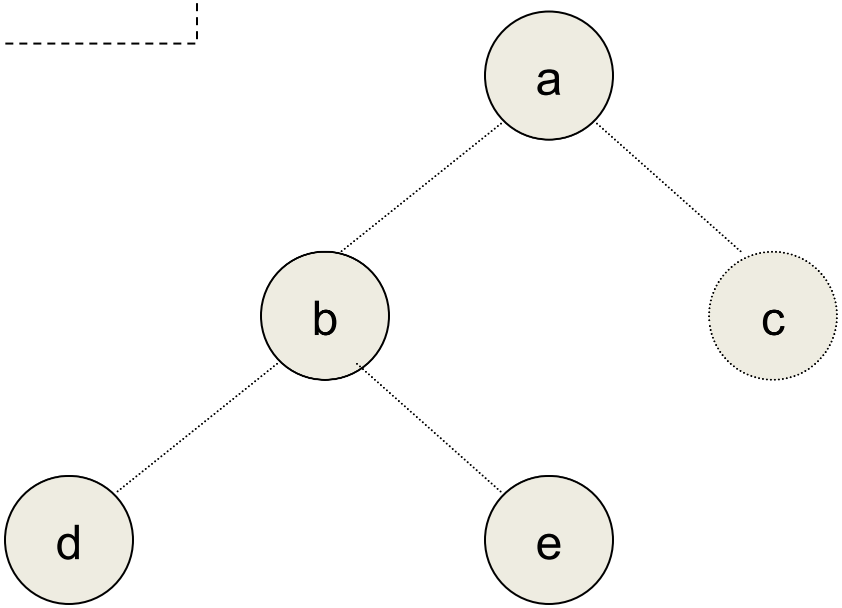


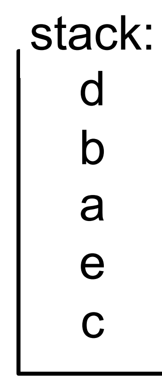
Example: Simplification (Assume K = 2)



Example: Simplification (Assume K = 2)

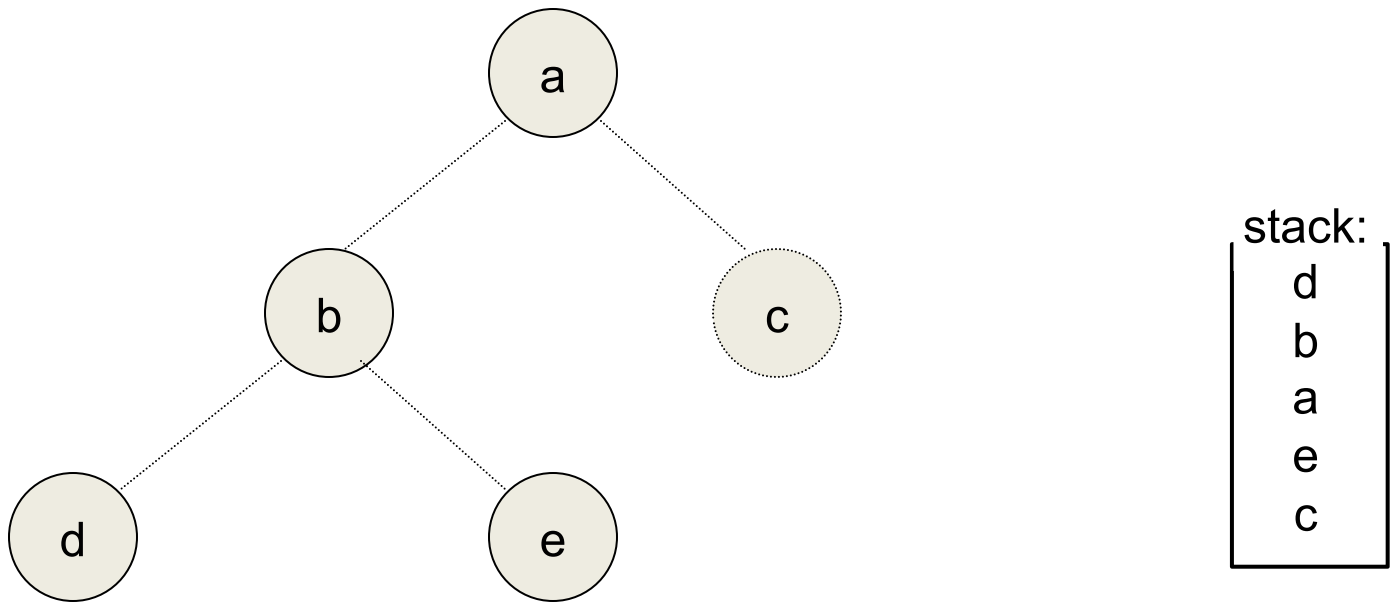
- A vertex such that its degree < k is always k-colorable
- Remove such vertices and push them to a stack until the graph becomes empty!





Ingredient IV: Select

- **Suppose that the simplification works**
  - At each step, we can choose a node to remove
  - After a few steps, the graph becomes empty!



Ingredient IV: Select

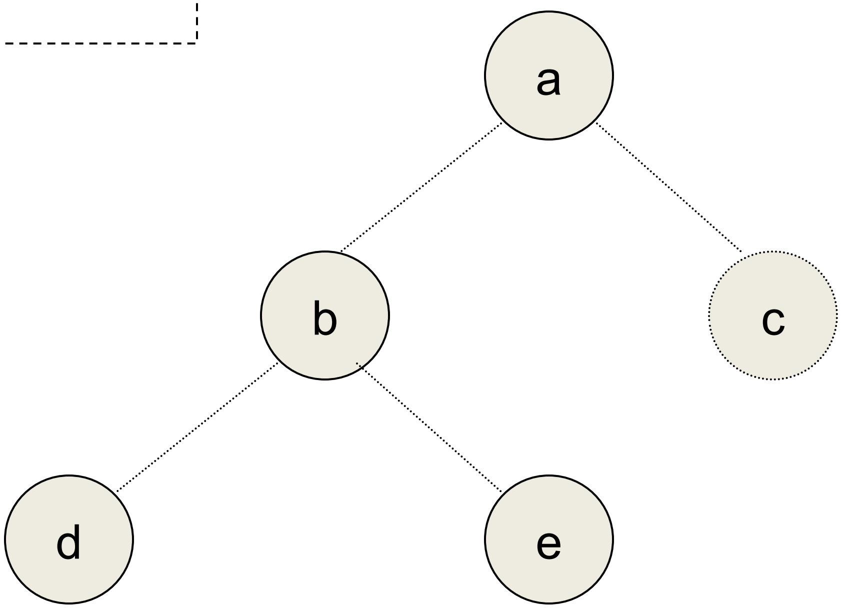
- **Suppose that the simplification works**
  - At each step, we can choose a node to remove
  - After a few steps, the graph becomes empty!

- **We can start assigning colors to nodes in the graph**
  - Starting with the empty graph, rebuild the original graph by repeatedly adding a node from the top of the stack.
  - When adding a node, there must be a color for it.

Example: Select (Assume K = 2)

- Rebuild and color the graph!

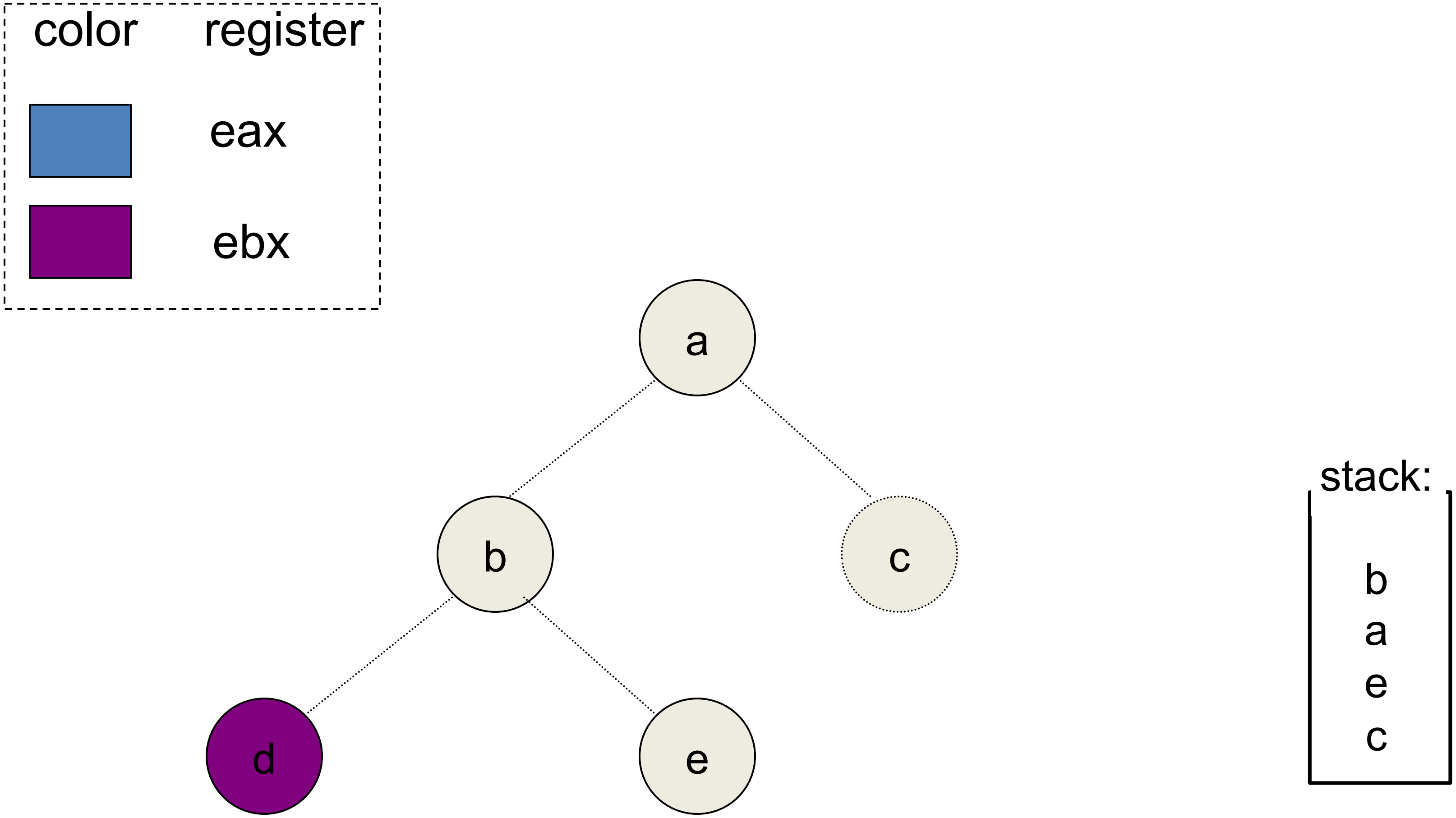
color	register
blue	eax
purple	ebx



stack:
d
b
a
e
c

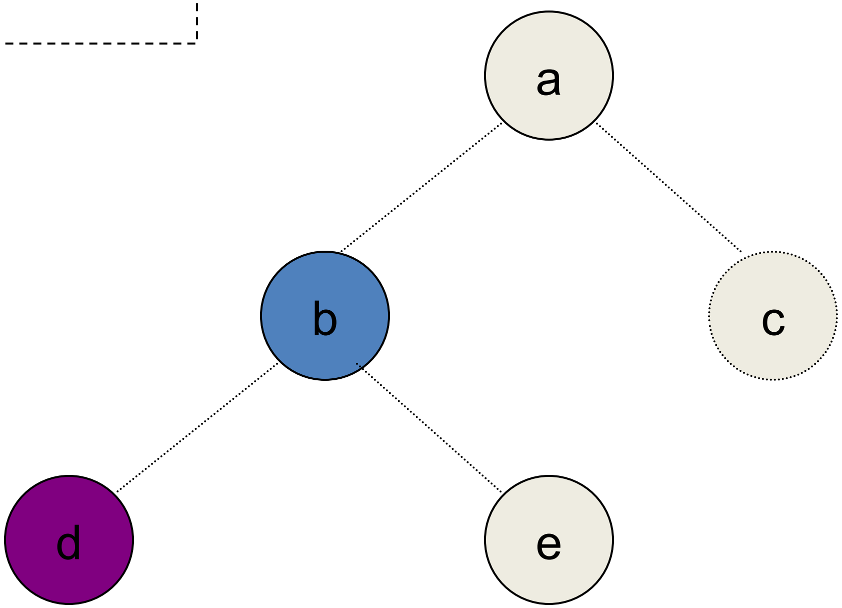
Example: Select (Assume K = 2)

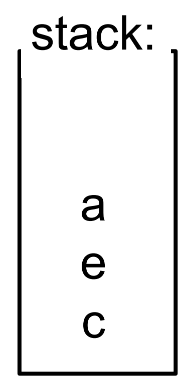
- Rebuild and color the graph!



Example: Select (Assume K = 2)

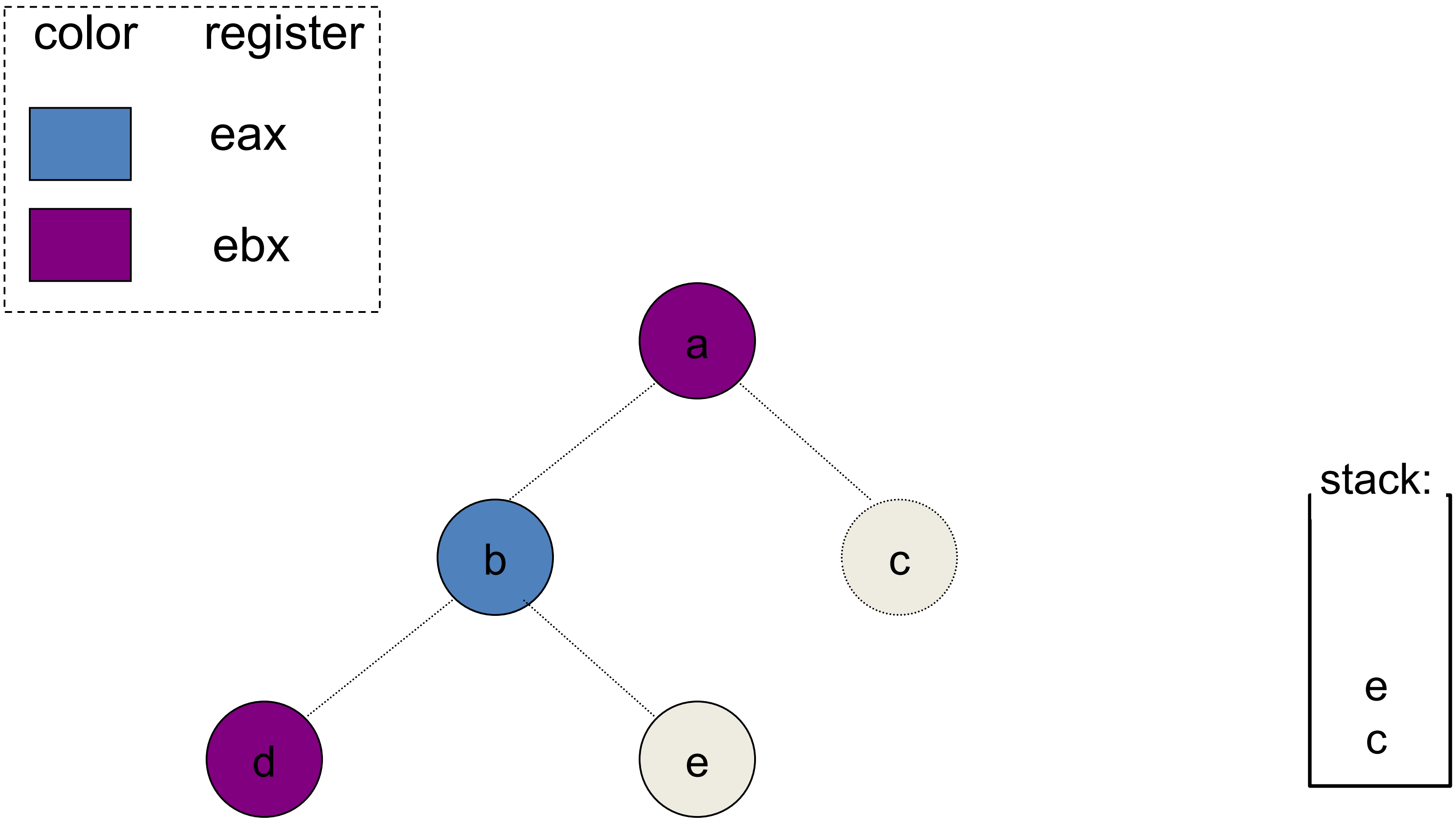
- Rebuild and color the graph!





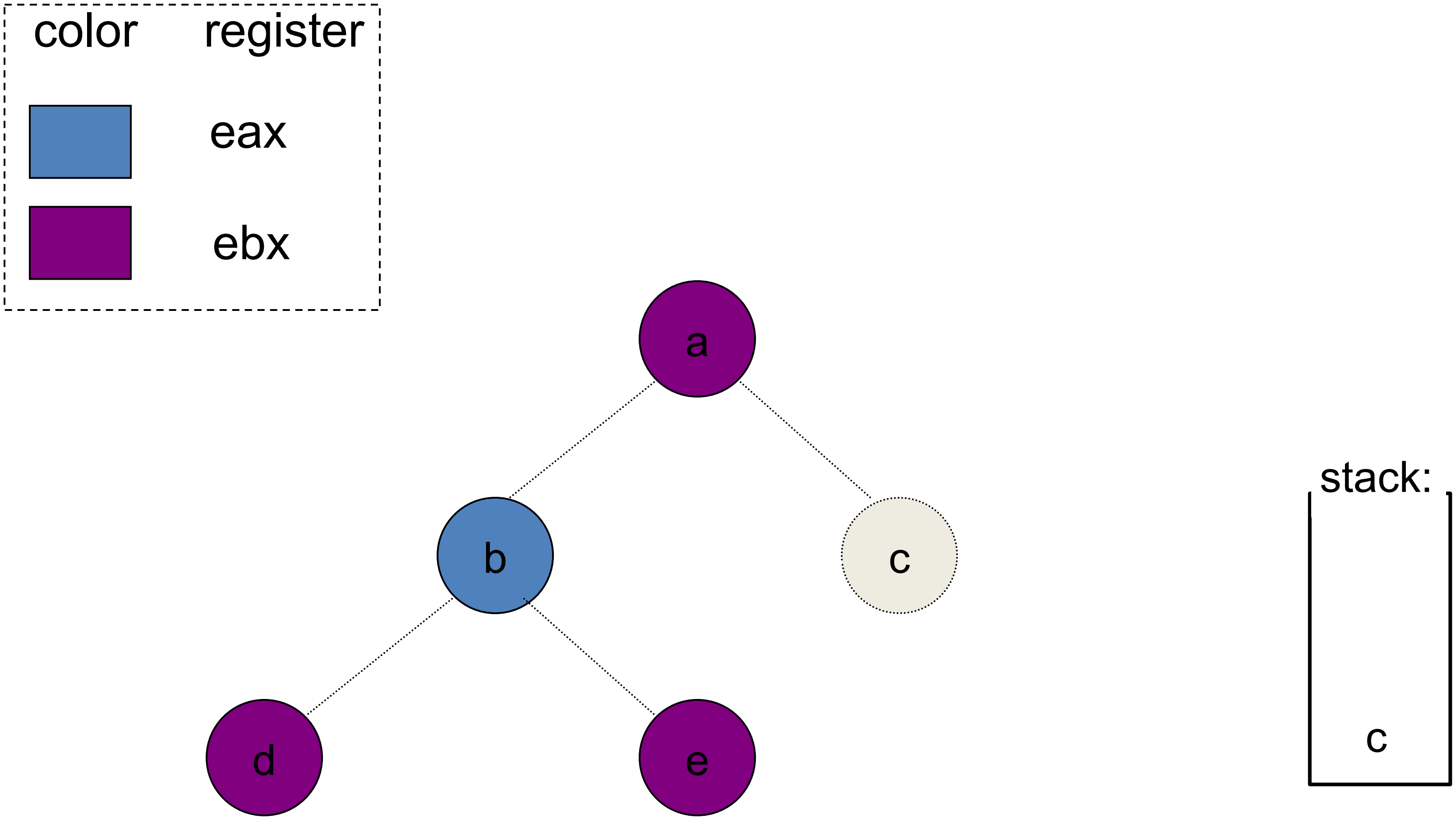
Example: Select (Assume K = 2)

- Rebuild and color the graph!



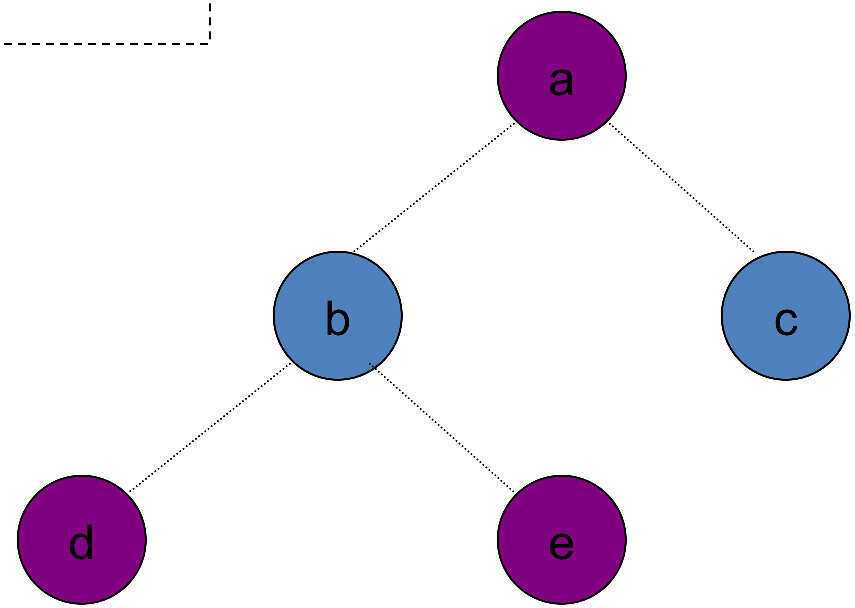
Example: Select (Assume K = 2)

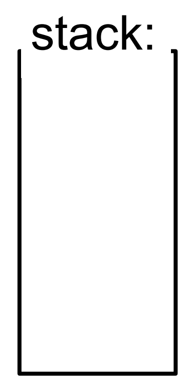
- Rebuild and color the graph!



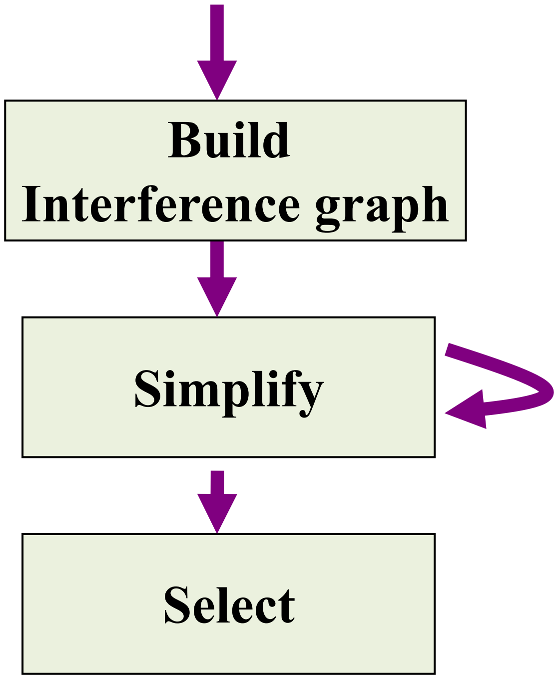
Example: Select (Assume K = 2)

- Rebuild and color the graph!





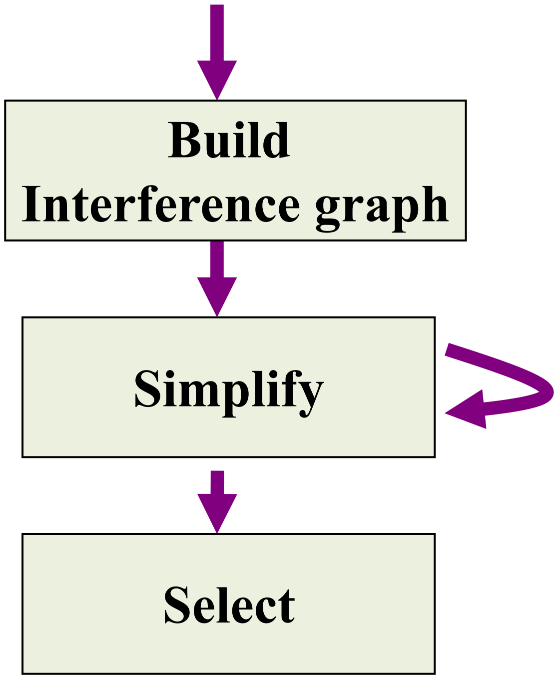
Summary: Simplification → Select




```
while graph G has node N  with degree less than k
        Remove N  and its edges from G and push N
        on a stack S
end while

if all nodes removed then graph is k-colorable
    while stack S contains node N
        Add N  to graph G and assign it a color
    end while
```

build	the conflict graph from the program
simplify	the nodes with insignificant degree
select (or color)	while rebuilding the graph.

Summary: Simplification → Select



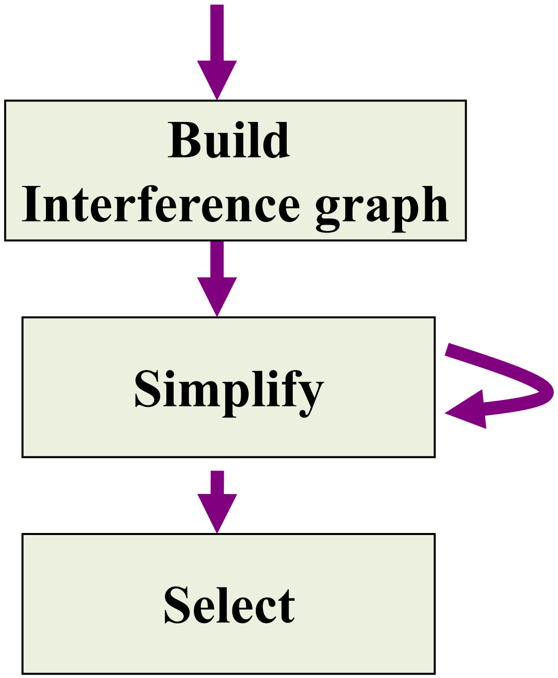
while graph G has node **N** with degree less than **k**
    Remove **N** and its edges from G and push **N** on a stack S
end while

if all nodes removed then graph is **k**-colorable
  while stack S contains node **N**
    Add **N** to graph G and assign it a color
  end while

build	the conflict graph from the program
simplify	the nodes with insignificant degree
select (or color)	while rebuilding the graph.

What if the algorithm fails?

Summary: Simplification → Select




```
while graph G has node N  with degree less than k
      Remove N  and its edges from G and push N
      on a stack S
end while

if all nodes removed then graph is k-colorable
   while stack S contains node N
         Add N  to graph G and assign it a color
   end while
```

build	the conflict graph from the program
simplify	the nodes with insignificant degree
select (or color)	while rebuilding the graph.

- The algorithms is just a fast (linear time) heuristic.
- When failed, it does not mean the graph is not k-colorable !

3. Coloring By Simplifications

- Coloring by Simplifications
  - Simplification & Select
  - Spillling
- Coalescing
- Precolored Nodes

Ingredient III: Spilling

- At some point during simplification, the graph G has nodes only of significant degree (that is, nodes of degree $\geq$ K).
- The previous algorithm does not work!!

```
while graph G has node N with degree less than k
    Remove N and its edges from G and push N on a stack S
end while

if all nodes removed then graph is k-colorable
  while stack S contains node N
      Add N to graph G and assign it a color from k colors
  end while
```

Ingredient III: Spilling

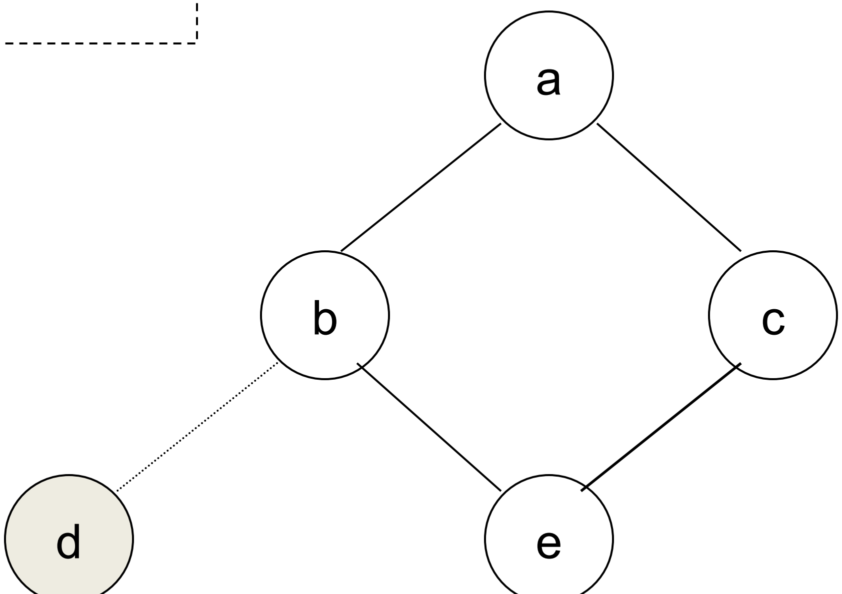
- At some point during simplification, the graph G has nodes only of significant degree (that is, nodes of degree $\geq$ K).
- The previous algorithm does not work!!

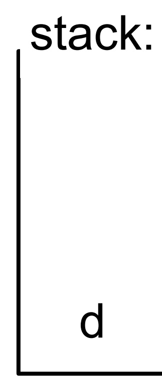
```
while graph G has node N with degree less than k
    Remove N and its edges from G and push N on a stack S
end while

if all nodes removed then graph is k-colorable
  while stack S contains node N
     Add N to graph G and assign it a color from k colors
  end while
```

- **Spilling**: We MAY need to choose some node in the graph and decide to represent it in memory, not registers

Example: Spilling (K = 2)

- What if during simplification we get to a state where all nodes have k or more neighbors ?





all nodes have
2 neighbours!

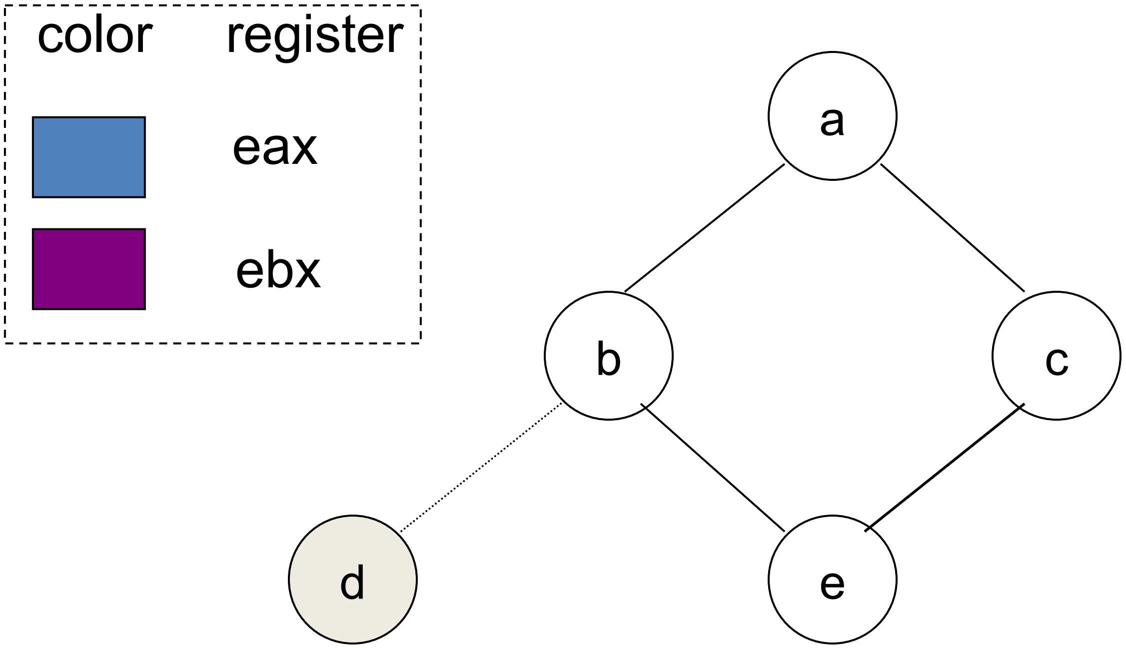
Ingredient III: Spilling

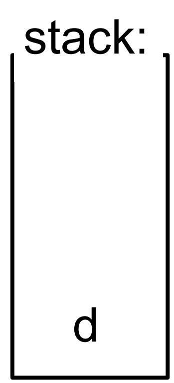
- At some point during simplification, the graph G has nodes only of significant degree (that is, nodes of degree $\geq$ K).
- We MAY need to choose some node in the graph and decide to represent it in memory, not registers

- **Optimistic Coloring** [Chaitin-Briggs Algorithm]
  - An **optimistic approximation** to the effect of spilling: the spilled node does not interfere with any of the other nodes remaining in the graph
  - It can therefore be removed and pushed on the stack, and **the simplify process continued**

Example: Optimistic Coloring (K=2)

- **Pick a node as a candidate for <u>spilling</u>**
  - Remove it from the graph and put it into the stack
  - For example, we choose b and continue the simplification

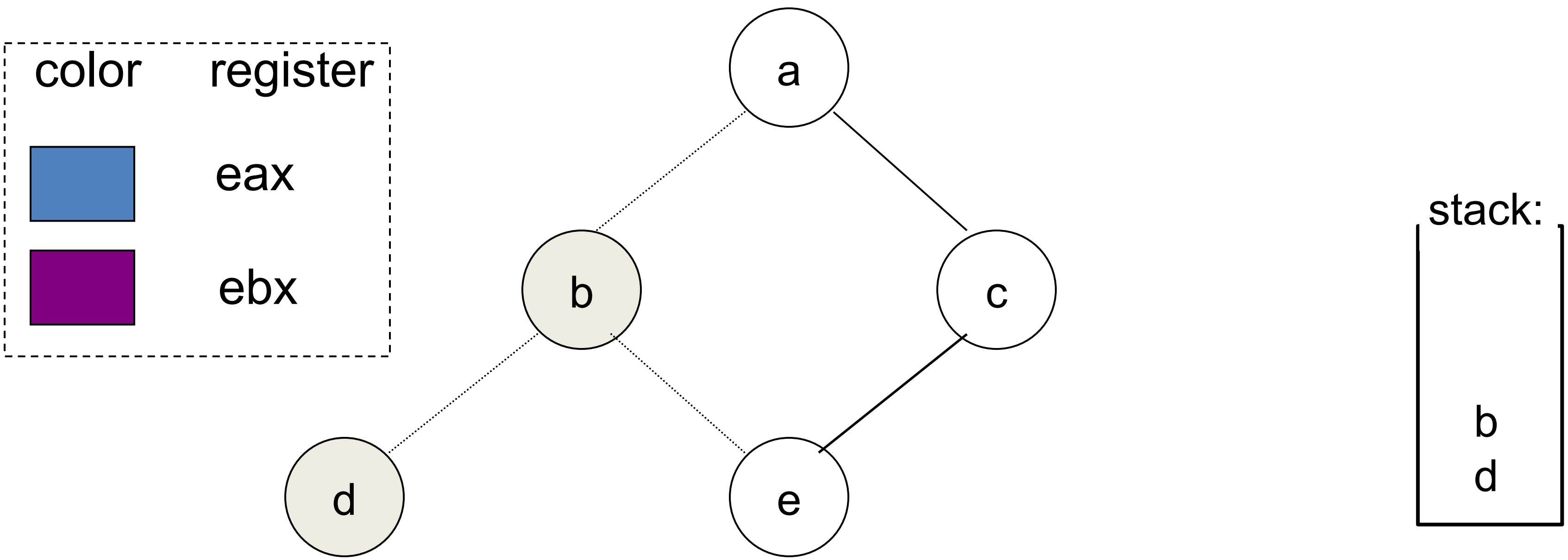




- 做乐观假设，”照常”把节点删除、放栈上
- 不”打断”simplification策略的运行

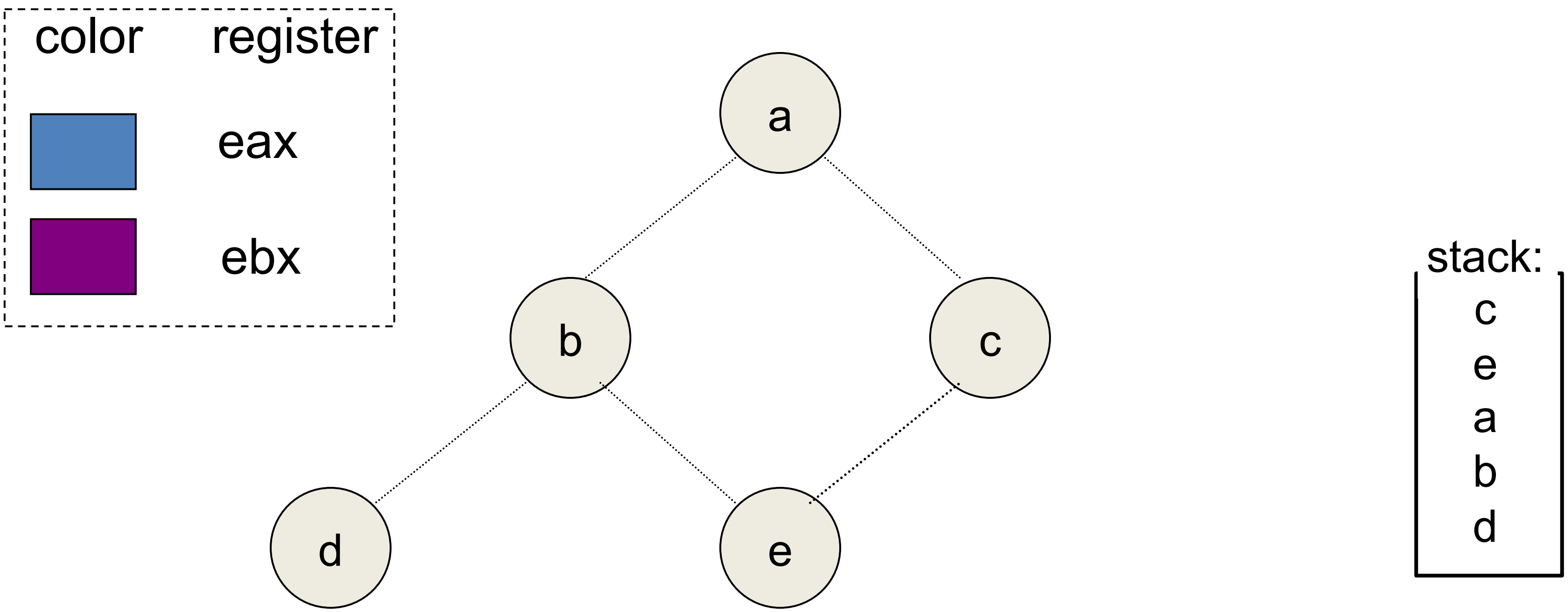
Example: Optimistic Coloring (K=2)

- Pick a node as a candidate for <u>spilling</u>
  - After choosing b, the stack looks as follows



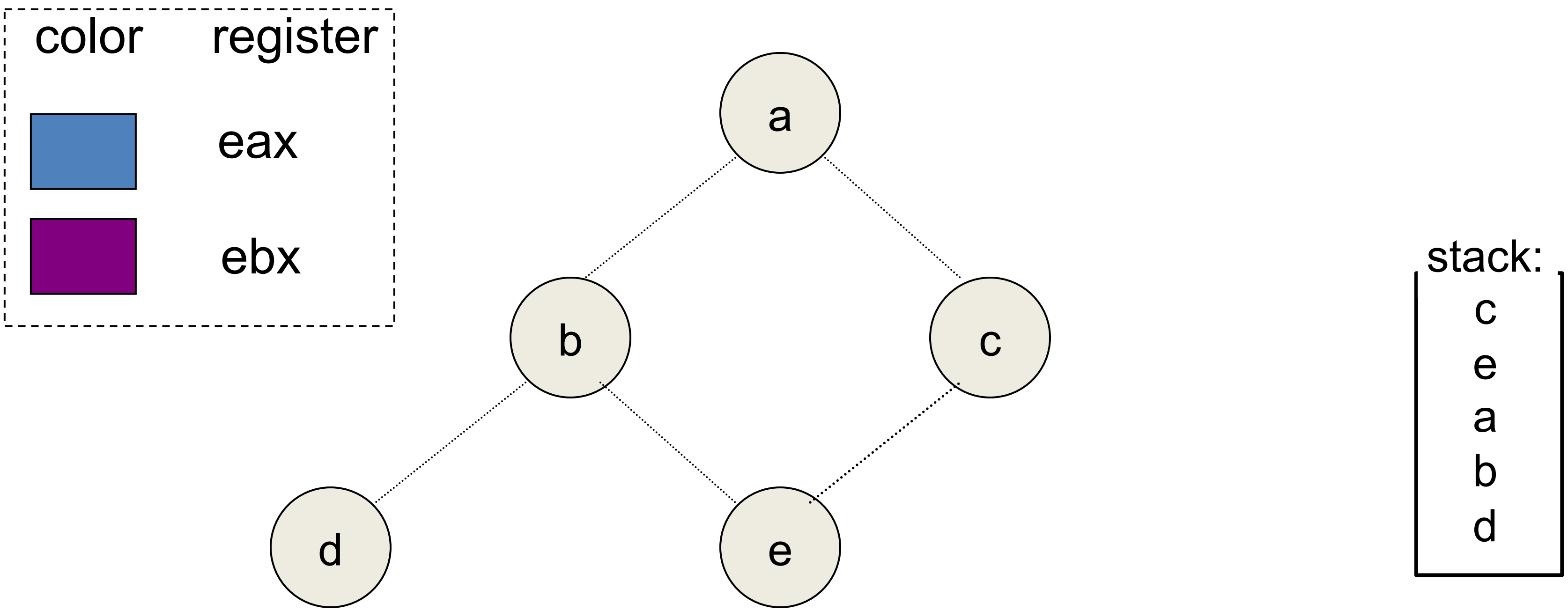
Example: Optimistic Coloring (K=2)

- Continue the simplification
- After a few steps, the simplification succeeds: a, e, c!



Example: Optimistic Coloring (K=2)

- Suppose that the graph becomes empty, and we need to start the "select" (coloring)



Can we complete the coloring in the select phase as before?

Ingredient IV: Select (Revised)

- Suppose that the graph becomes empty, and we start the "select" (coloring)
- **Problem**: When potential spill node n that was pushed using the Spill heuristic is popped, there is no guarantee that it will be colorable.

Ingredient IV: Select (Revised)

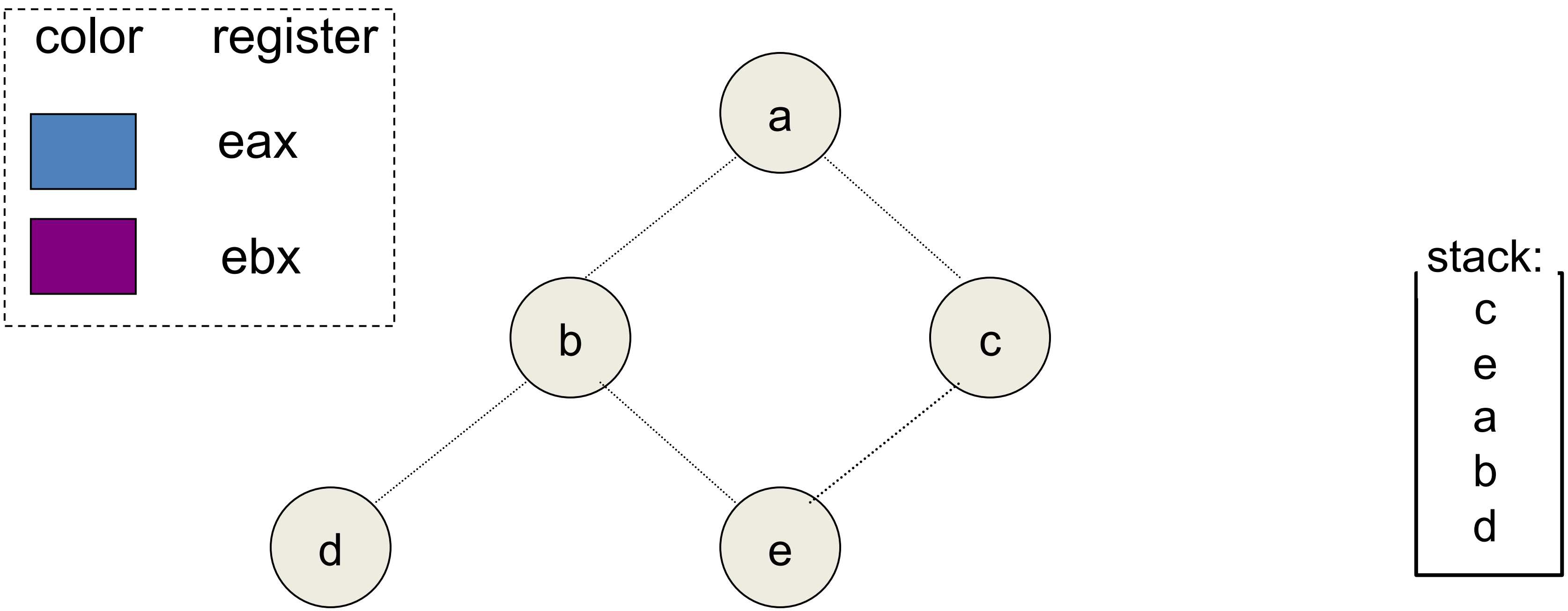
- Suppose that the graph becomes empty, and we start the "select" (coloring)
- **Problem**: When potential spill node n that was pushed using the Spill heuristic is popped, there is no guarantee that it will be colorable.

1. **If n's neighbors are colored with fewer than K colors**
   - We can color n and n does not become an actual spill.
   - The *optimistic coloring works!*

It is possible that the graph is still K-colorable*!!*

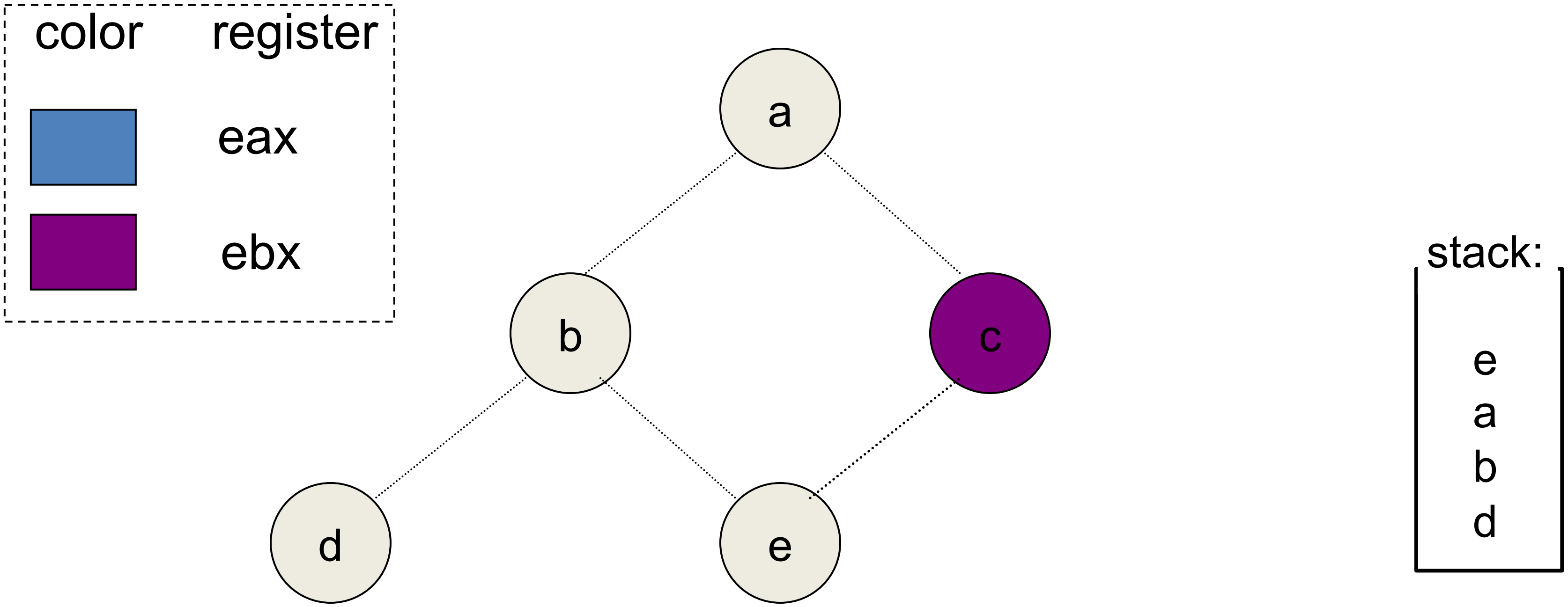
Example: Select after Optimistic Coloring

- Continue the simplification
- After a few steps, the simplification succeeds: a, e, c!



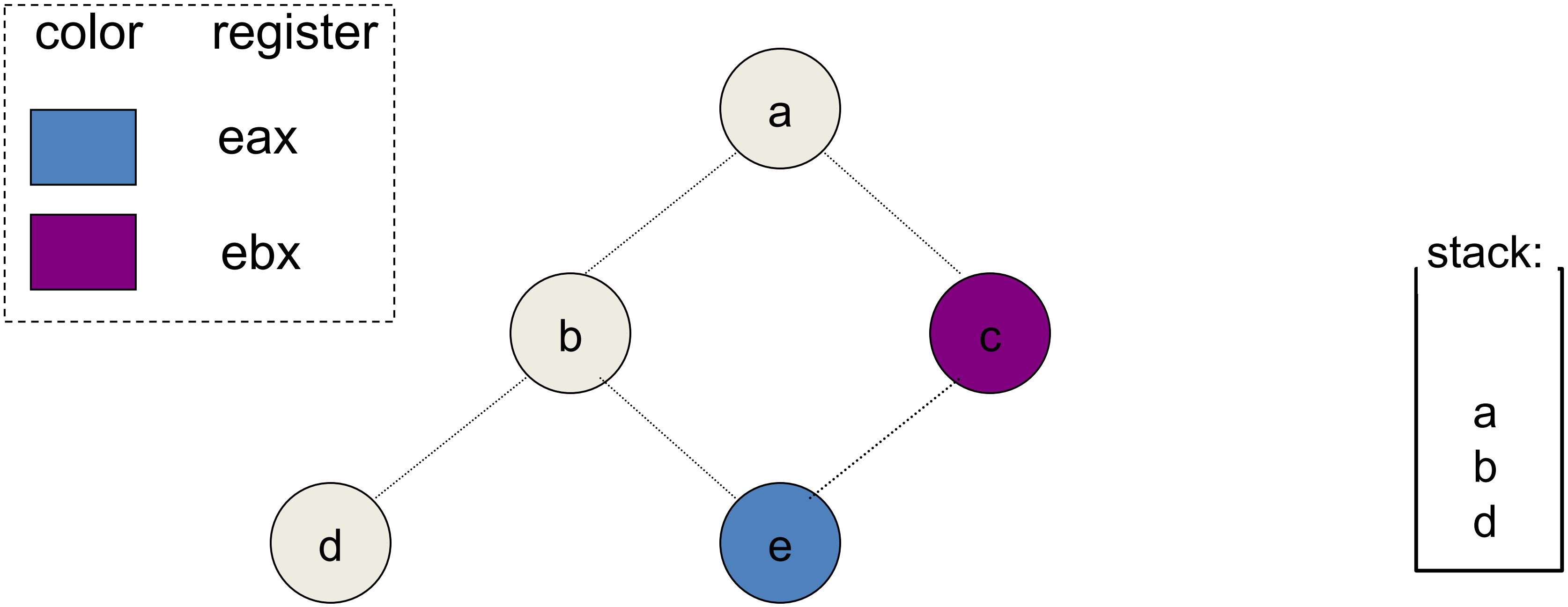
Example: Select after Optimistic Coloring

- Rebuild and color the graph! (following the previous “select”)



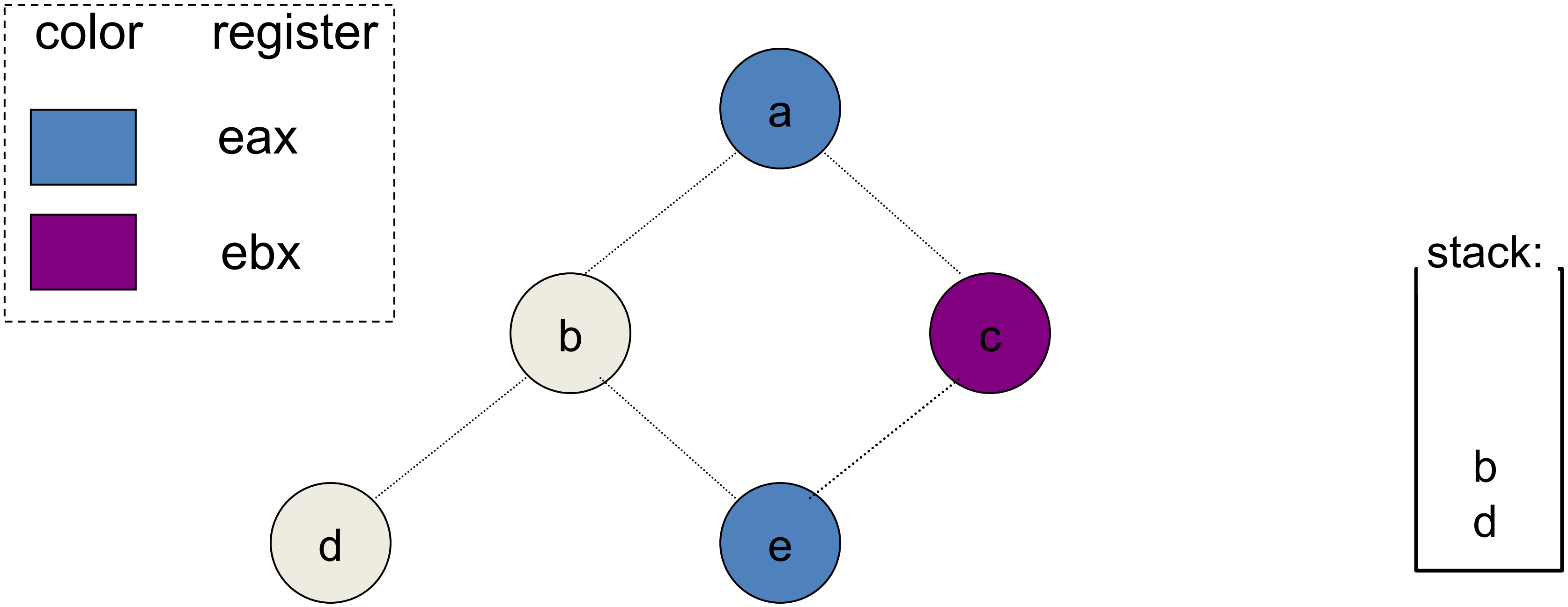
Example: Select after Optimistic Coloring

- Rebuild and color the graph! (following the previous “select”)



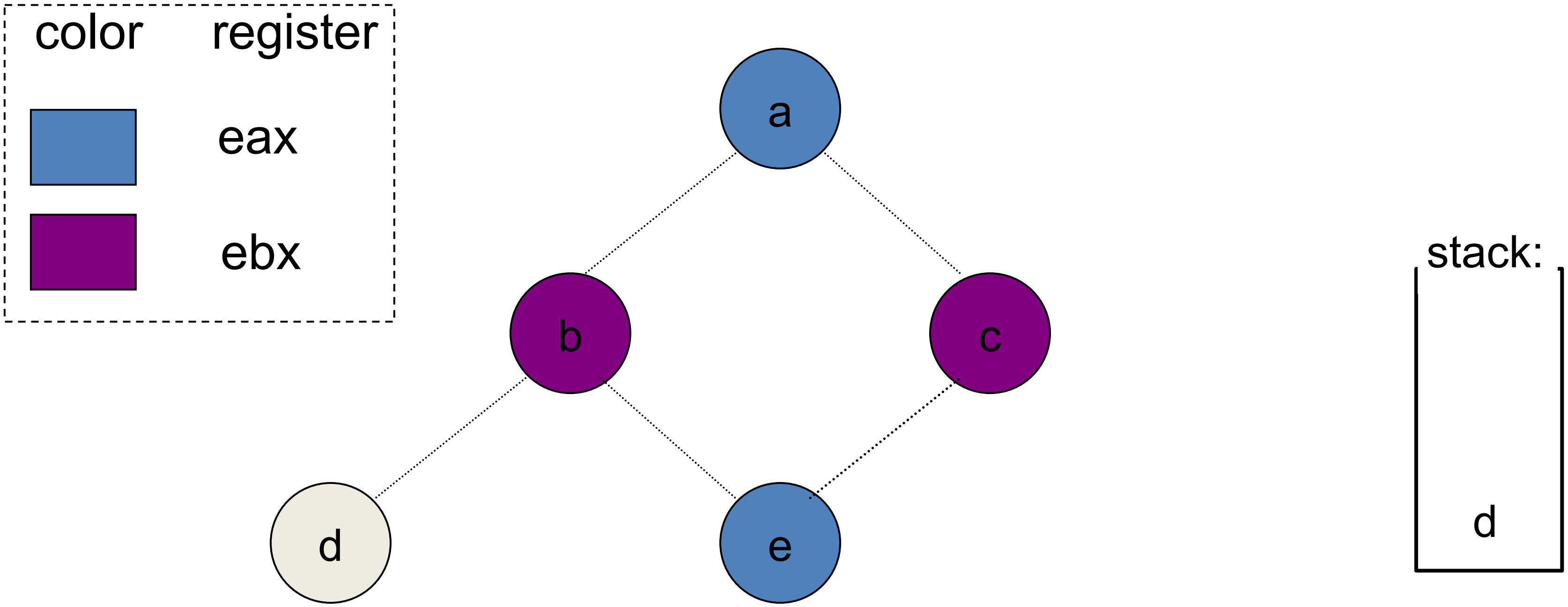
Example: Select after Optimistic Coloring

- Rebuild and color the graph! (following the previous “select”)



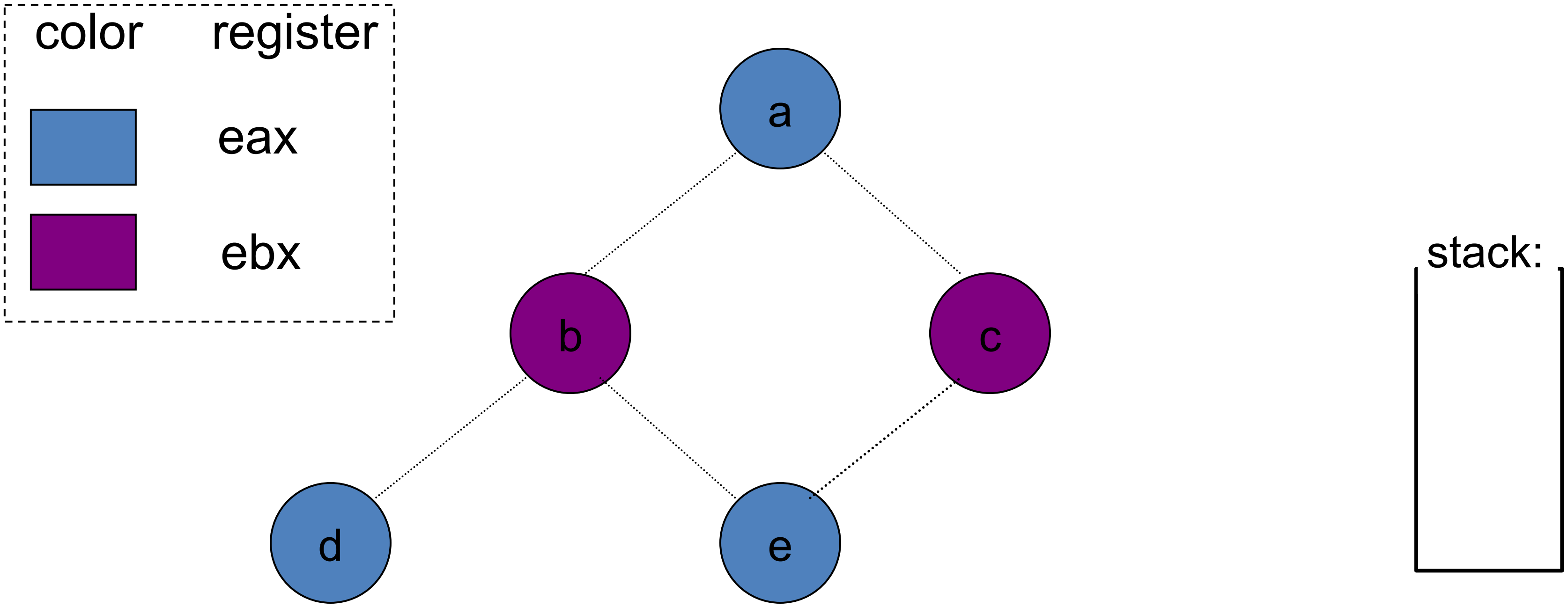
Example: Select after Optimistic Coloring

- Rebuild and color the graph! (following the previous “select”)



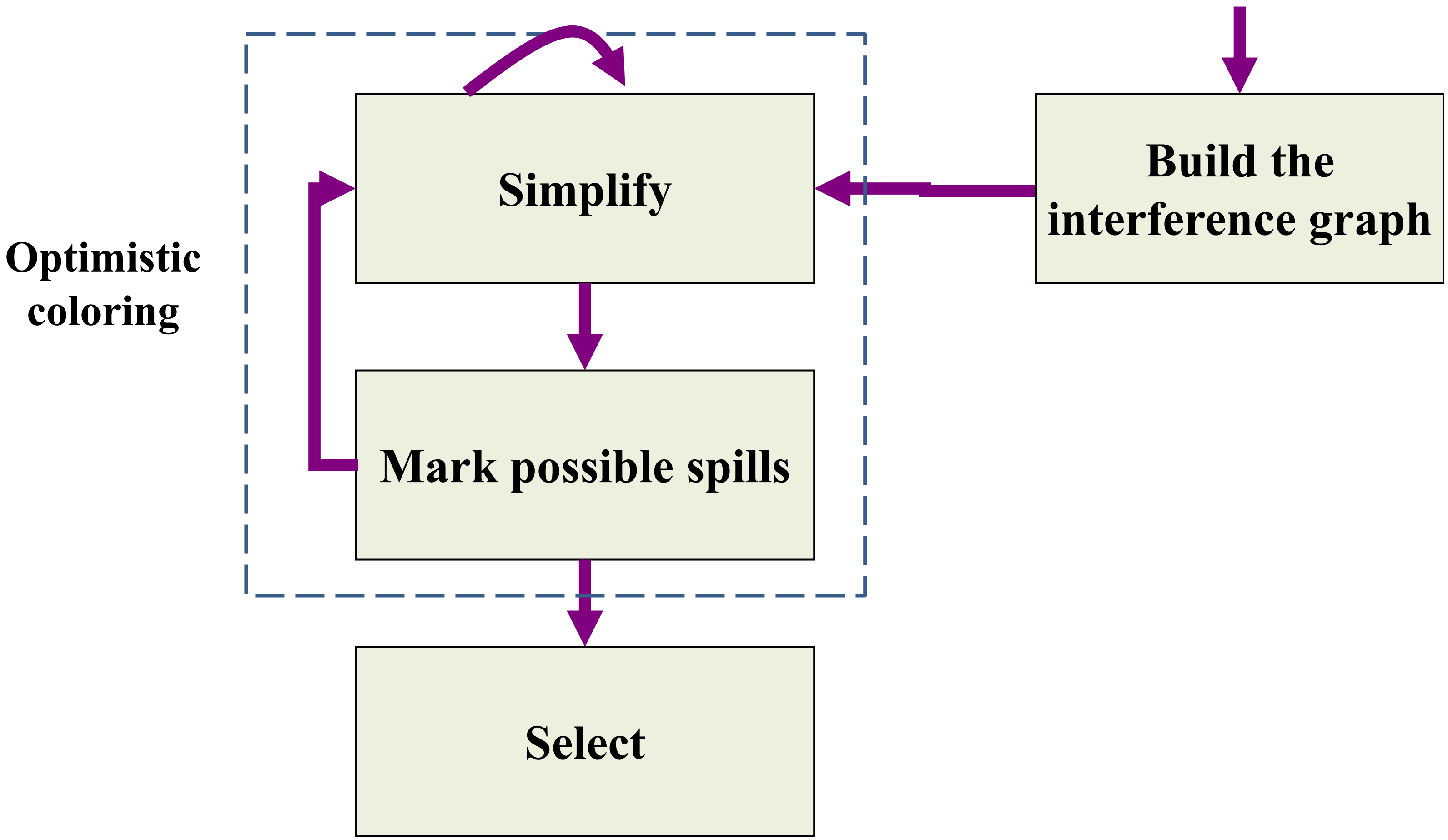
Example: Select after Optimistic Coloring

- Rebuild and color the graph! (following the previous "select")
- We got lucky: the "simplification & select" still works!



Sometimes, it is not necessary to do the actual spill!

Summary: Simplify with Optimistic Coloring → Select



Sometimes, it is not necessary to do the actual spill!

Ingredient IV: Select (Revised)

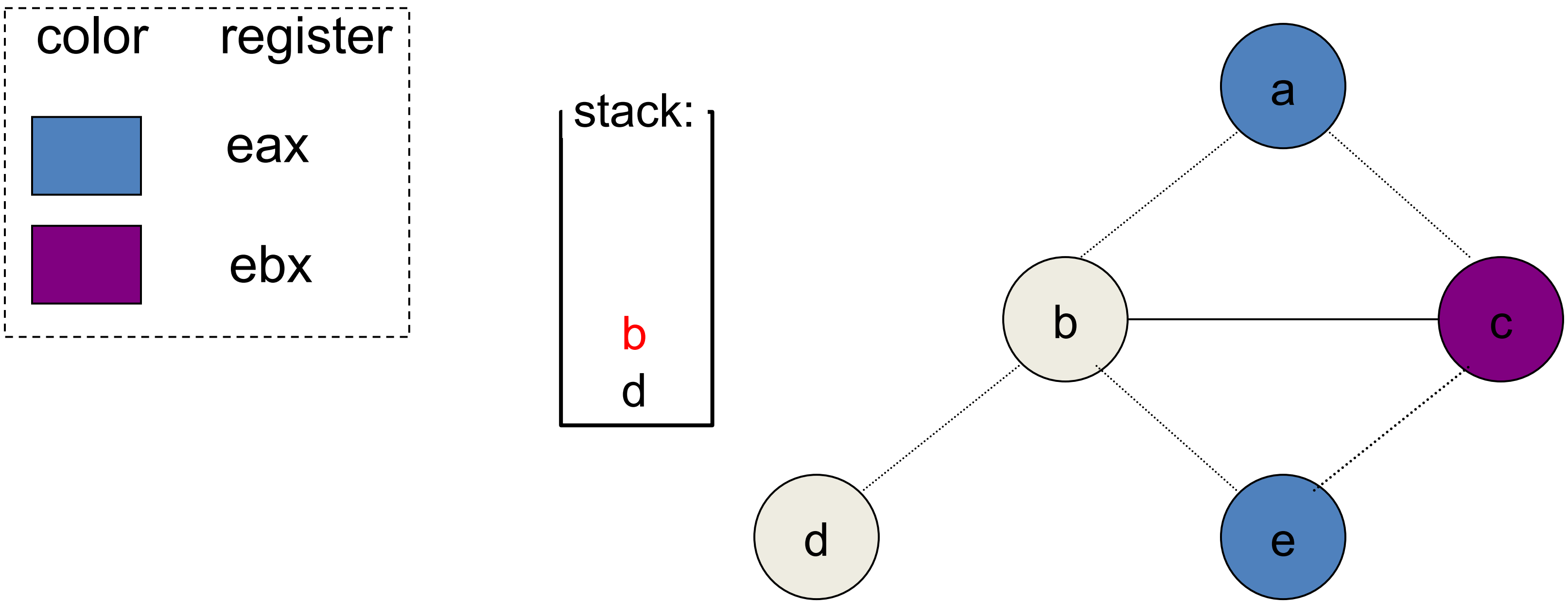
- Suppose that the graph becomes empty, and we start the "select" (coloring)
- **Problem**: When potential spill node n that was pushed using the Spill heuristic is popped, there is no guarantee that it will be colorable.

1. **If n's neighbors are colored with fewer than K colors**
   - We can color n and n does not become an actual spill.
   - The optimistic coloring works *(graph still k-colorable)*

However, the optimistic coloring heuristic can fail!

When the Optimistic Heuristic Fails

- What happens if no color can be assigned to a marked spilled node ?
  - When we have to assign a color to b whose neighbors have 2 different colors already!



Ingredient IV: Select (Revised)

- **Problem**: When potential spill node n that was pushed using the Spill heuristic is popped, there is no guarantee that it will be colorable.

1. **If n's neighbors are colored with fewer than K colors**
   - We can color n and n does not become an actual spill.
   - The *optimistic coloring works*
2. **If n's neighbors have been colored with K different colors**
   - We have to perform an **actual spill!**
   - We do not assign any color, but continue the **Select** Phase to identify other actual spill

Start Over (重新开始)

If the **Select** phase is unable to find a color for some node(s)

1. Do the actual spill: the program is rewritten to
   – fetch them from memory just before each use, and
   – store them back after each def.
2. The algorithm is repeated on this rewritten program
   – Recompute liveness → build interference graph → …

- This process is iterated util simplify succeeds with no spills.
- In practice, one or two iterations almost always suffice.

Example: Start Over

1. Do the Actual Spill

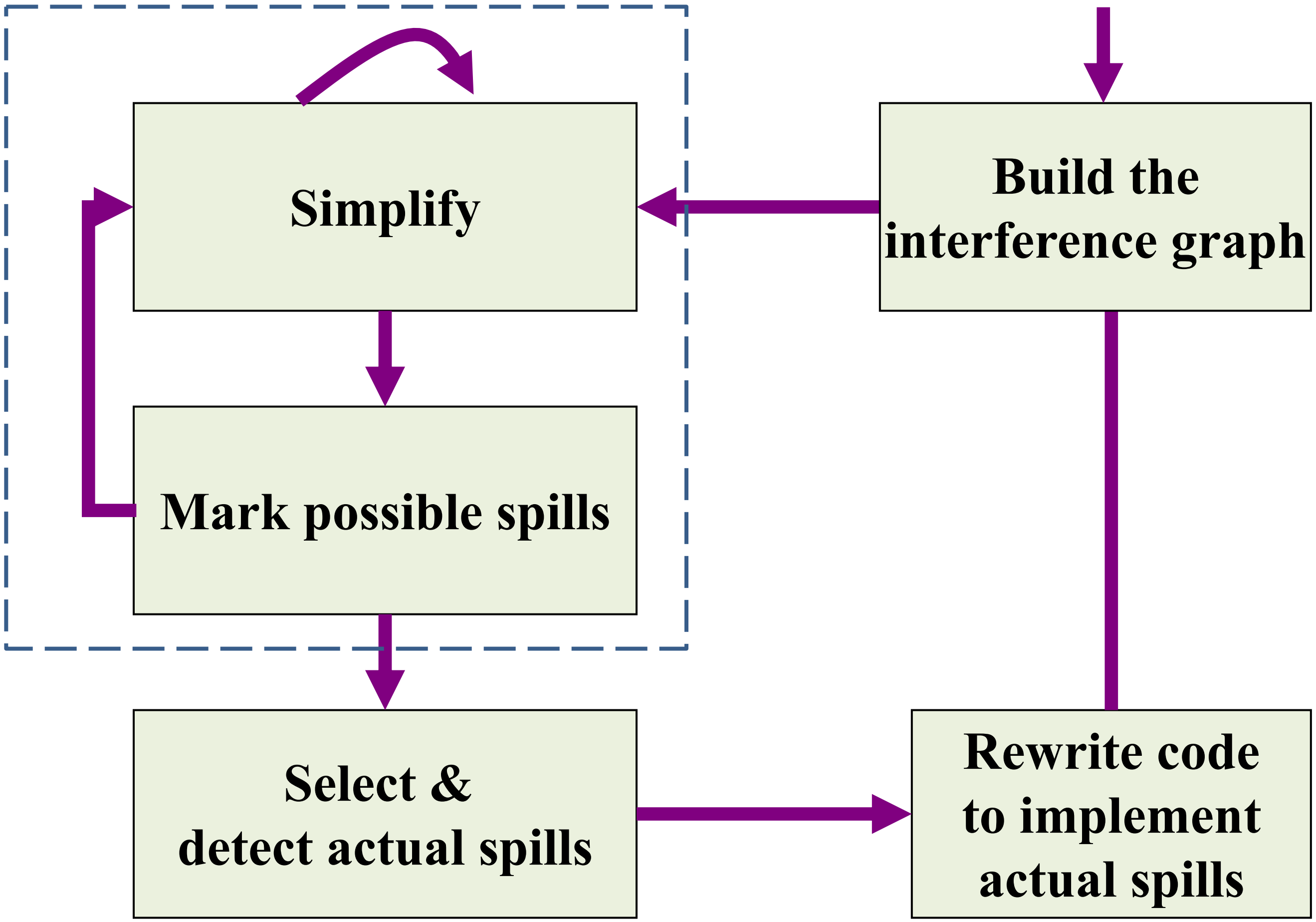
- Optimistic coloring failed = must spill variable f
- We must allocate memory location as home of f
  - Typically in current stack frame (call this address fa)
- Before each operation that uses f, insert fx := load fa
- After each operation that defines f, insert store fx, fa

A spilled temporary will turn into several new temporaries with tiny live ranges. (and make the new interference graph in the next steps “simpler”)

2. Recompute liveness information

3. Rerun the coloring algorithm

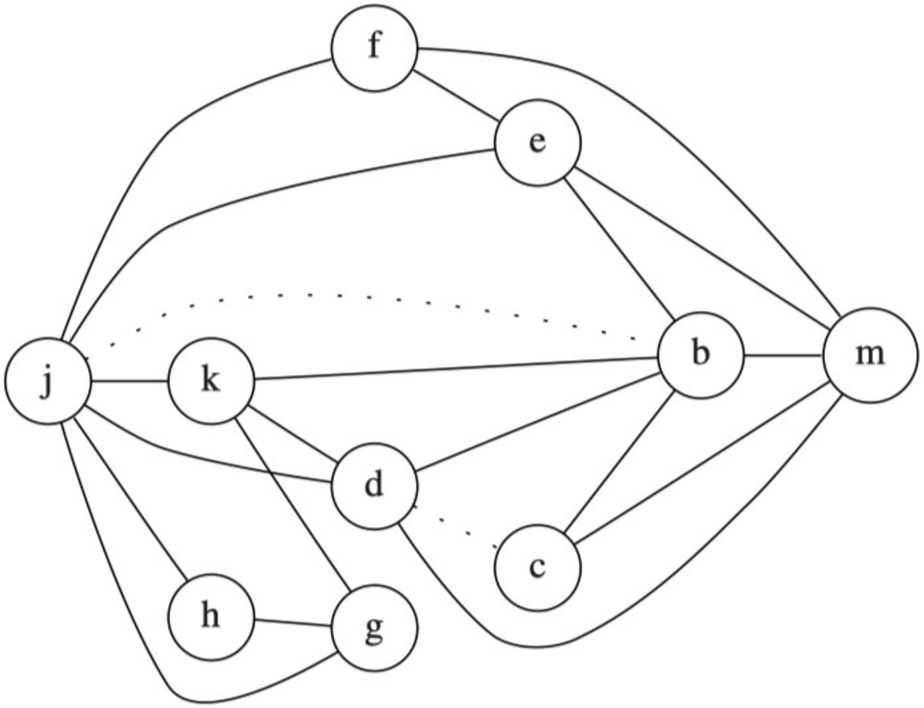
Summary: “Simplification → Select” Revised



Example (K = 4)

- The simplify phase can start with the nodes g, h, c, and f in its working set.
  - Since they have less than four neighbors each

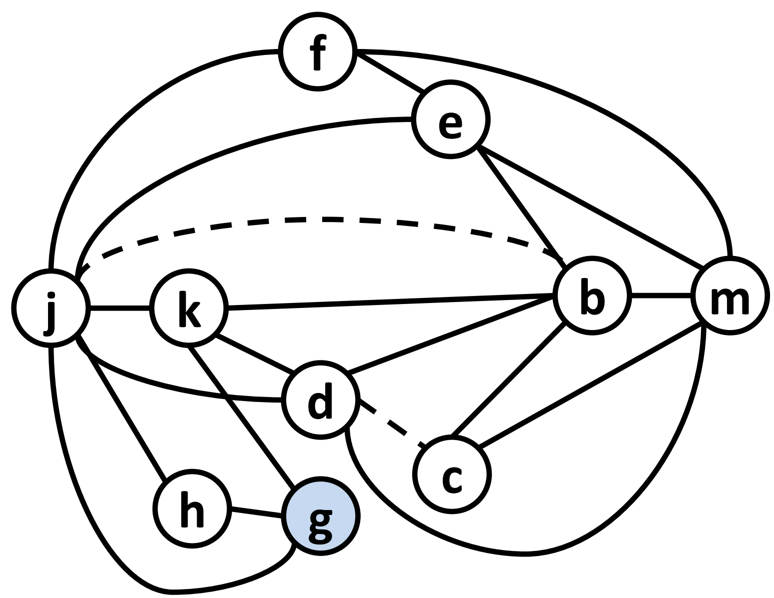
Live in: k j
g := mem[j+12]
h := k-1
f := g*h
e := mem[j+8]
m := mem[j+16]
b := mem[f]
c := e+8
d := c
k := m+4
j := b
Live out: d k j



Dotted lines indicate move instructions

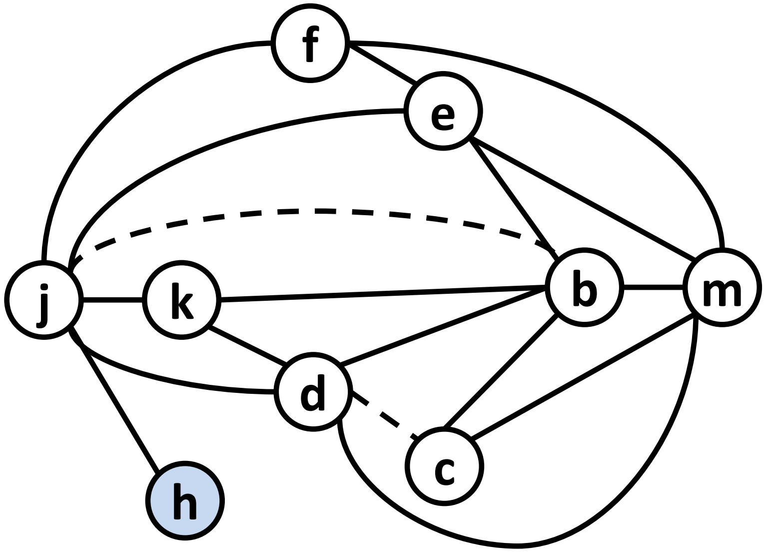
Example (K = 4)

Stack	**Assignment**



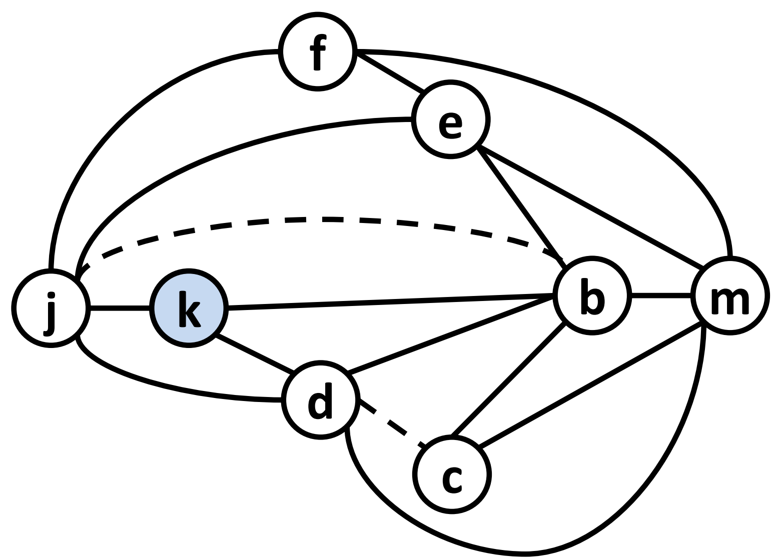
Example (K = 4)

g	
Stack	**Assignment**



Example (K = 4)

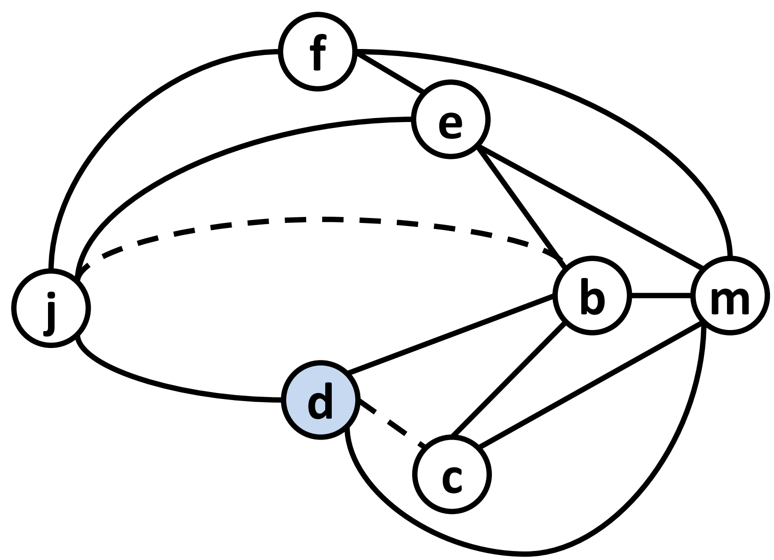
h	
g	
Stack	**Assignment**



Example (K = 4)

- Remove d

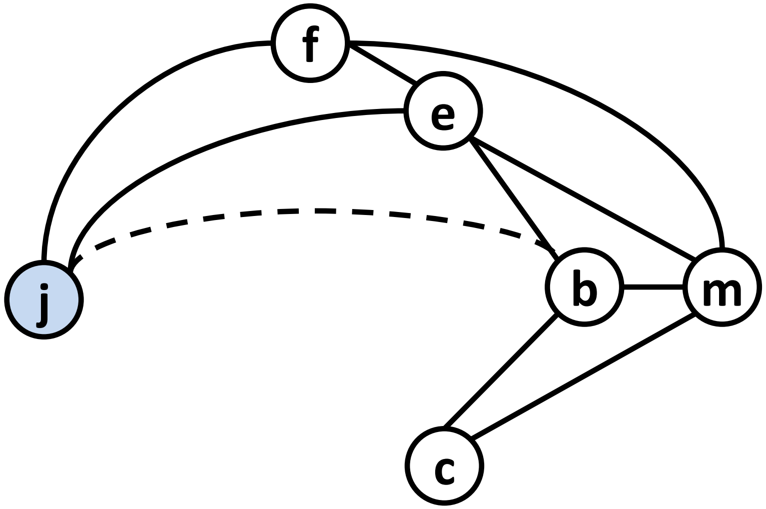
k	
h	
g	
Stack	**Assignment**



Example (K = 4)

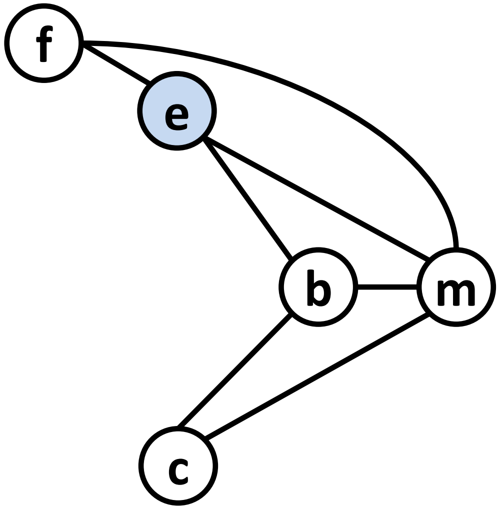
- Remove j

d	
k	
h	
g	
Stack	**Assignment**



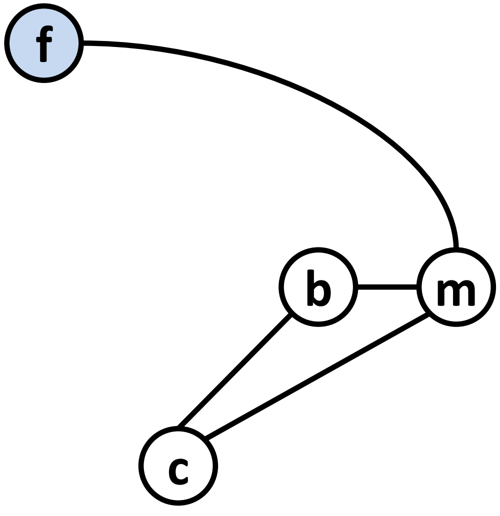
Example (K = 4)

j	
d	
k	
h	
g	
Stack	**Assignment**



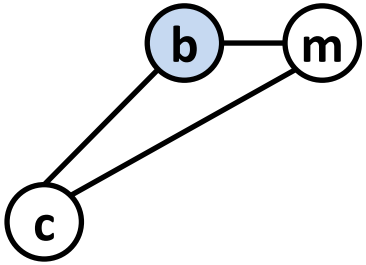
Example (K = 4)

e	
j	
d	
k	
h	
g	
Stack	**Assignment**



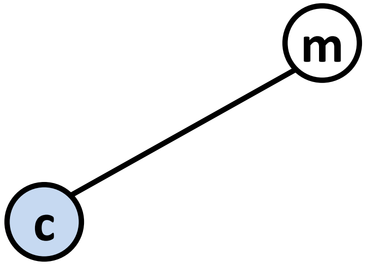
Example (K = 4)

f	
e	
j	
d	
k	
h	
g	
Stack	**Assignment**



Example (K = 4)

b	
f	
e	
j	
d	
k	
h	
g	
Stack	**Assignment**



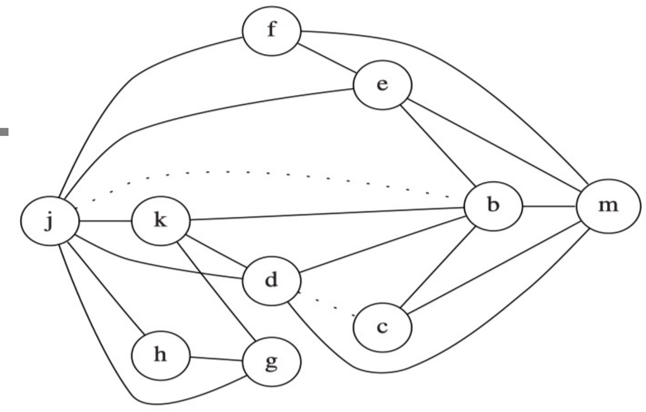
Example (K = 4)

c	
b	
f	
e	
j	
d	
k	
h	
g	
Stack	**Assignment**

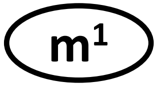
Example (K = 4)

m	
c	
b	
f	
e	
j	
d	
k	
h	
g	
Stack	**Assignment**

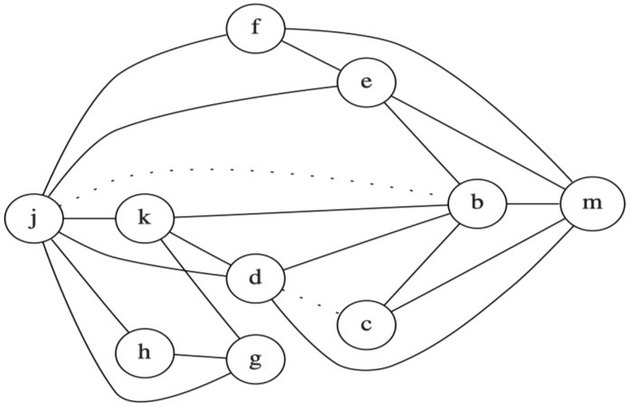
Example (K = 4)



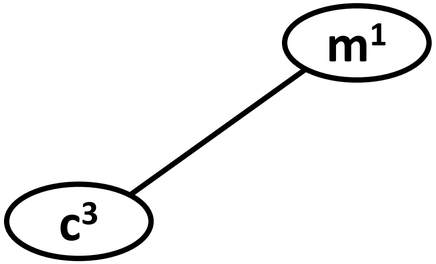
m	1
c	
b	
f	
e	
j	
d	
k	
h	
g	
Stack	**Assignment**



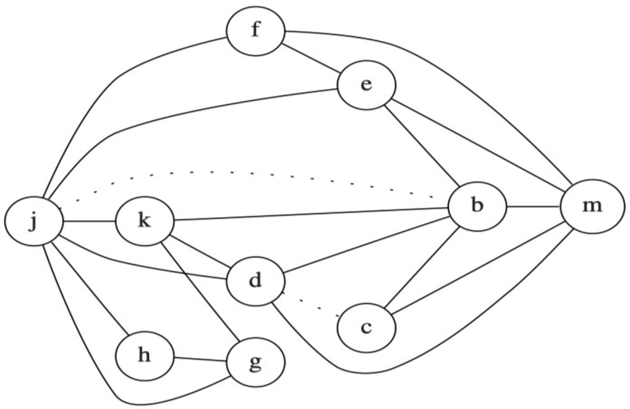
Example (K = 4)



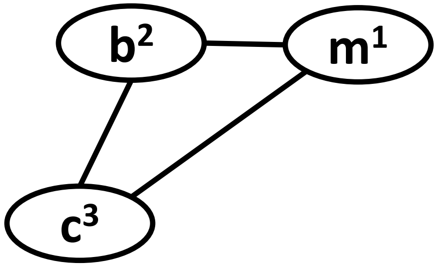
m	1
c	3
b	
f	
e	
j	
d	
k	
h	
g	
Stack	**Assignment**



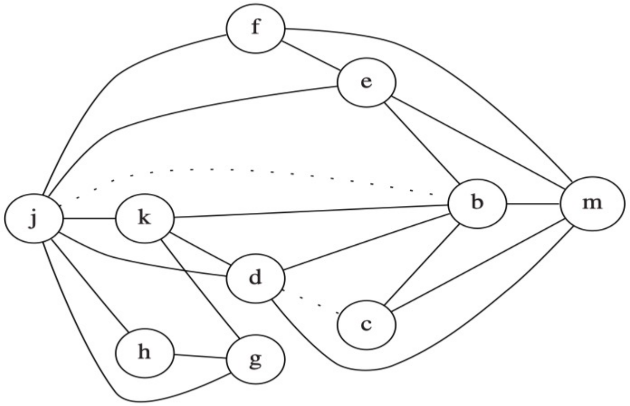
Example (K = 4)



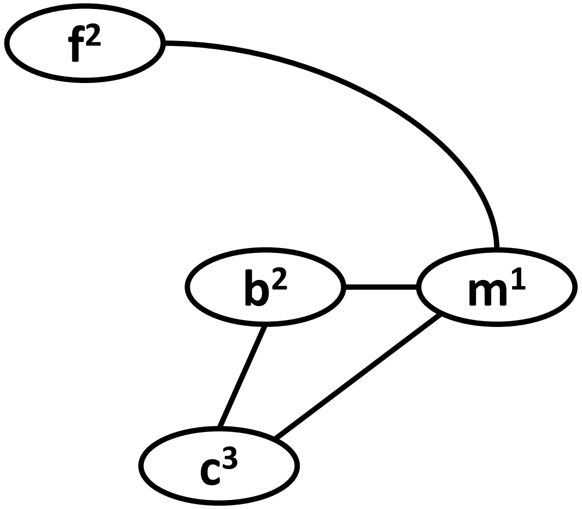
m	1
c	3
b	2
f	
e	
j	
d	
k	
h	
g	
Stack	**Assignment**



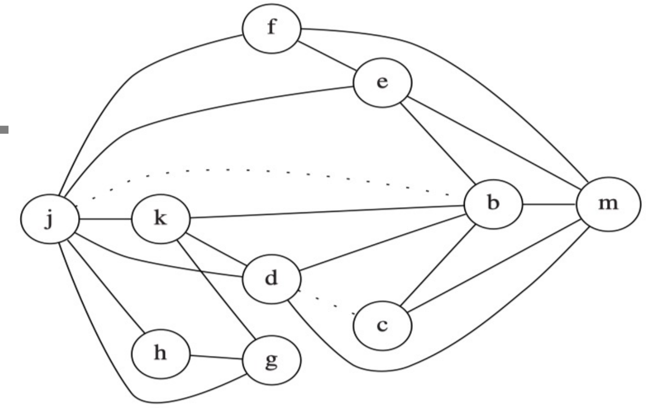
Example (K = 4)



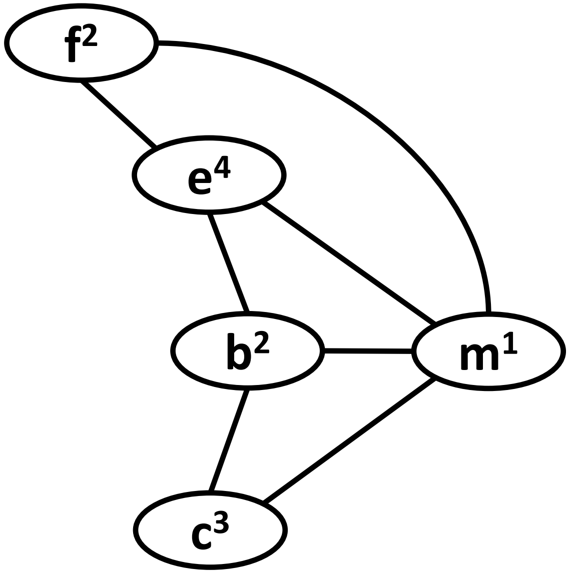
m	1
c	3
b	2
f	2
e	
j	
d	
k	
h	
g	
Stack	**Assignment**



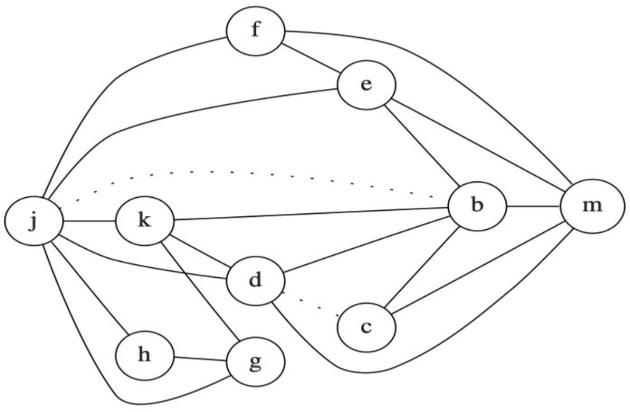
Example (K = 4)



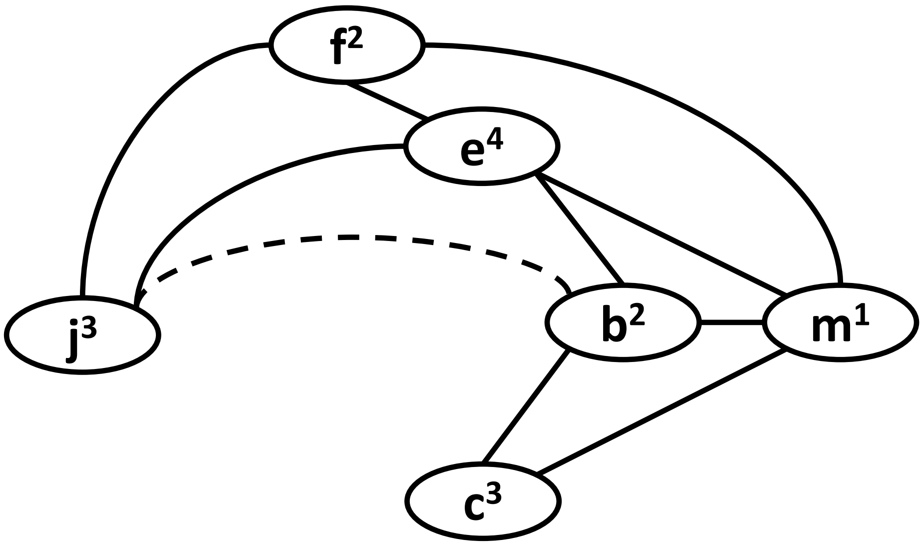
m	1
c	3
b	2
f	2
e	4
j	
d	
k	
h	
g	
Stack	**Assignment**



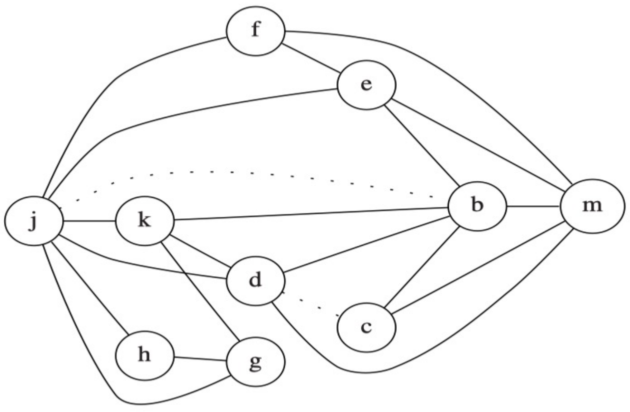
Example (K = 4)



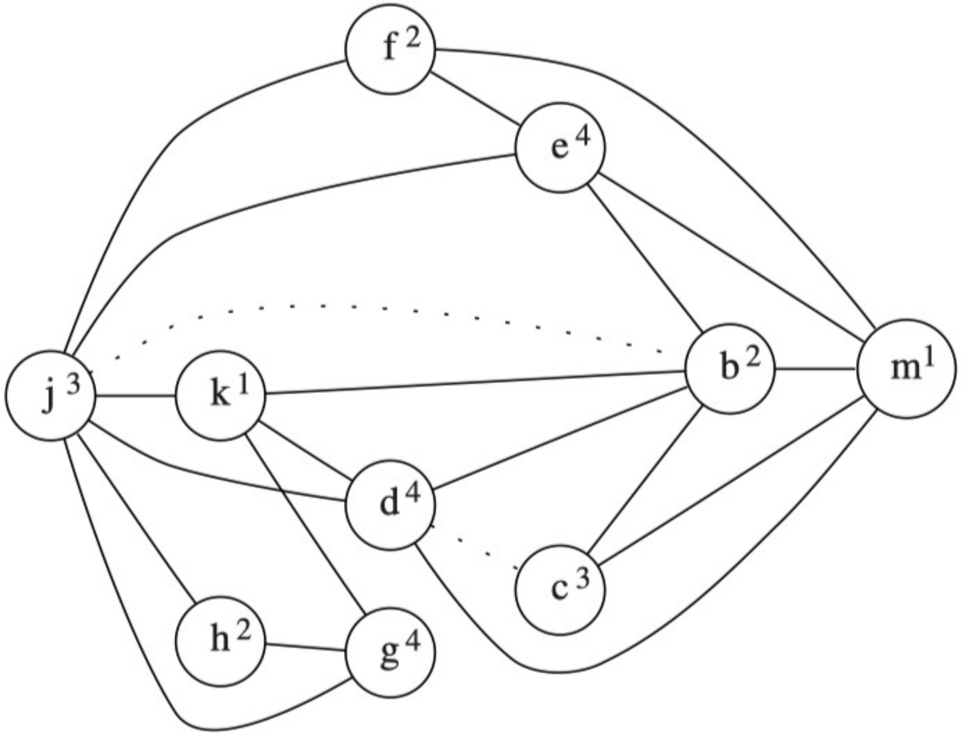
m	1
c	3
b	2
f	2
e	4
j	3
d	
k	
h	
g	
Stack	**Assignment**



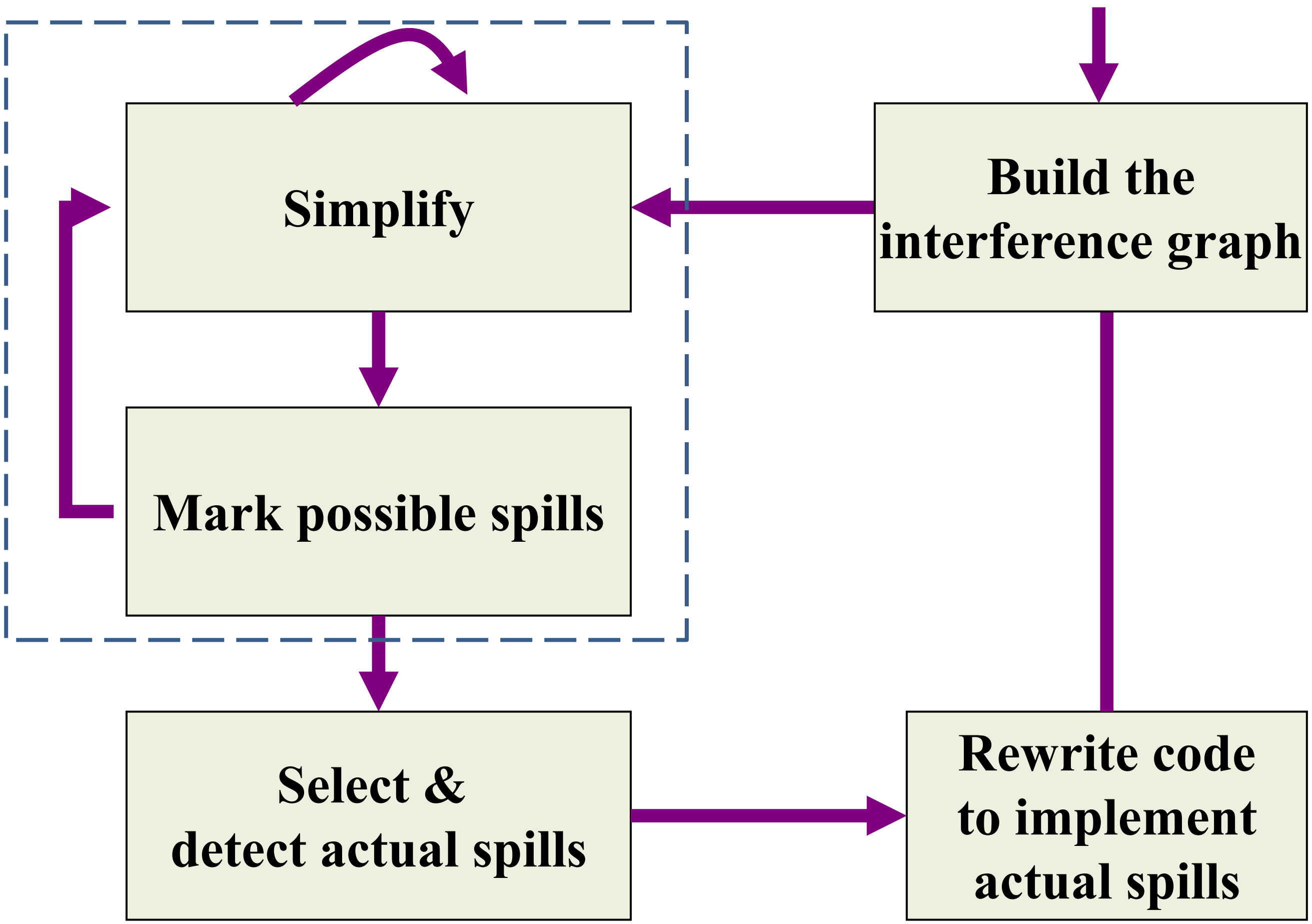
Example (K = 4)



m	1
c	3
b	2
f	2
e	4
j	3
d	4
k	1
h	2
g	4
Stack	**Assignment**

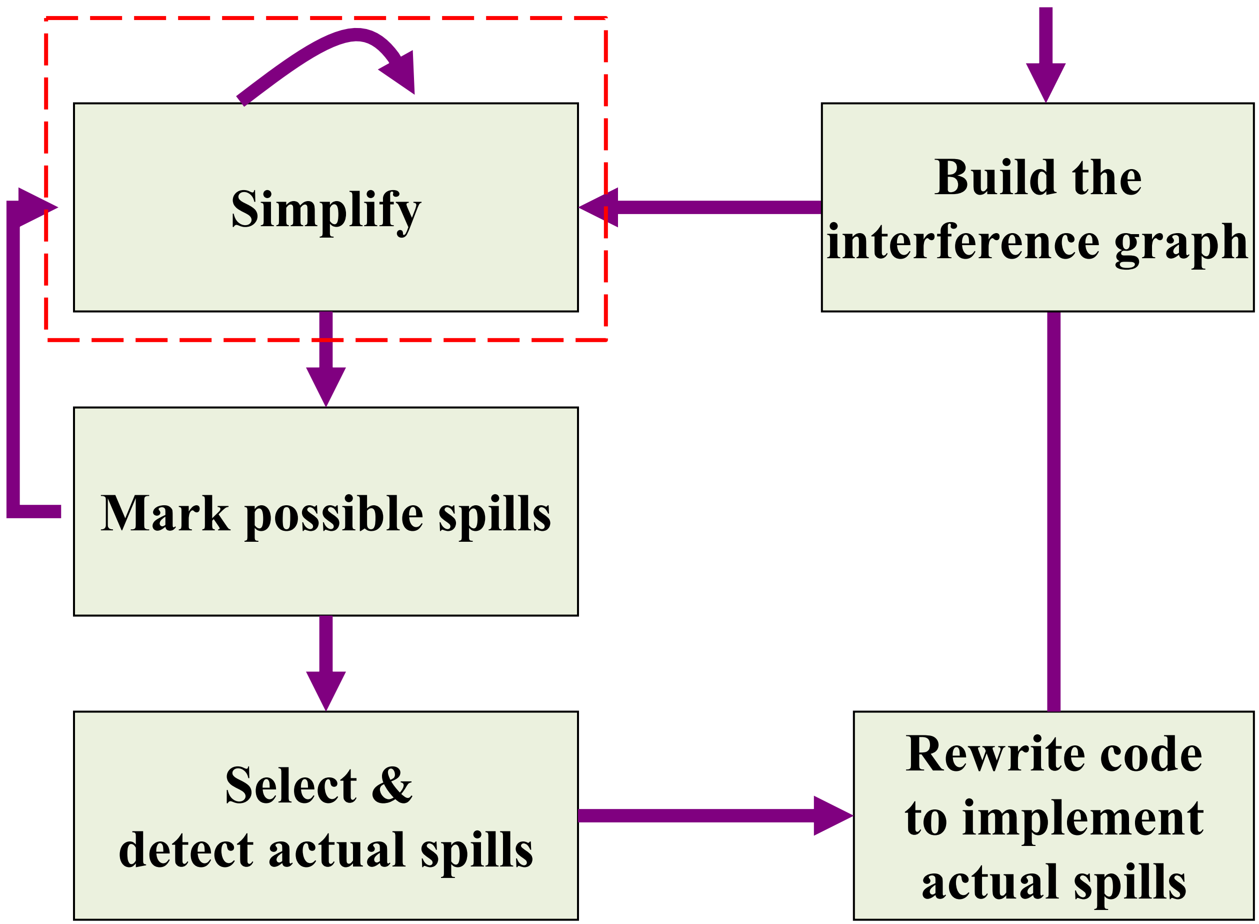


Recap: “Simplification → Select” Revised



Can we improve the above procedure?

Recap: “Simplification → Select” Revised



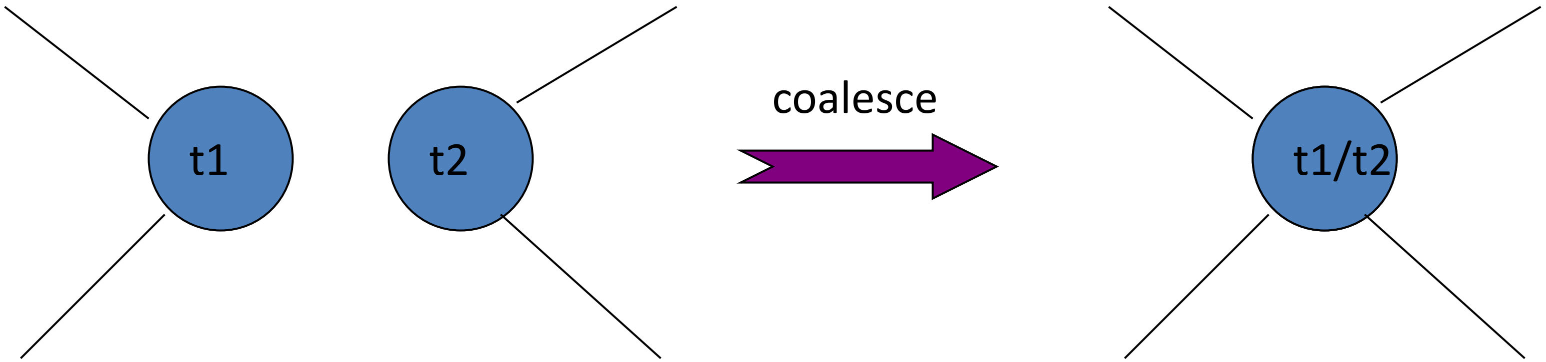
Let’s rethink about the simplification

3. Coloring By Simplifications

- Coloring by Simplifications
- Coalescing
- Precolored Nodes

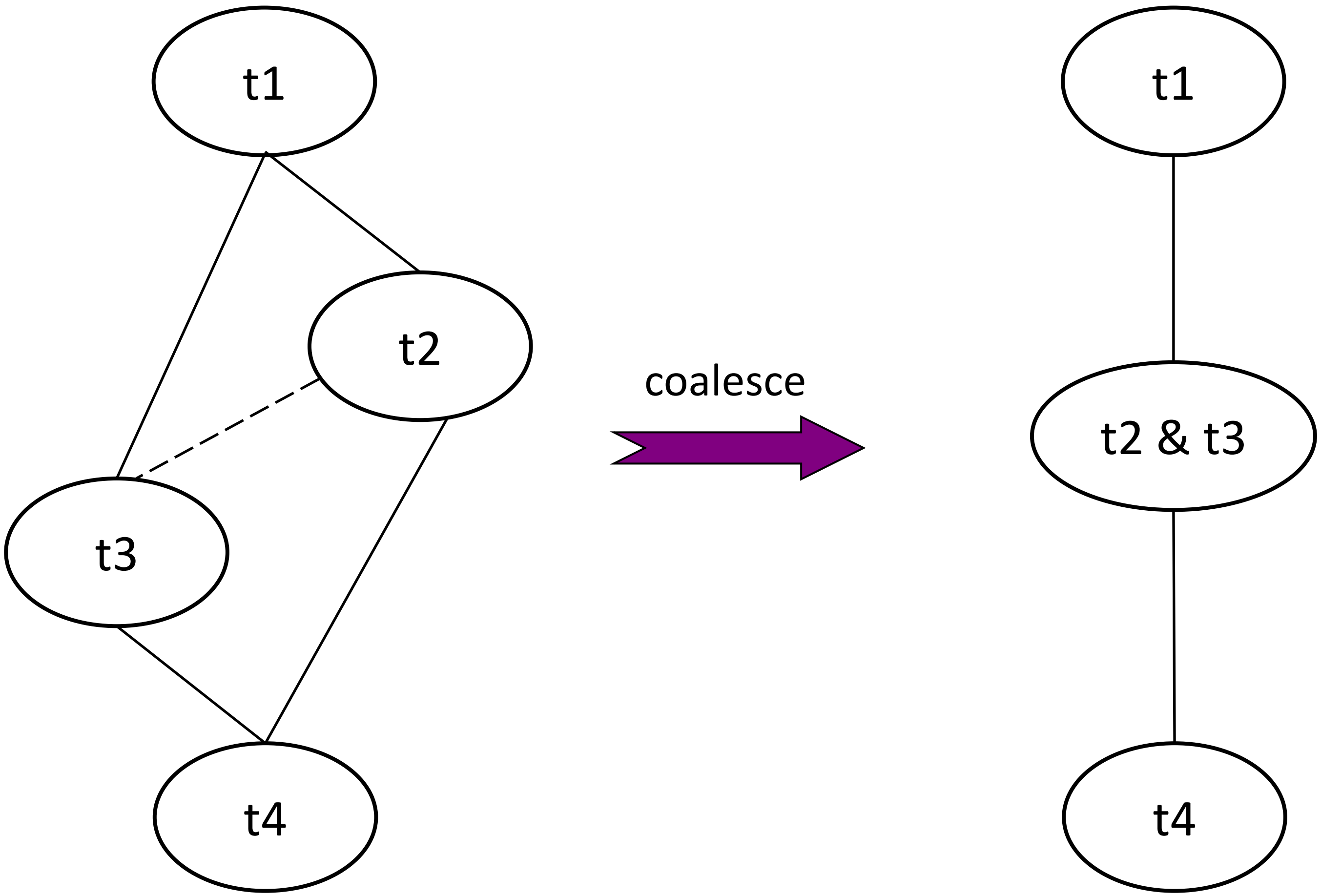
What is Coalescing

- If there is no edge in the interference graph between the source and destination of a MOVE
  1. The move instruction can be eliminated, and
  2. The source and destination nodes are coalesced into a new node, whose edges are the union of those of the nodes being replaced.



Why Coalescing

- Coalescing may improve the coloralibility



After coalescing, t1 and t4 have one neighbor less!

Why Not Coalescing

- **Problem**: coalescing may increase the number of interference edges and make a graph uncolorable!

- **Idea**: Conservative coalescing: don’t make it harder.
- **Solution**: Coalesce a and b if

Briggs	ab has fewer than k neighbors of significant degree.
George	every neighbor of a is • of insignificant degree • already interfering with b

Heuristic Coalescing

- **Briggs: avoid creation of high-degree (>= K) nodes**
  – Nodes a and b can be coalesced if the resulting node ab will have fewer than K neighbors of significant degree

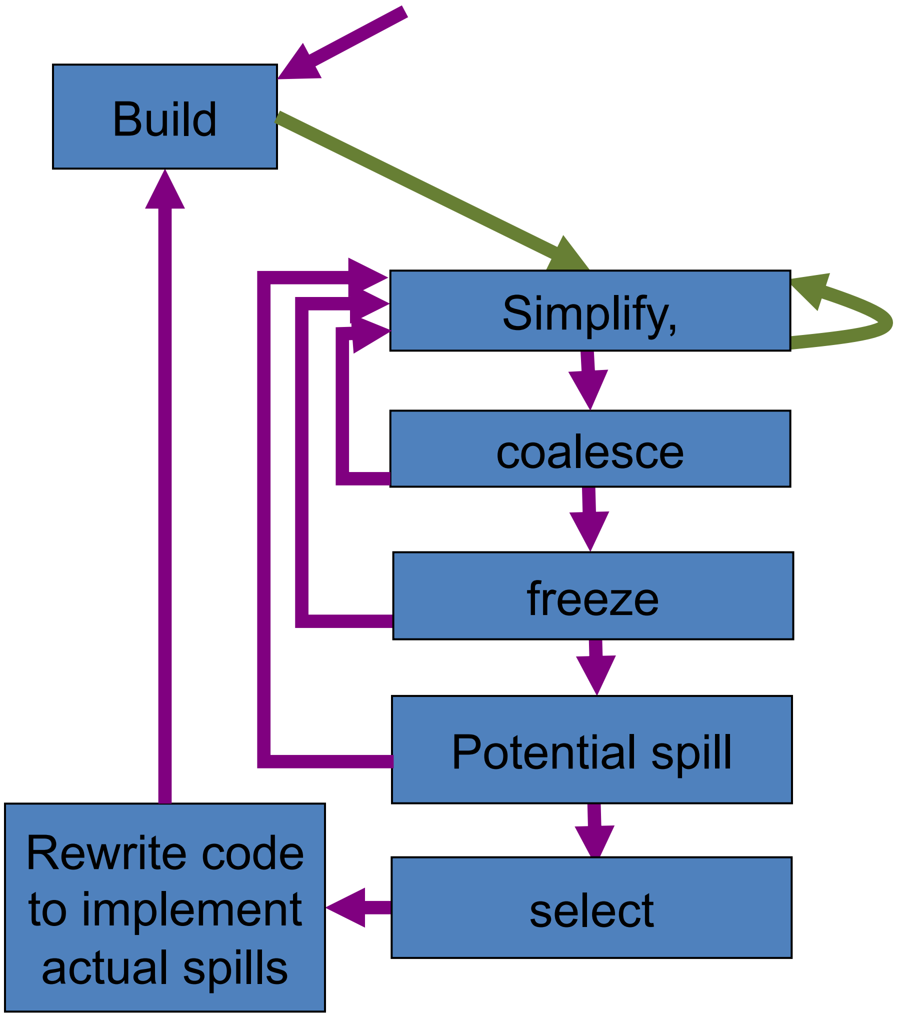
- The coalescing is guaranteed not to turn a K-colorable graph into a non-K-colorable graph. (Why?)
  – The simplify phase has removed all the insignificant-degree nodes from the graph.
  – The coalesced node will be adjacent only to those neighbors that were of significant degree.
  – fewer than K neighbors of significant degree => *simplify* can remove the coalesced node from the graph.

Heuristic Coalescing

- **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 insignificant degree (< K)

- This coalescing is safe (in terms of not turning a K-colorable graph into a non-K-colorable graph) (Why)
  - If t already interferes with b, (a, t) and (b, t) will be merged into (ab, t), not leading to the increase of degree.
  - if t is of insignificant degree, t will be removed by the **simplify** phase, also not leading to the increase of degree.

Coloring with Coalescing



- **Build**
- **Simplify**
  - Remove non-move-related nodes of low-degree
- **Coalesce**
  - Resulting node may become non-move-related node
- **Freeze**
  - Freeze the moves node of low-degree
- **Potential Spill**
- **Select**
- If failed, rewrite code to implement actual spill and rebuild the interference graph

Coloring with Coalescing

- The coalesce, simplify, and spill procedures should be alternated until the graph is empty

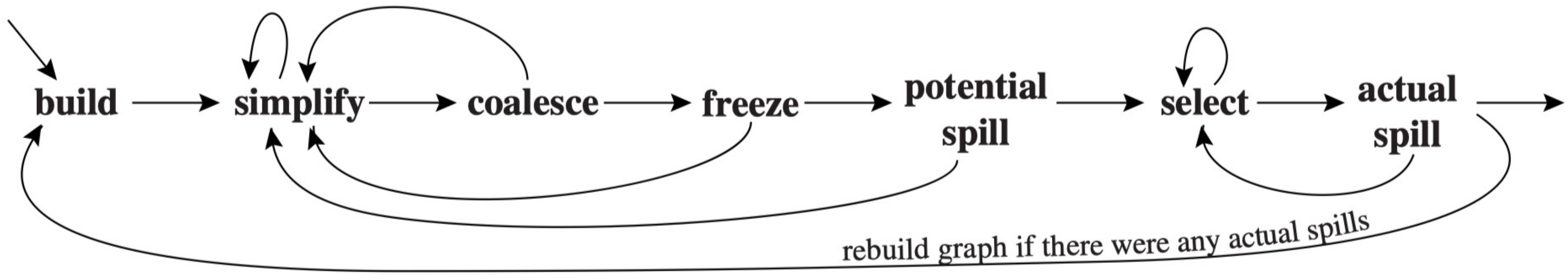


FIGURE 11.4. Graph coloring with coalescing.

Coloring with Coalescing

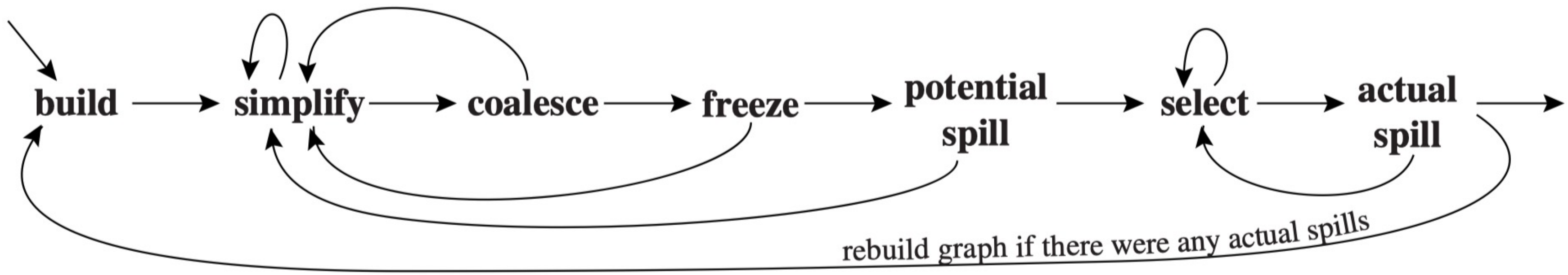


FIGURE 11.4. Graph coloring with coalescing.

1. **Build**
   - Construct the interference graph
   - Categorize each node as either move-related or non-move-related
     - A move-related node is one that is either the source or destination of a move instruction

Coloring with Coalescing

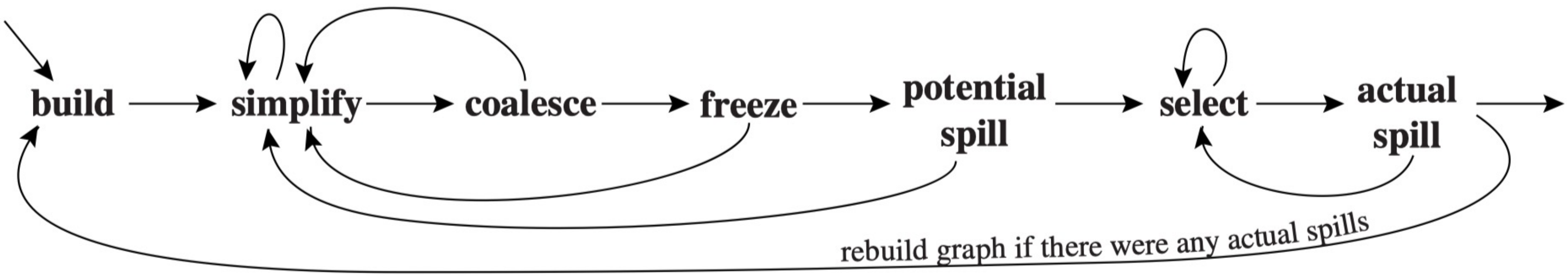


FIGURE 11.4. Graph coloring with coalescing.

2. Simplify

- One at a time, remove **non-move-related** nodes of low (<K) degree from the graph.

Coloring with Coalescing

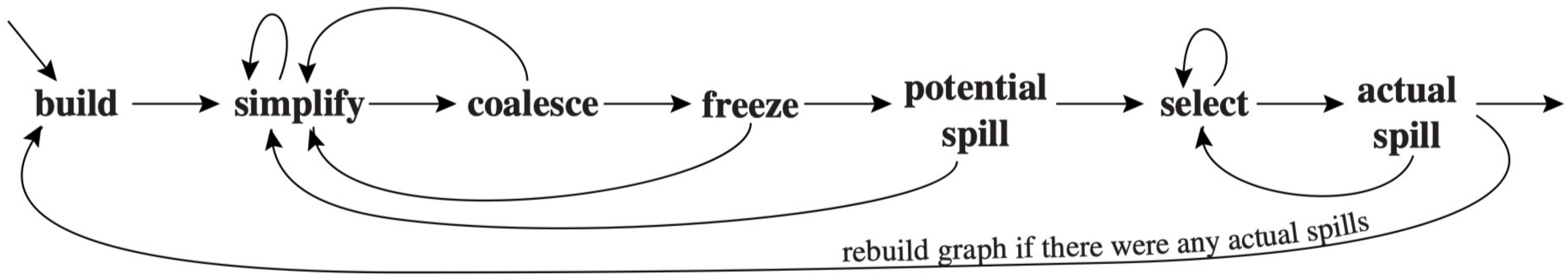


FIGURE 11.4. Graph coloring with coalescing.

3. Coalesce

- Perform conservative coalescing on the reduced graph.
- The resulting node is no longer move-related, and will be available for the next round of simplification.
- Simplify and coalesce are repeated until only significant-degree or move-related nodes remain.

Coloring with Coalescing

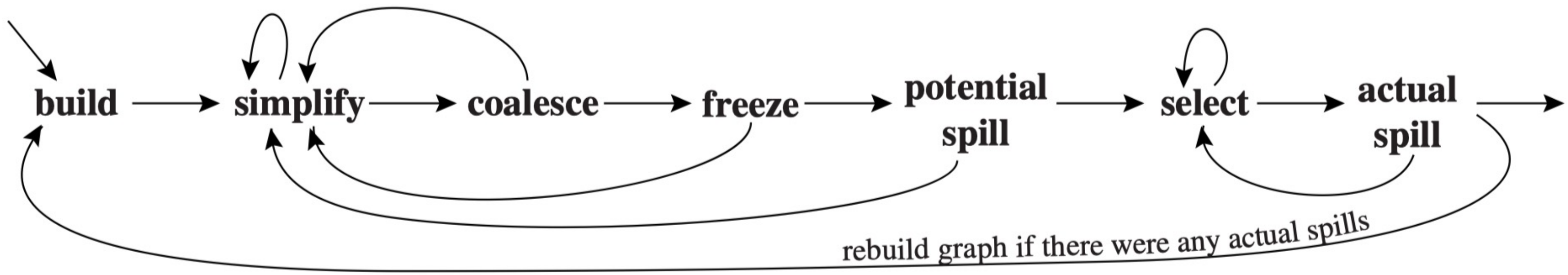


FIGURE 11.4. Graph coloring with coalescing.

4. Freeze

- If neither simplify nor coalesce applies, we look for a move-related node of low degree. We freeze the moves in which this node is involved.
  - We give up the hope of coalescing those moves.
  - Those nodes are considered non-move-related.
- Simplify and coalesce are resumed.

Coloring with Coalescing

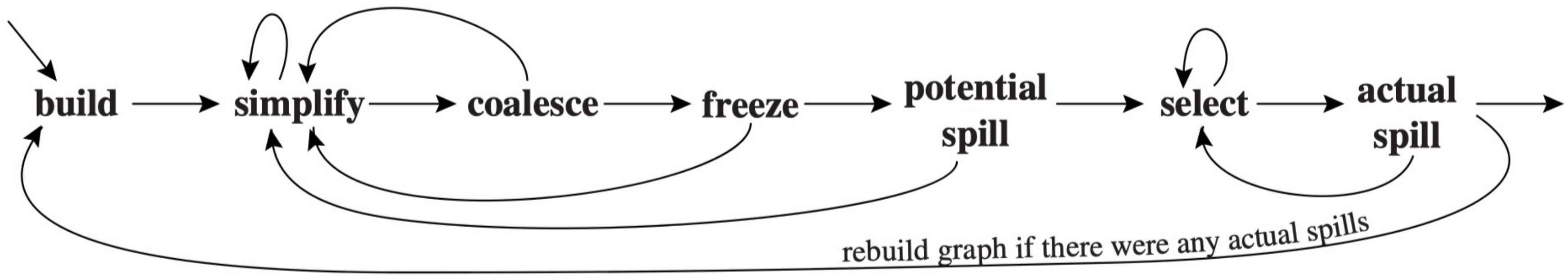


FIGURE 11.4. Graph coloring with coalescing.

5. Spill

- If there are no low-degree nodes, we select a significant-degree node for potential spilling and push it on the stack.

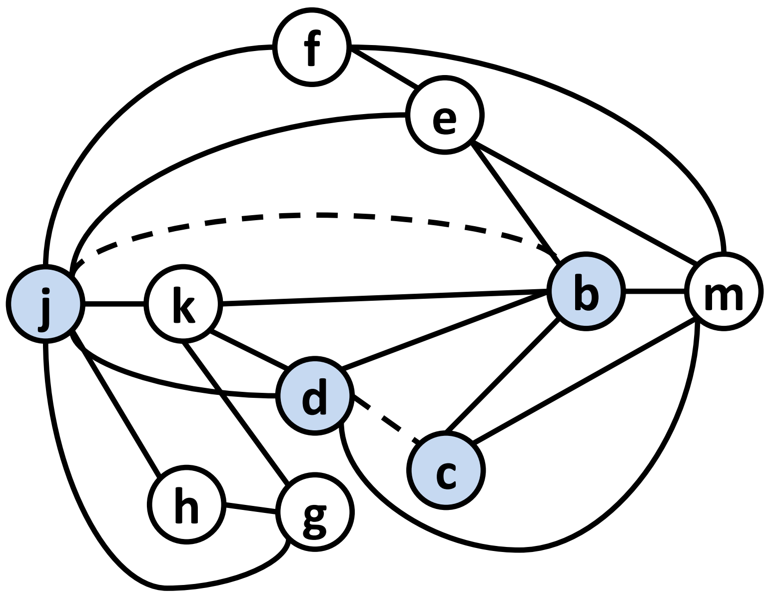
6. Select

- Pop the entire stack, assigning colors.

7. Rebuild graph if there are any actual spills!

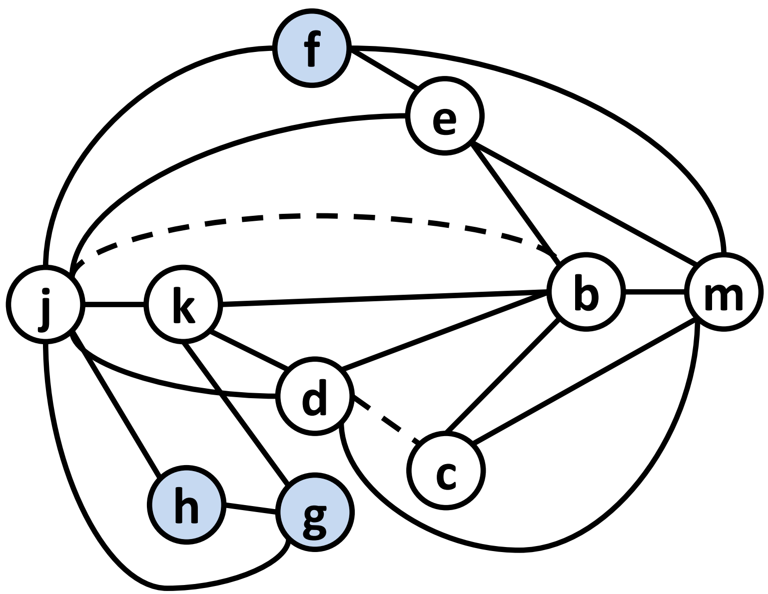
Example (K = 4), Revisited

- Nodes b, c, d and j are the only move-related nodes.



Example (K = 4), Revisited

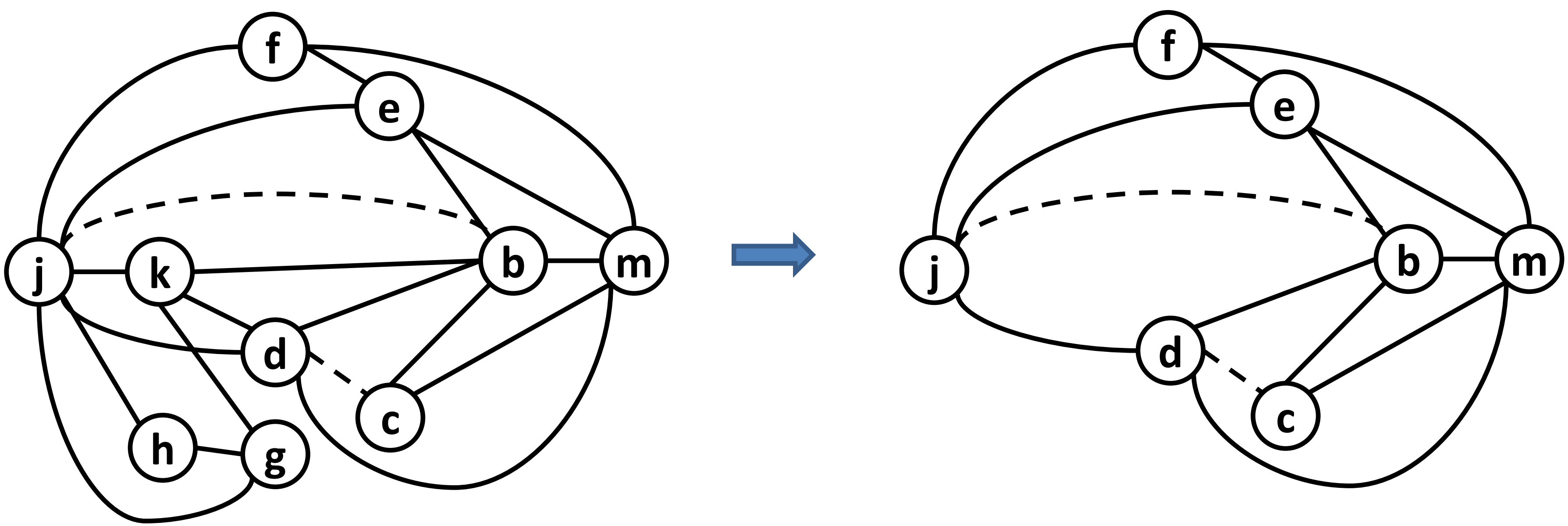
- Nodes b, c, d and j are the only move-related nodes.
- The initial work-list used in the simplify phase must contain only non-move-related nodes: g, h, f
  - (candidates for simplifications)



Why not select e, k, m as candidates of simplifications?

Example (K = 4), Revisited

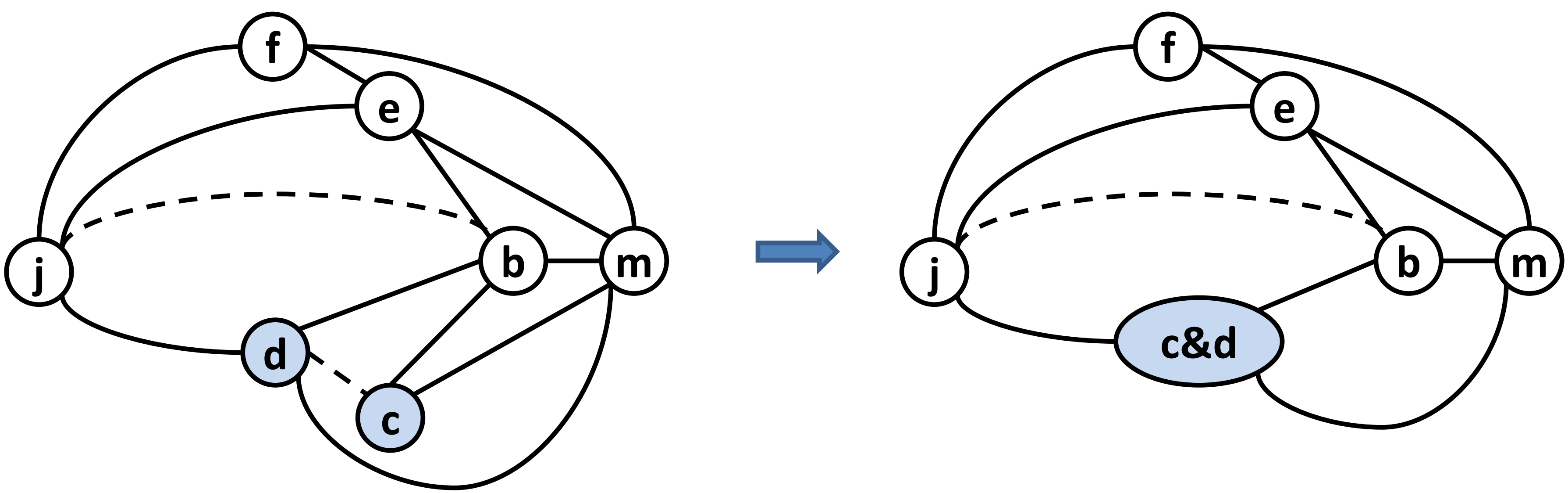
- Nodes b, c, d and j are the only move-related nodes.
- The initial work-list used in the simplify phase must contain only non-move-related nodes: g, h, f
- After removing g, h, f, we obtain the graph on the right



Example (K = 4), Revisited

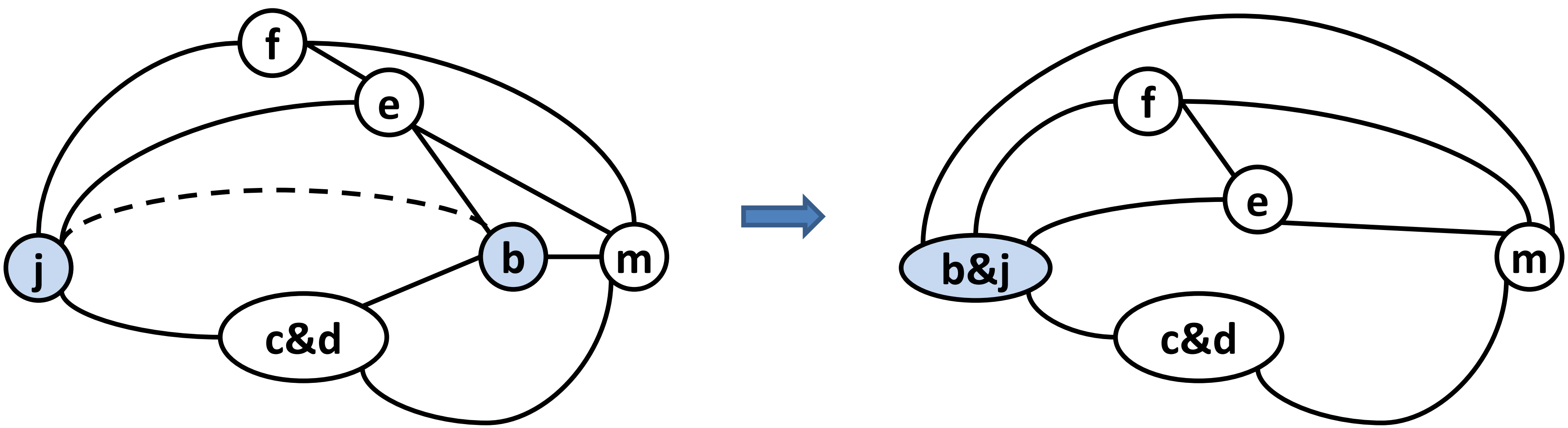
- If we invoke a round of coalescing at this point
  - We discover that c and d are indeed coalesceable.

  Why? (The coalesced node has only two neighbors of significant degree: `m` and `b`.)



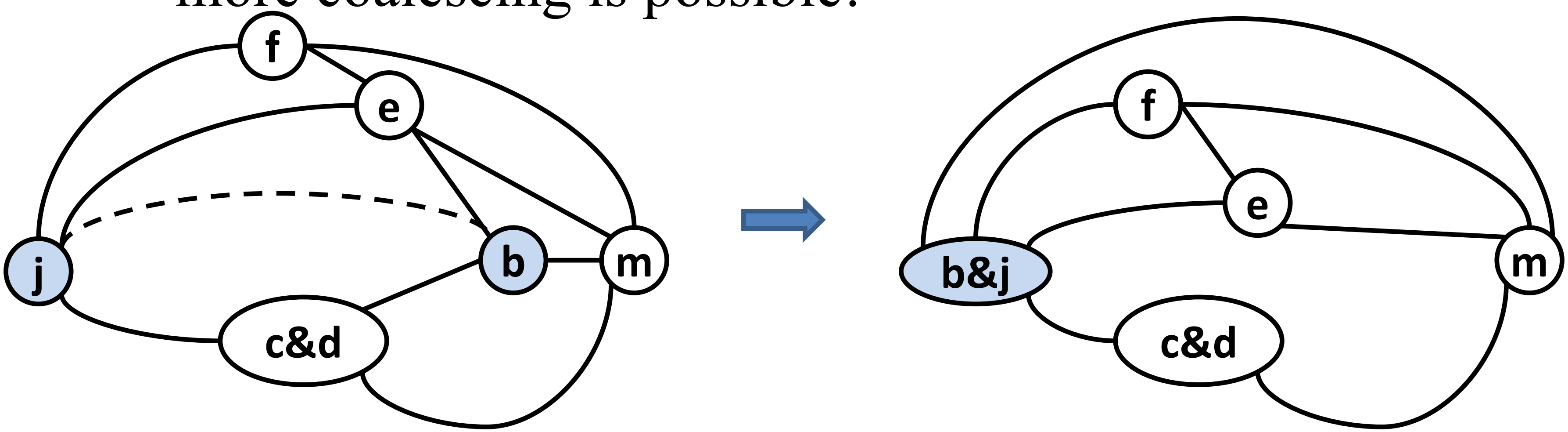
Example (K = 4), Revisited

- If we invoke a round of coalescing at this point
  - We discover that c and d are indeed coalesceable.
  - We further find that b and j are coalesceable



Example (K = 4), Revisited

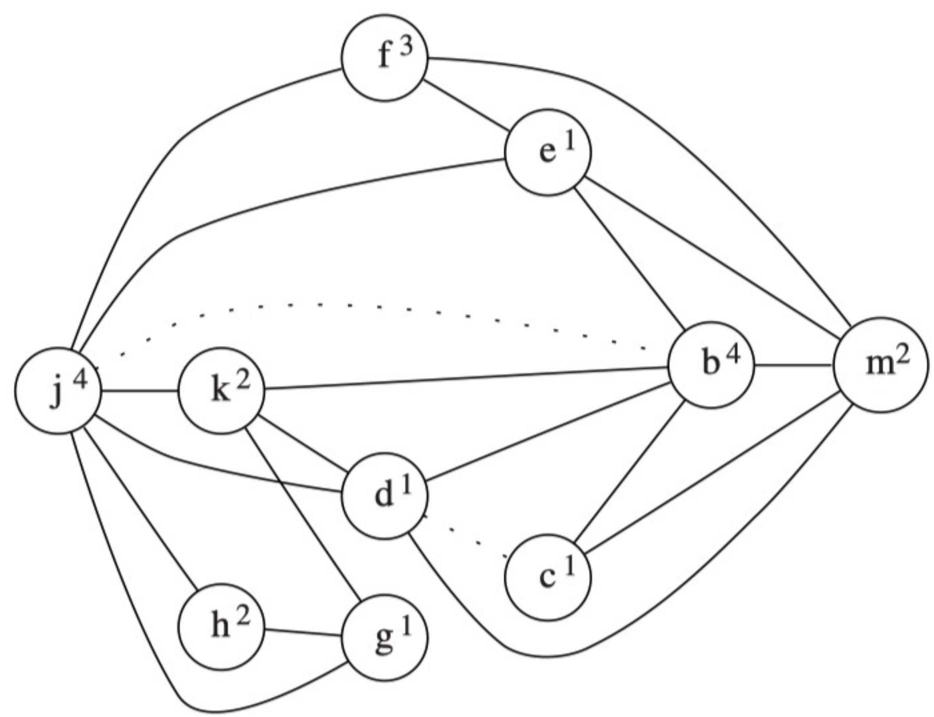
- If we invoke a round of coalescing at this point
  - We discover that c and d are indeed coalesceable.
  - We further find that b and j are coalesceable
  - There are no more move-related nodes, and therefore no more coalescing is possible!



Example (K = 4), Revisited

- The simplify phase can be invoked one more time to remove all the remaining nodes.
- A possible assignment of colors:

stack	coloring
e	1
m	2
f	3
j&b	4
c&d	1
k	2
h	2
g	1



3. Coloring By Simplifications

- Coloring by Simplifications
- Coalescing
- Precolored Nodes

Precolored Nodes

- Some real registers are used for special purposes
  - The stack point, frame point
  - The argument registers
  - The return value, return address
  - etc.
- For each of such registers, use the particular temporary that is permanently bound to that register
- Such temporaries are **precolored**.
  - Only one precolored node of each color
  - precolored nodes all interfere with each other.

Precolored Nodes

- It is common to give an ordinary temporary the same color as a precolored register, as long as they don't interfere.
  - A standard calling-convention register can be reused inside a procedure as a temporary variable
- We **cannot** simplify a precoloared node.
- We **should not** spill precolored nodes to memory.
  - machine registers are by definition registers

Temporary Copies of Machine Registers

- The coloring algorithm works by calling simplify, coalesce, and spill until only the precolored nodes remain
- Because precolored nodes do not spill, the front end must be careful to keep their live ranges short:
  - by generating MOVE instructions to move values to and from precolored nodes.
- Suppose r7 is a callee-save register:

```
Enter:    def(r7)

             ⋮

Exit:     use(r7)
```

```
Enter:    def(r7)
          t231 ← r7
          ⋮
          r7 ← t231
Exit:     use(r7)
```

- If there is register pressure (a high demand for registers) in this function, t_{231} will spill; otherwise t_{231} will be coalesced with r7 and the MOVE instructions will be eliminated.

Caller-Save and Callee-Save Registers

```
foo (){
    t = ...
    ... = ... t ...
    s = ...
    f()
    g()
    ... = ... s ...
}
```

A local variable or compiler temporary that is not live across any procedure call should usually be allocated to a caller-save register

Any variable that is live across several procedure calls should be kept in a callee-save-register

- If a variable x is live across a procedure call,
  - then it interferes with all the caller-save (precolored) registers
  - *and* it interferes with all the new temporaries created for callee-save registers (e.g., t231)
  - a spill will occur
  - But which variable will be spilled first? x or t231?

Put it all Together Example(K=3)

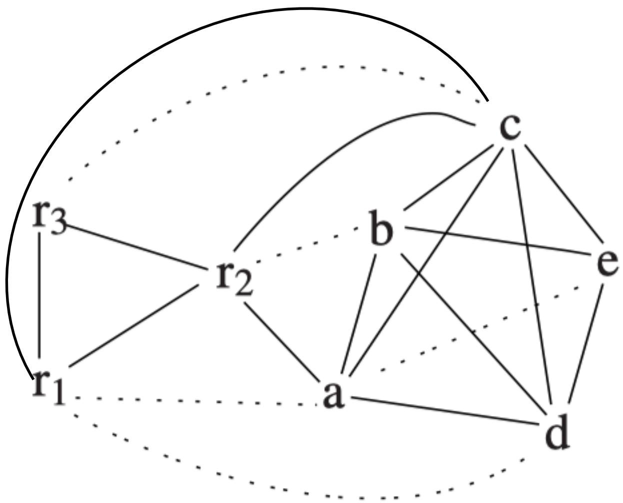
```
int f(int a, int b) {
  int d = 0;
  int e = a;
  do {
      d = d+b;
      e = e-1;
  } while (e>0);
  return d;
}
```

enter:	$c \leftarrow r3$
	$a \leftarrow r1$
	$b \leftarrow r2$
	$d \leftarrow 0$
	$e \leftarrow a$
loop:	$d \leftarrow d + b$
	$e \leftarrow e - 1$
	if $(e > 0)$ goto loop
	$r1 \leftarrow d$
	$r3 \leftarrow c$
	return (r1, r3 live out)

For a machine with 3 registers:

- r1 and r2 are caller-save
- r3 is callee save

Put it all Together Example(K=3)

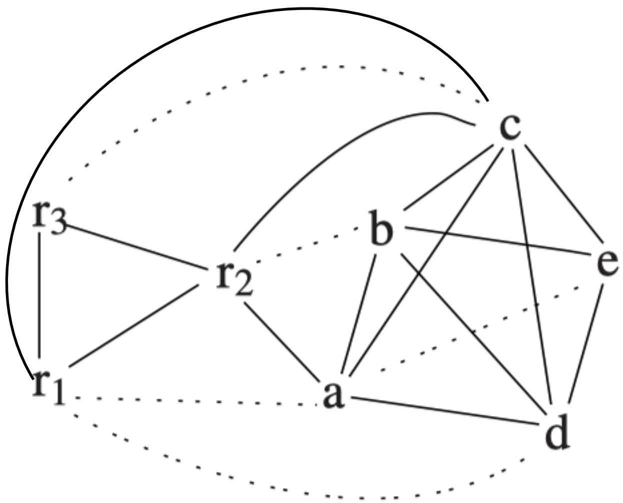


Interference Graph

```
enter:   c←r3
         a ← r1
         b ← r2
         d ← 0
         e ← a
loop:    d ← d + b
         e ← e – 1
         if (e > 0) goto loop
         r1 ← d
         r3 ← c
         return    (r1, r3 live out)
```

- No opportunity for simplify or freeze
  - All the non-precolored nodes have degree >= K
- We must spill some node
- How to choose the node to spill?
- Answer: the node with mode degrees but is rarely used
  - Why?

Put it all Together Example(K=3)

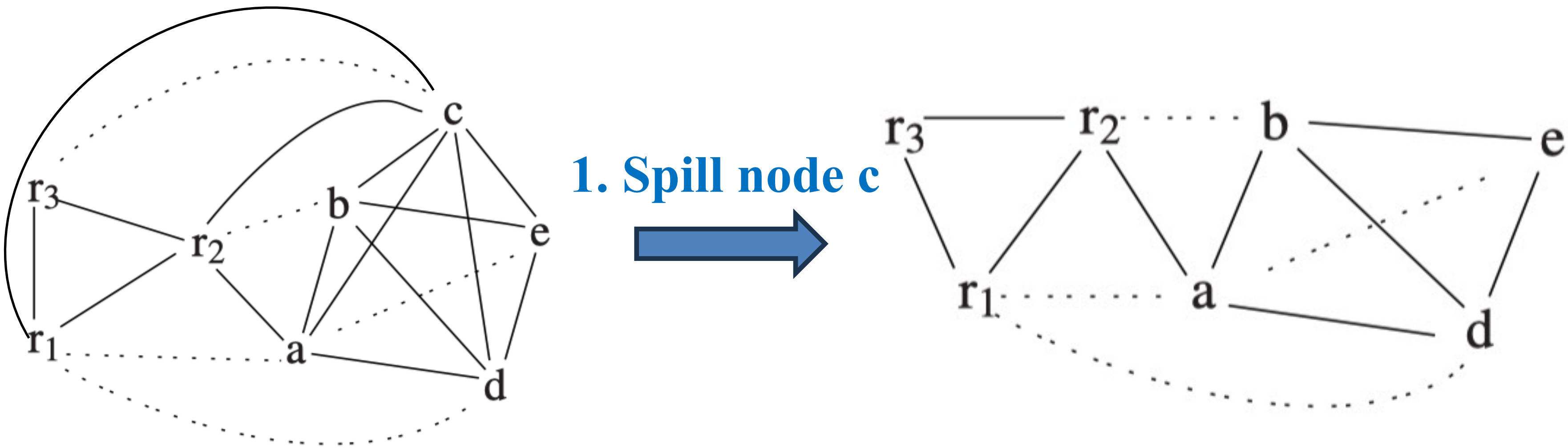


Interference Graph

```
enter:  c←r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop:   d ← d + b
        e ← e – 1
        if (e > 0) goto loop
        r1 ← d
        r3 ← c
        return   (r1, r3 live out)
```

Node	Use+Def Outside loop	Use+Def inside loop	Degree	Spill priority
a	2	0	4	(2+10*0)/4=0.5
b	1	1	4	(1+10*1)/4=2.75
c	2	0	6	(2+10*0)/6=0.33
d	2	2	4	(2+10*2)/4=5.5
e	1	3	3	(1+10*3)/3=10.33

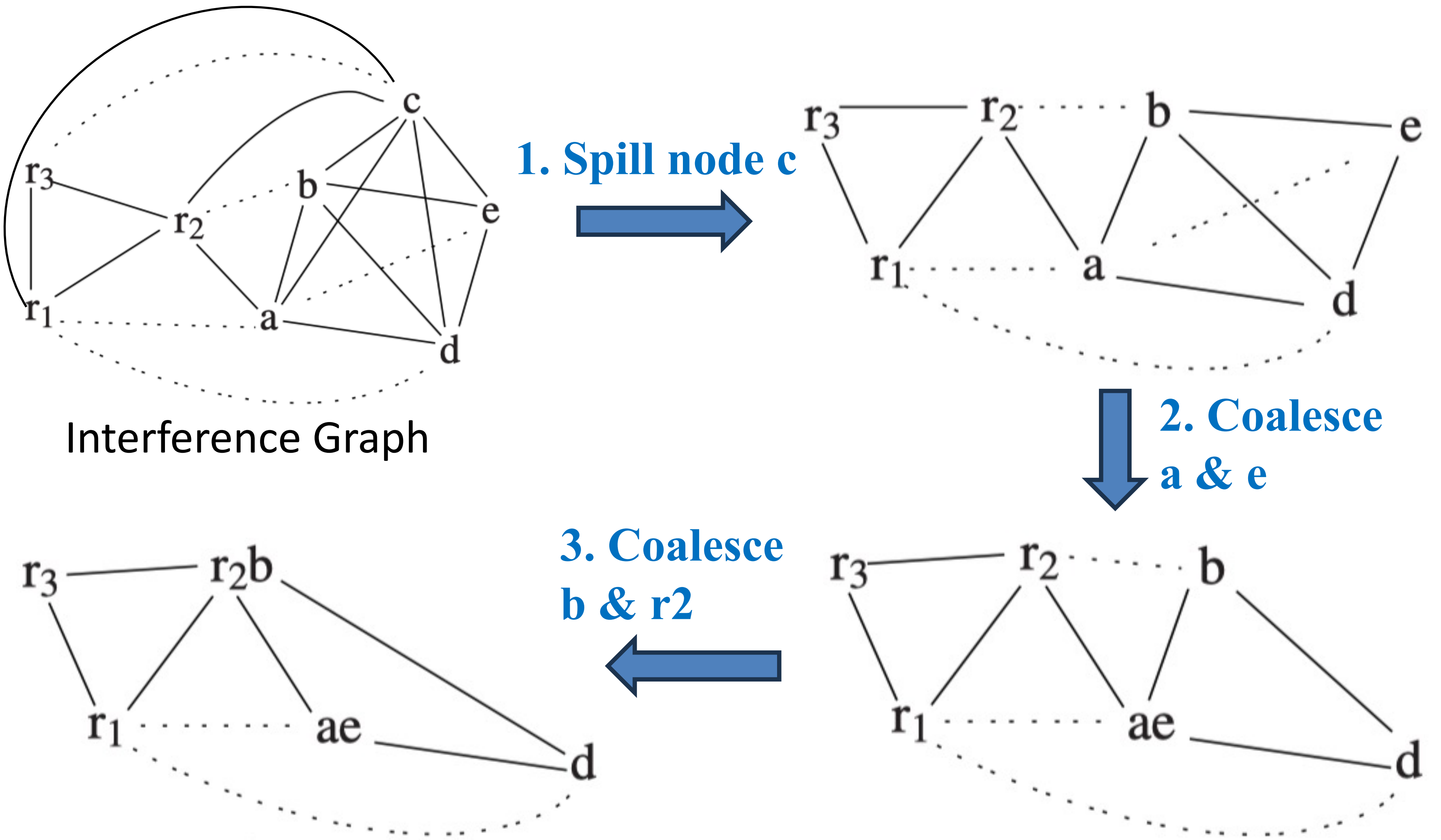
Put it all Together Example(K=3)



Interference Graph

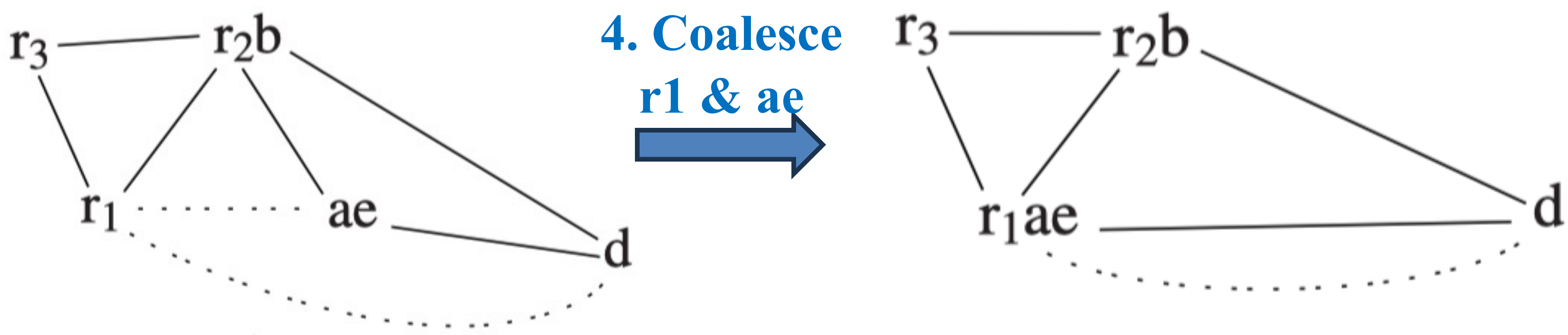
- Spill node c
- No simplify is possible
  - All non-precolored nodes are move-related

Put it all Together Example(K=3)



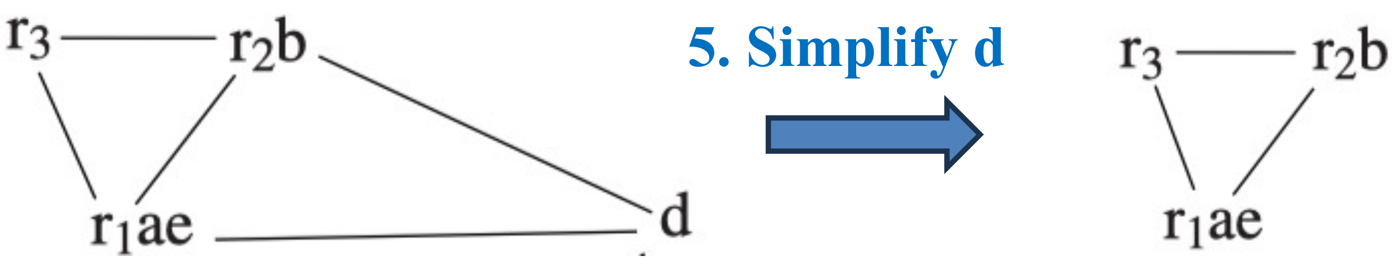
- Perform coalescing

Put it all Together Example(K=3)



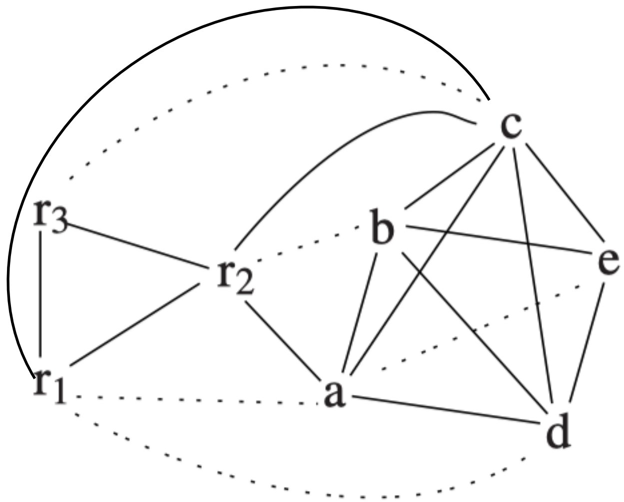
- Perform coalescing
- Now, can we coalesce r1ae and d?
  - No, r1ae interferes with d
- The move between r1ae and d is **constrained**
  - We remove it from further considerations
  - d is no longer treated as move-related

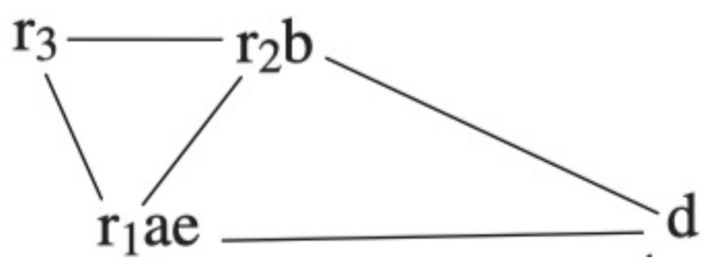
Put it all Together Example(K=3)

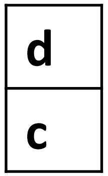


- We must simplify d
- Now, only precolored nodes.

Put it all Together Example(K=3)







6. Select

- Pop nodes from the stack and assign color to them:
  - Pick d, assign color r3
  - Nodes a, b, e have already been assigned colors by coalescing
  - Pop c: c turns into an actual spill

Put it all Together Example(K=3)

7. Rewrite

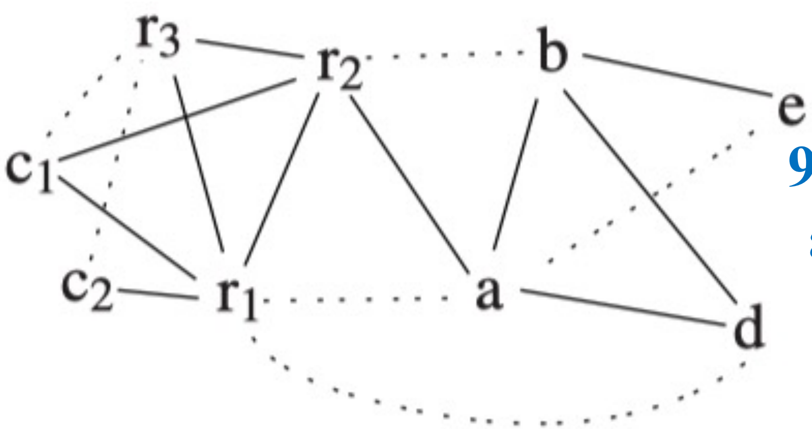
- Before each use -> fetch
- After each def -> store

```
enter:  c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop:   d ← d + b
        e ← e – 1
        if (e > 0) goto loop
        r1 ← d
        r3 ← c
        return
```

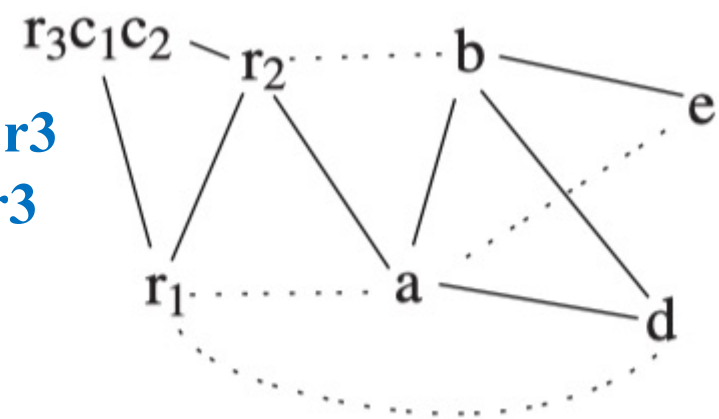
rewrite

```
enter:  c1 ← r3
        M[cloc] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop:   d ← d + b
        e ← e – 1
        if (e > 0) goto loop
        r1 ← d
        c2 ← M[cloc]
        r3 ← c2
        return
```

Put it all Together Example(K=3)

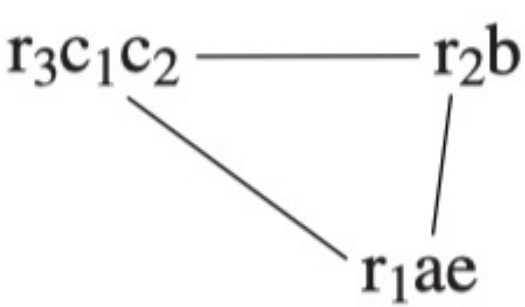


9. coalesce c1&r3 and then c2&r3

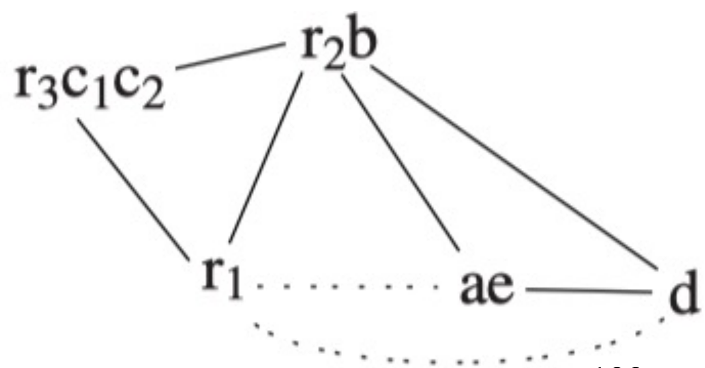


8. Build a new inference graph

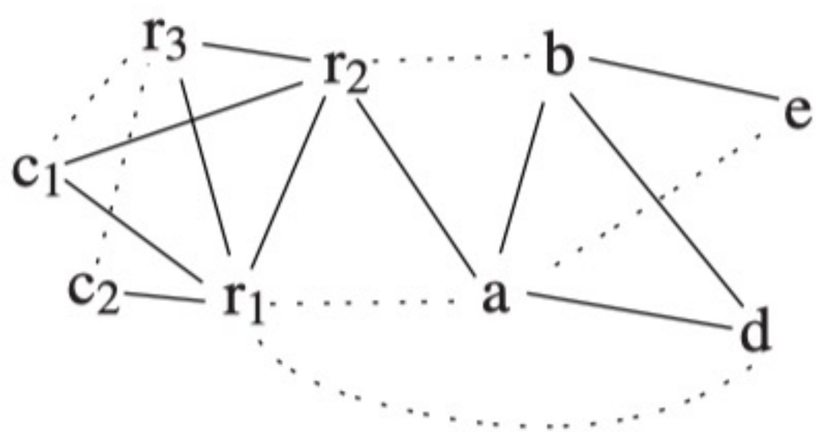
10. coalesce a&e and then b&r2



11. coalesce ae&r1 and then simplify d



Put it all Together Example(K=3)



Node	Color
a	r1
b	r2
c	r3
d	r3
e	r1

12. Select

- Poping from the stack and select color r3 for d
- all other nodes were coalesced or precolord

Put it all Together Example(K=3)

13. Rewrite the program using the register assignment

```
enter:  c1 ←r3
        M[cloc] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop:   d ← d + b
        e ← e – 1
        if (e > 0) goto loop
        r1 ← d
        c2 ← M[cloc]
        r3 ← c2
        return
```

Node	Color
a	r1
b	r2
c	r3
d	r3
e	r1

```
enter:  r3 ←r3
        M[cloc] ← r3
        r1 ← r1
        r2 ← r2
        r3 ← 0
        r1 ← r1
loop:   r3 ← r3 + r2
        r1 ← r1 – 1
        if (r1 > 0) goto loop
        r1 ← r3
        r3 ← M[cloc]
        r3 ← r3
        return
```

Put it all Together Example(K=3)

14. Delete any move instruction whose source and destination are the same:

enter:	$r3 \leftarrow r3$
	$M[c_{loc}] \leftarrow c_1$
	$r1 \leftarrow r1$
	$r2 \leftarrow r2$
	$r3 \leftarrow 0$
	$r1 \leftarrow r1$
loop:	$r3 \leftarrow r3 + r2$
	$r1 \leftarrow r1 - 1$
	if $(r1 > 0)$ goto loop
	$r1 \leftarrow r3$
	$r3 \leftarrow M[c_{loc}]$
	$r3 \leftarrow r3$
	return

$\Rightarrow$

enter:	$M[c_{loc}] \leftarrow c_1$
	$r3 \leftarrow 0$
loop:	$r3 \leftarrow r3 + r2$
	$r1 \leftarrow r1 - 1$
	if $(r1 > 0)$ goto loop
	$r1 \leftarrow r3$
	$r3 \leftarrow M[c_{loc}]$
	return

Summary

- Register allocation has three major parts
  - Liveness analysis
  - Graph coloring
  - Program transformation (move coalescing and spilling)
- Register allocation by graph coloring
  - Build
  - Simplify
  - Coalesce
  - Freeze
  - Spill
  - Select

Thank you all for your attention