

并行编译与优化 Parallel Compiler and Optimization

计算机研究所编译系统室

Lecture 11: Register Allocation Part2

第十一课：寄存器分配（二）

■ 寄存器分配

- ⊕ 将程序中的变量分配到寄存器或者存储器中

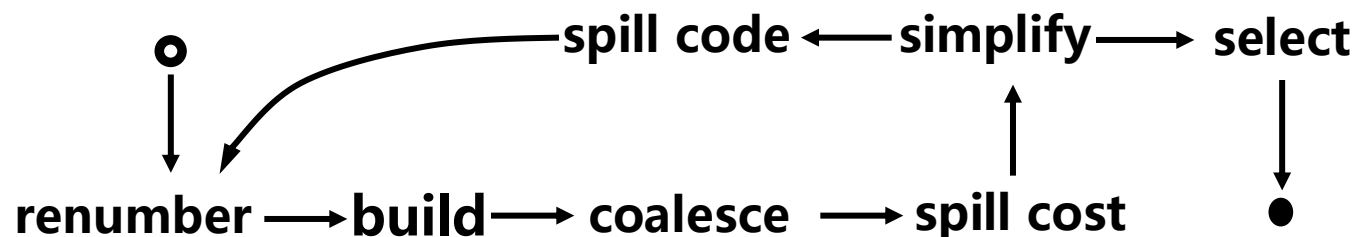
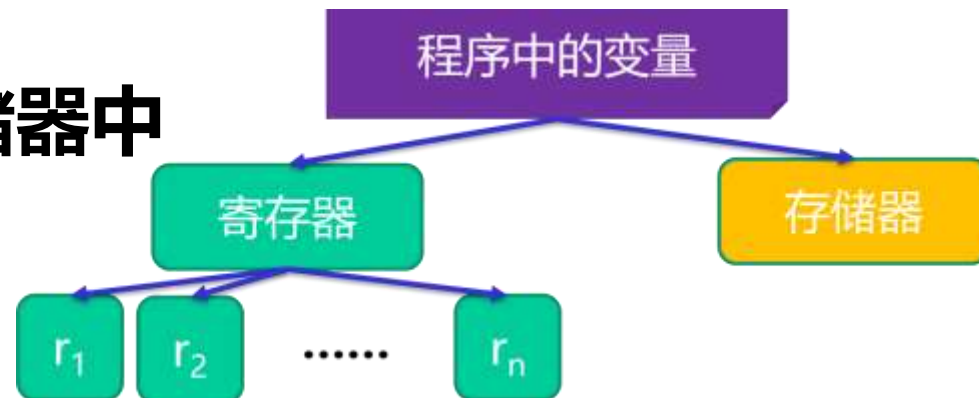
■ 寄存器分配方法

- ⊕ 基于**使用计数**的寄存器分配方法

- 局部方法(基本块)、全局方法(循环内跨越基本块边界)

- ⊕ 基于**图着色**的寄存器分配方法

- 全局方法(函数): 由Chaitin首次实现, 奠定了图着色寄存器分配算法的基础

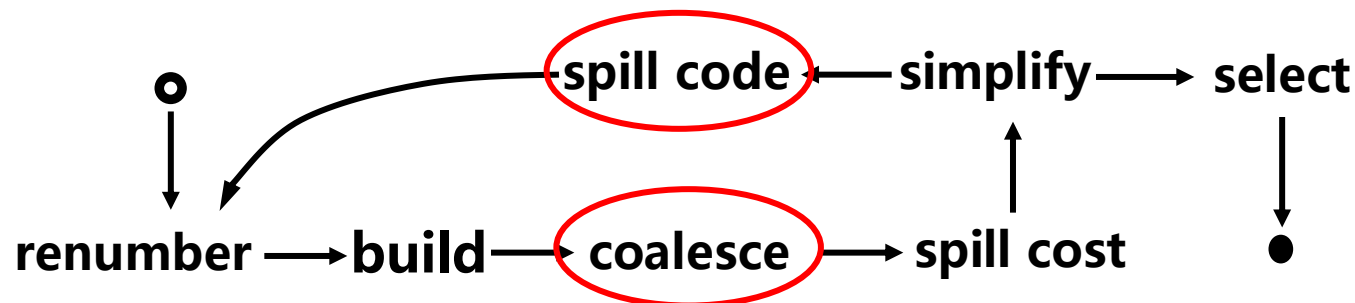


■是迭代方法，非常耗时

- ⊕一旦发生溢出则需要重构冲突图，重新进行整个寄存器分配流程

■合并策略过于激进

- ⊕无条件合并传送相关且无冲突边相连的结点
- ⊕可能导致合并后结点的度数增高，进而导致原来是 k -可着色的图变成不能用 k 种颜色着色，从而产生更多溢出



尽可能减少不必要的溢出

10.1 寄存器分配概述

10.2 基于使用计数的寄存器分配方法

10.3 基于图着色的寄存器分配方法

10.4 Briggs图着色寄存器分配改进算法

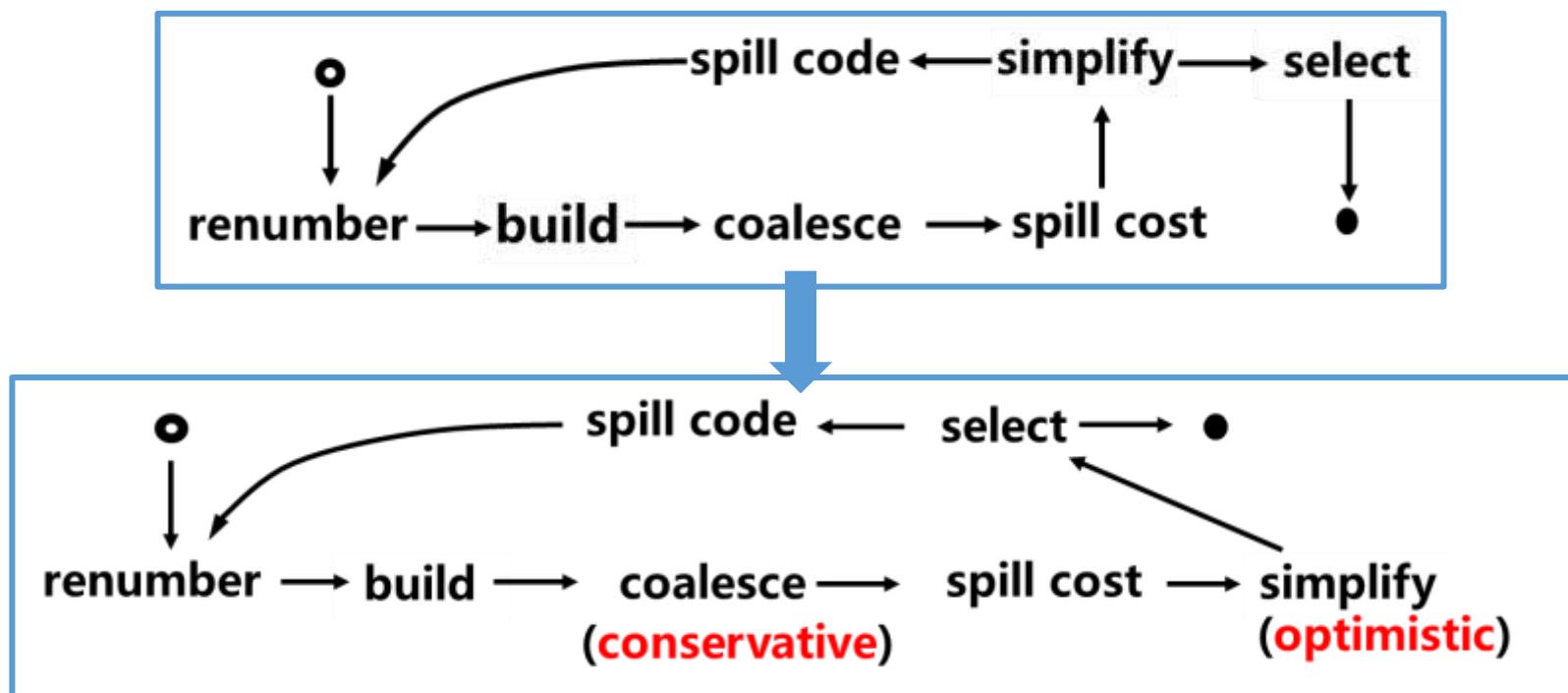
10.5 George图着色寄存器分配改进算法

10.6 基于线性扫描的寄存器分配方法

- 能够运用Briggs图着色寄存器分配改进算法，George图着色寄存器分配改进算法，以及线性扫描寄存器分配方法对给定代码进行寄存器分配
- 理解不同寄存器分配方法的优缺点

■ 针对Chaitin算法，提出两项改进

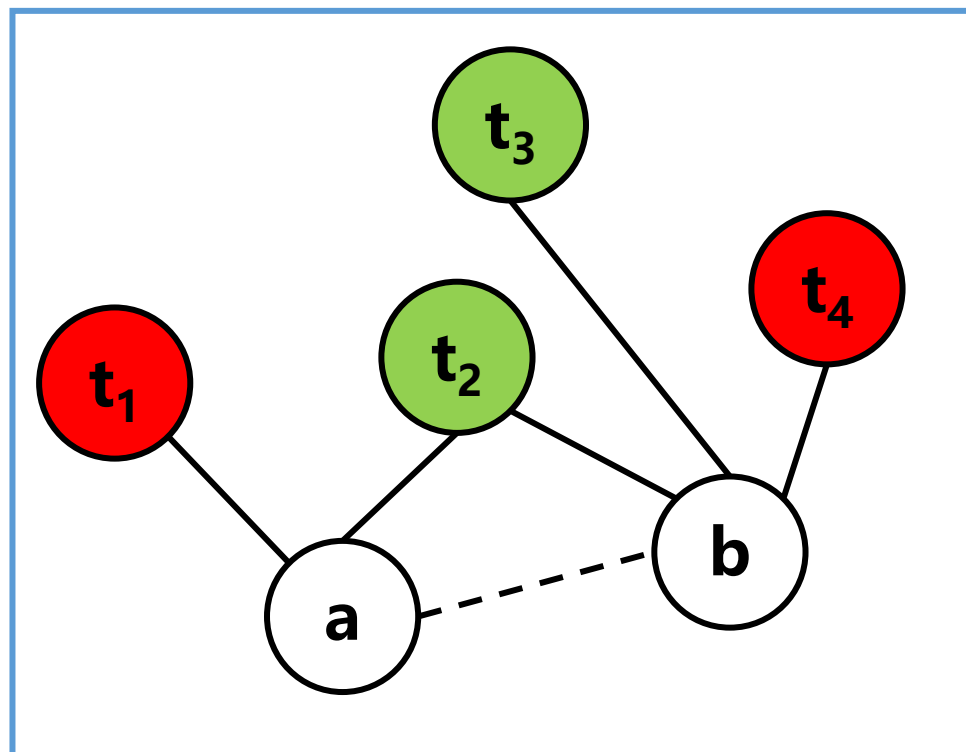
- ⊕ 保守合并
- ⊕ 乐观着色



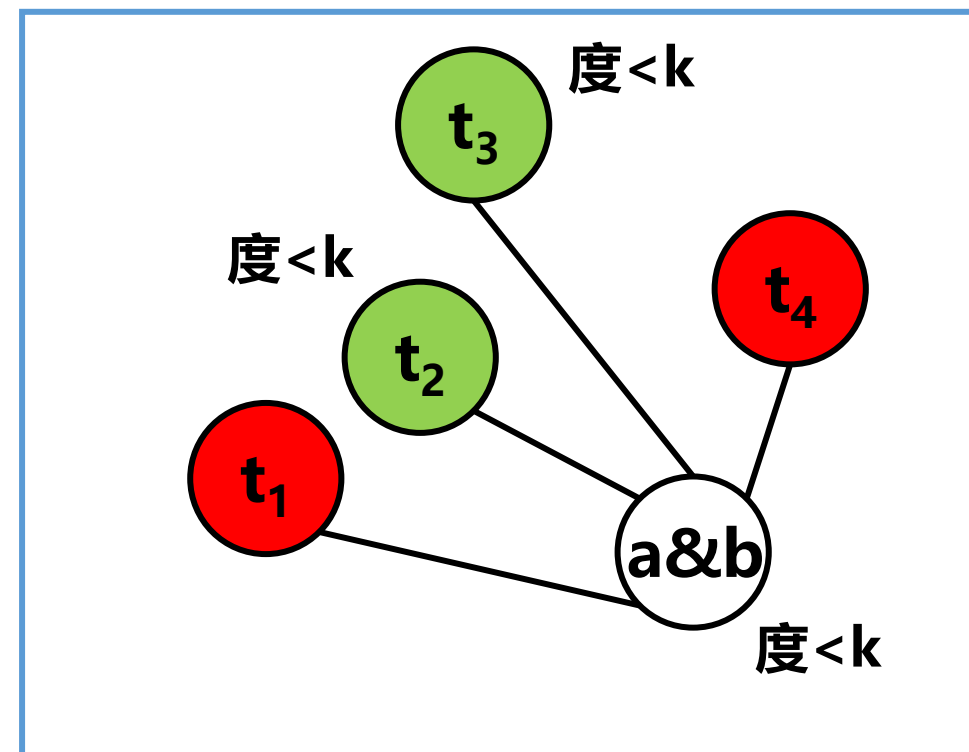
■保守合并：限定结点a和b可以被合并的条件

- ① 结点a和b传送相关，且无冲突边相连 (Chaitin合并条件)
- ② 合并后的新结点a&b的高度数(度 $\geq k$)邻居结点数小于k

保守合并不会改变原图色数



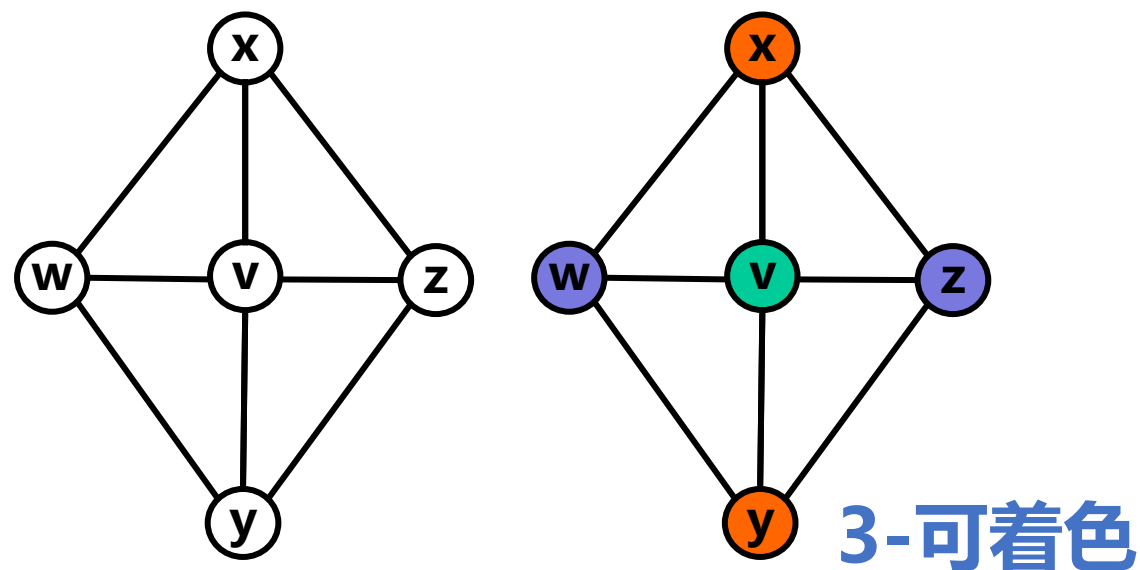
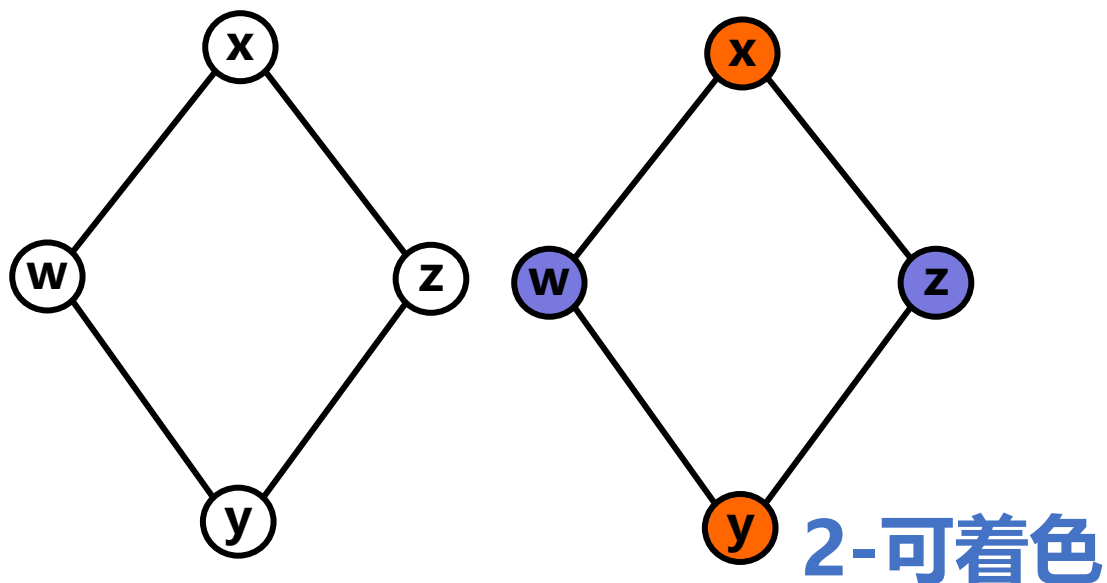
合并
→



红色邻居: $\text{degree} \geq k$
绿色邻居: $\text{degree} < k$

合并后的新结点 $a\&b$ 的高度数
($\text{度} \geq k$) 邻居结点数小于 k

■ 如下冲突图最少需用几种颜色着色？



■ 基于度 $< k$ 定理的化简着色方法不适用

⊕ 一个结点的两个邻居结点可能使用相同的颜色

■ 乐观着色

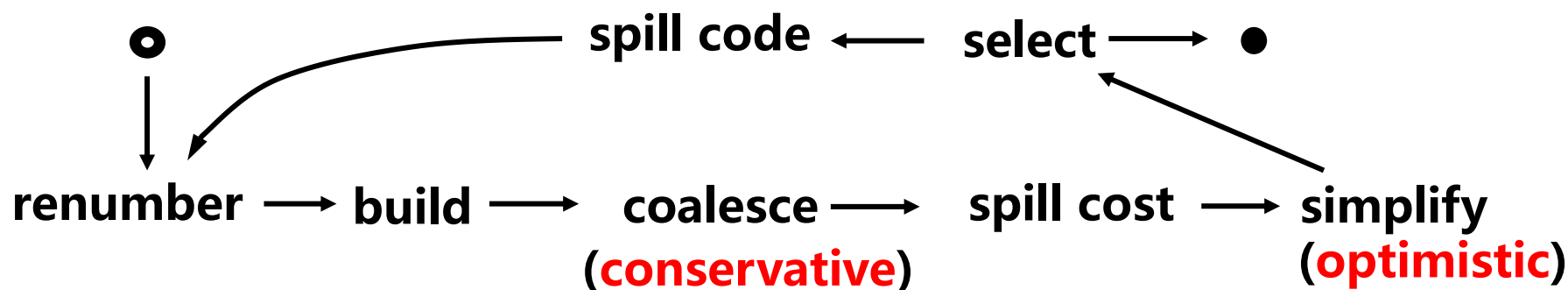
- ⊕ 当化简到不再有低度数结点时，选择一个溢出代价最小的结点
- ⊕ 乐观地将其压栈，并继续化简
- ⊕ 选择阶段，如果不能为其指派颜色，才发生实际溢出

■保守合并：对合并增加额外的限制条件

⊕ 合并不会改变原图色数，因此不会增加溢出

■乐观着色：将溢出推迟到选择阶段

⊕ 可以减少溢出，从而减少重构冲突图、合并、化简等迭代过程



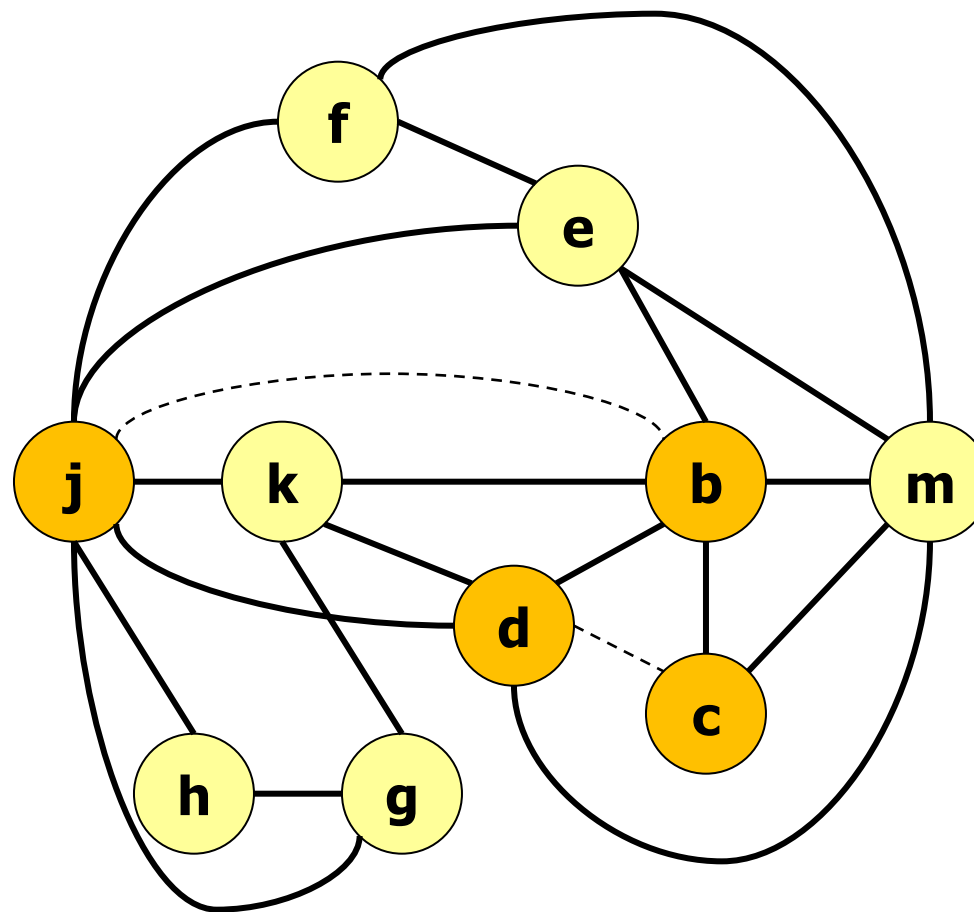
■比Chaitin算法更高效, 有效减少溢出的发生, 降低迭代代价

- 回顾前例，运用Briggs改进算法进行寄存器分配，假设寄存器数为4 ($k=4$)

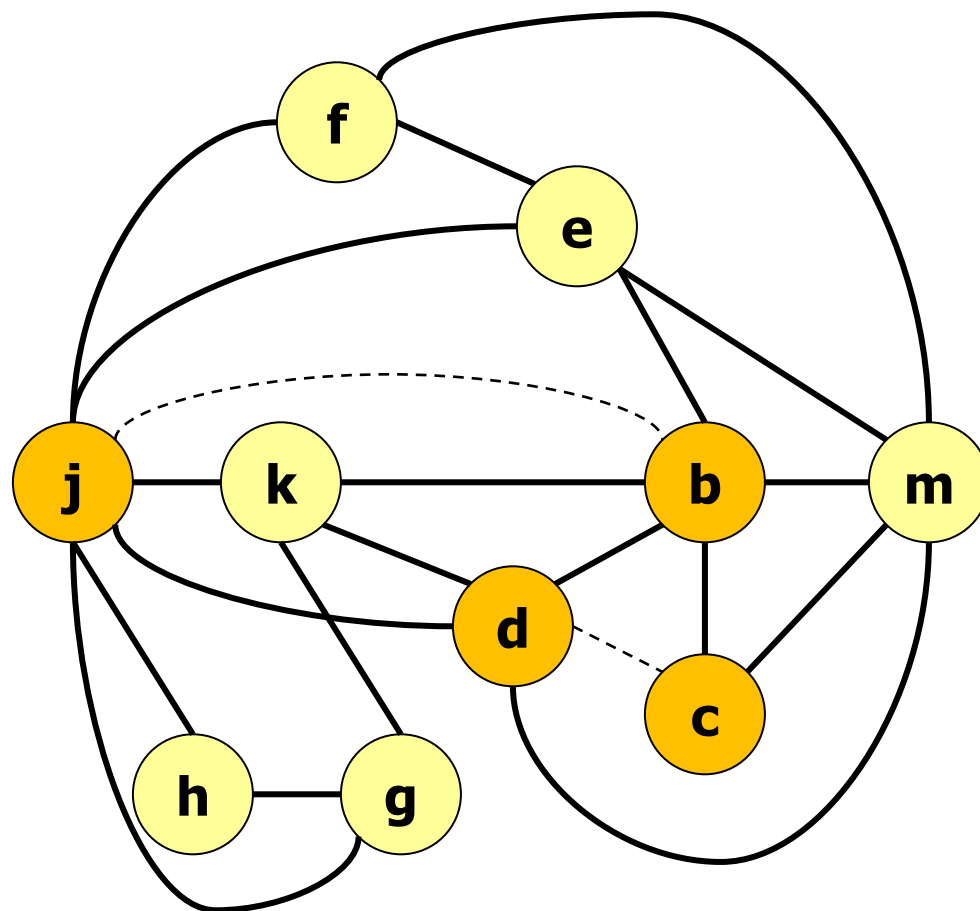
LIVE-IN: k j

```
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

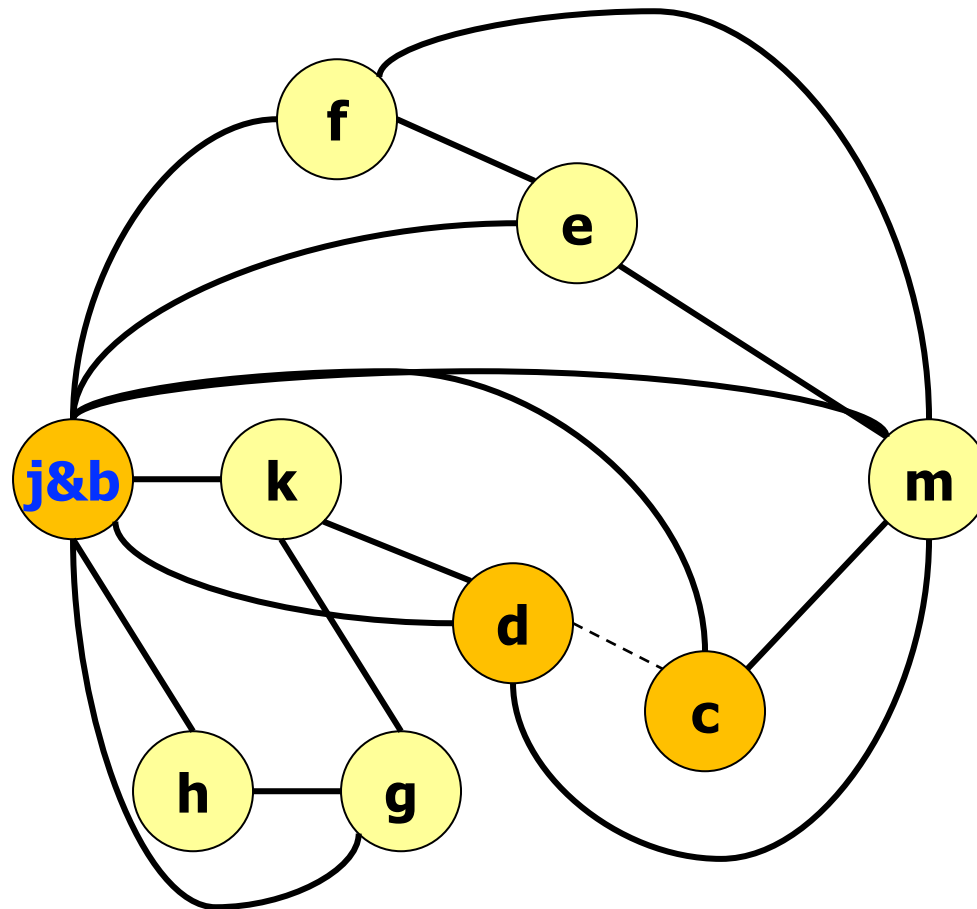


■ ①构建冲突图

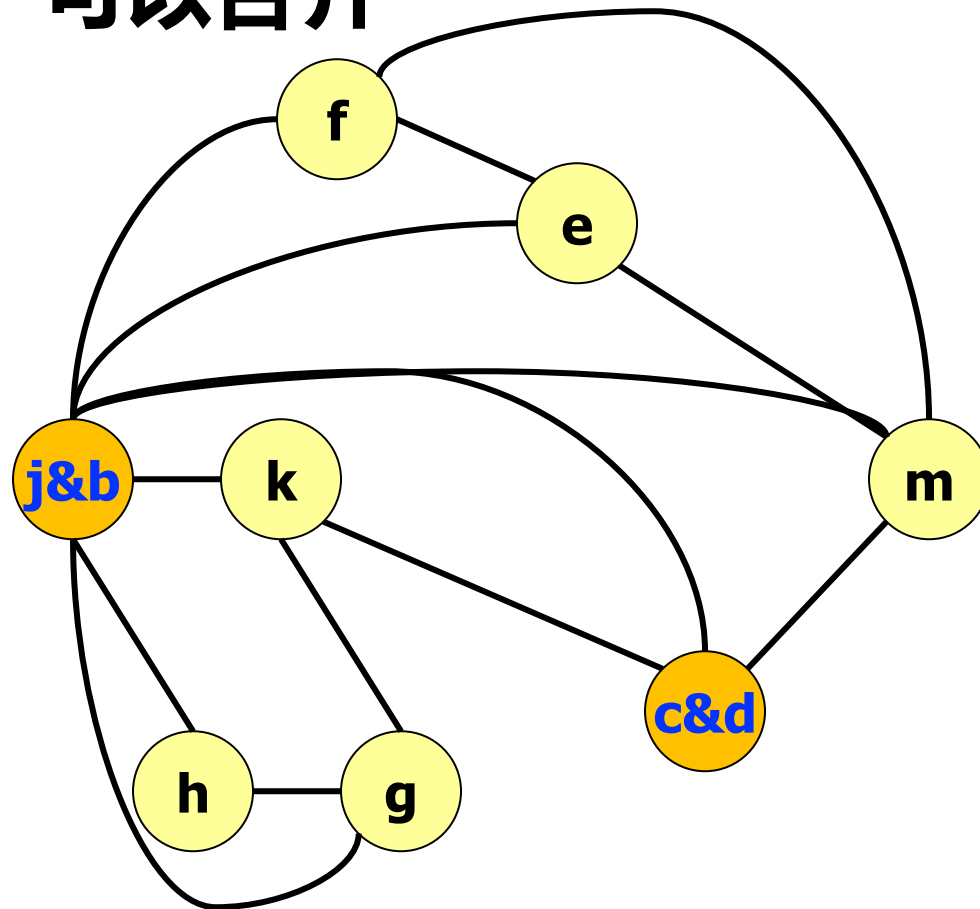


传送相关: j-b, c-d

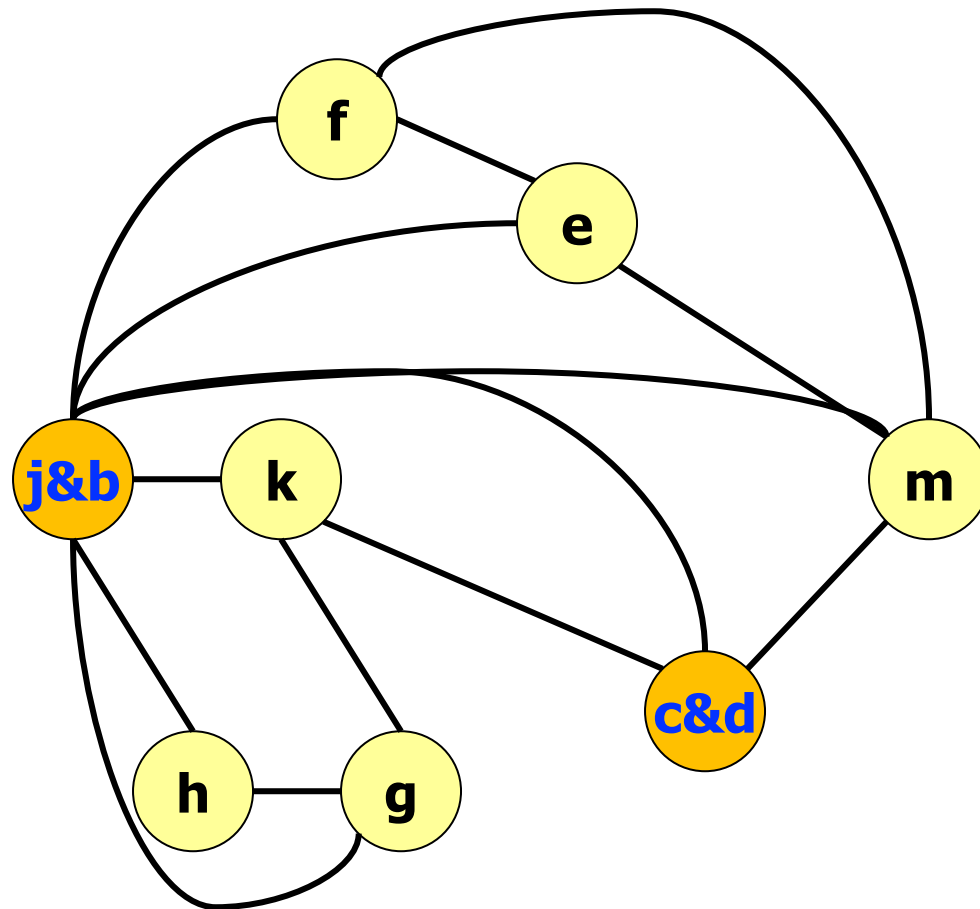
- ②合并：合并j-b后只有1个高度数邻居结点m，数目 <4 ，可以合并



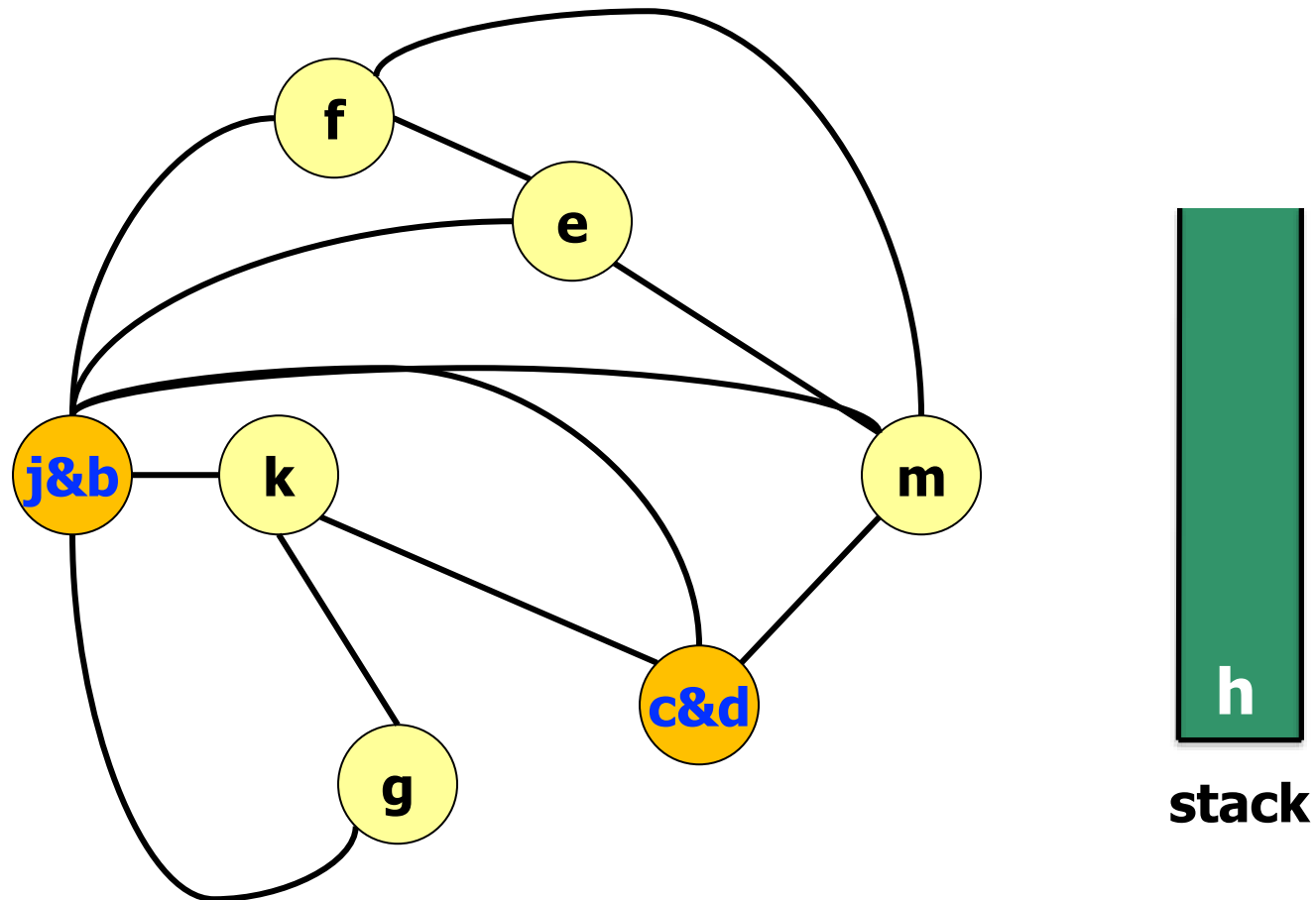
- ②合并：合并c-d后，只有2个高度数邻居结点m、j&b，数目小于4，可以合并



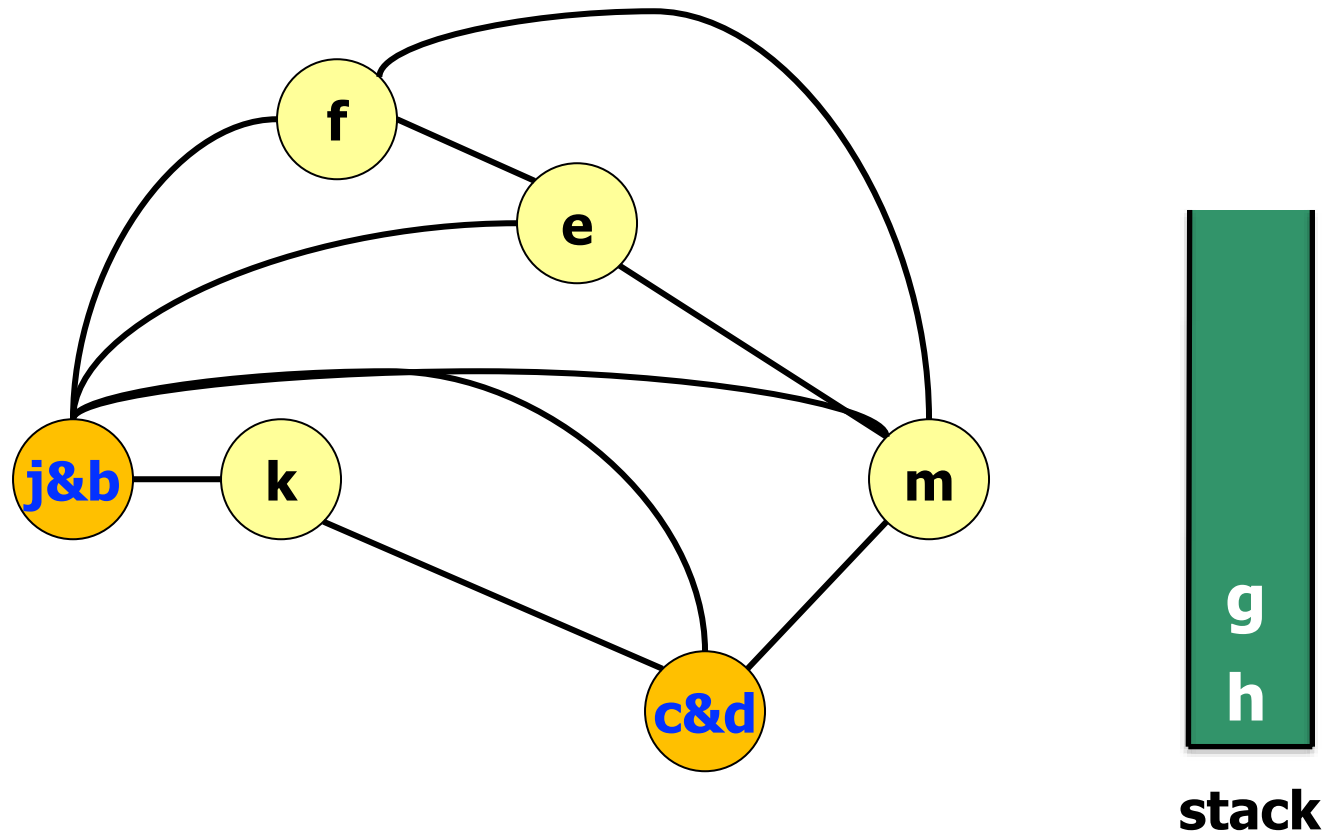
- ③评估溢出代价：如需溢出，溢出度数最高的结点j&b



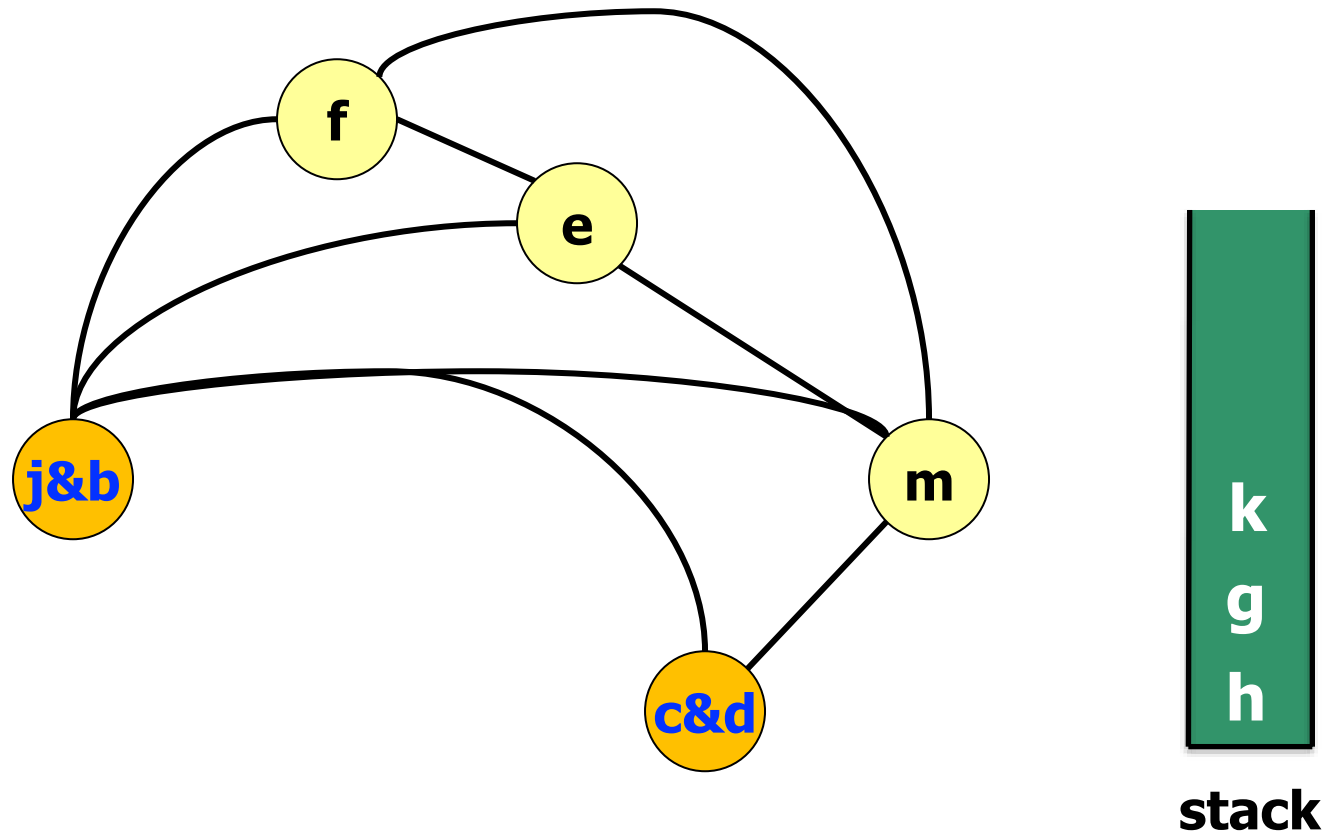
■④化简：删除度小于4的结点，压栈



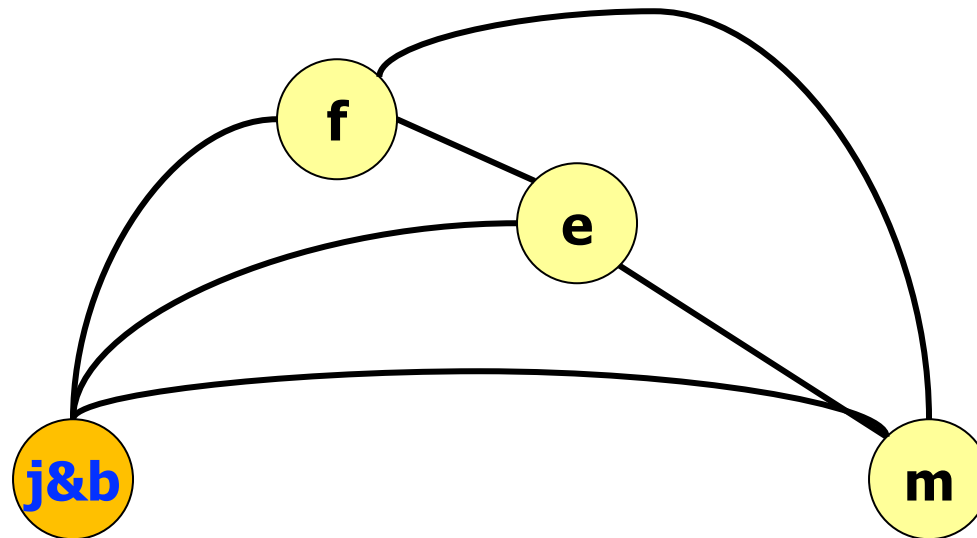
■④化简：删除度小于4的结点，压栈



■④化简：删除度小于4的结点，压栈



■④化简：删除度小于4的结点，压栈



剩余结点度均小于4，可以全部压栈

c&d
k
g
h

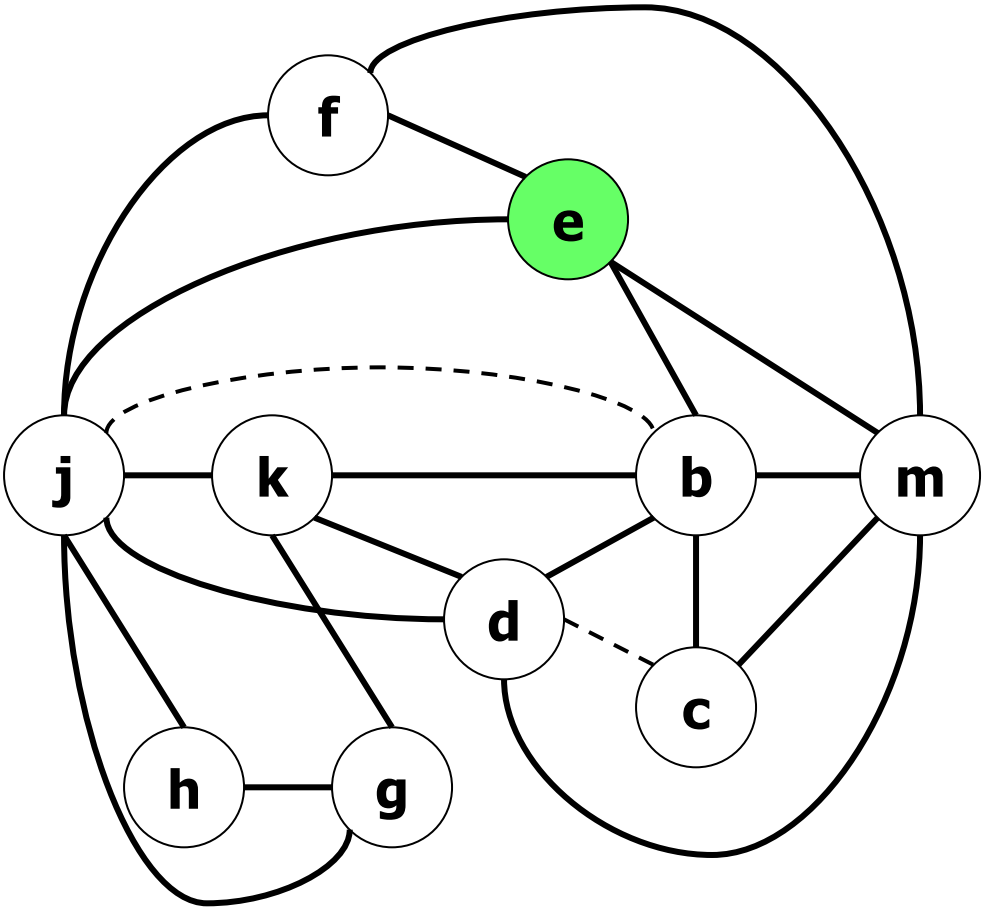
stack

■④化简：删除度小于4的结点，压栈

e
m
f
j&b
c&d
k
g
h

stack

■⑤选择：将结点依次弹栈，并指派颜色



e
m
f
j&b
c&d
k
g
h

stack

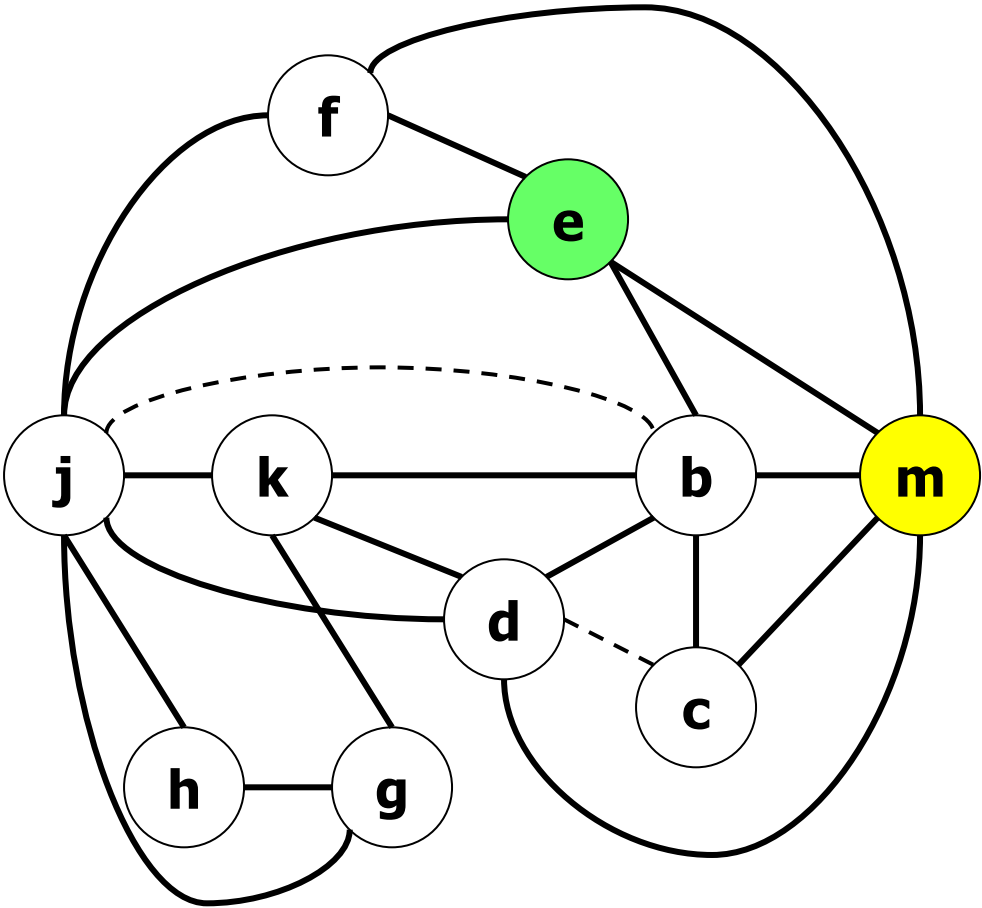
R1

R2

R3

R4

■⑤选择：依次弹栈，指派颜色



m
f
j&b
c&d
k
g
h

stack

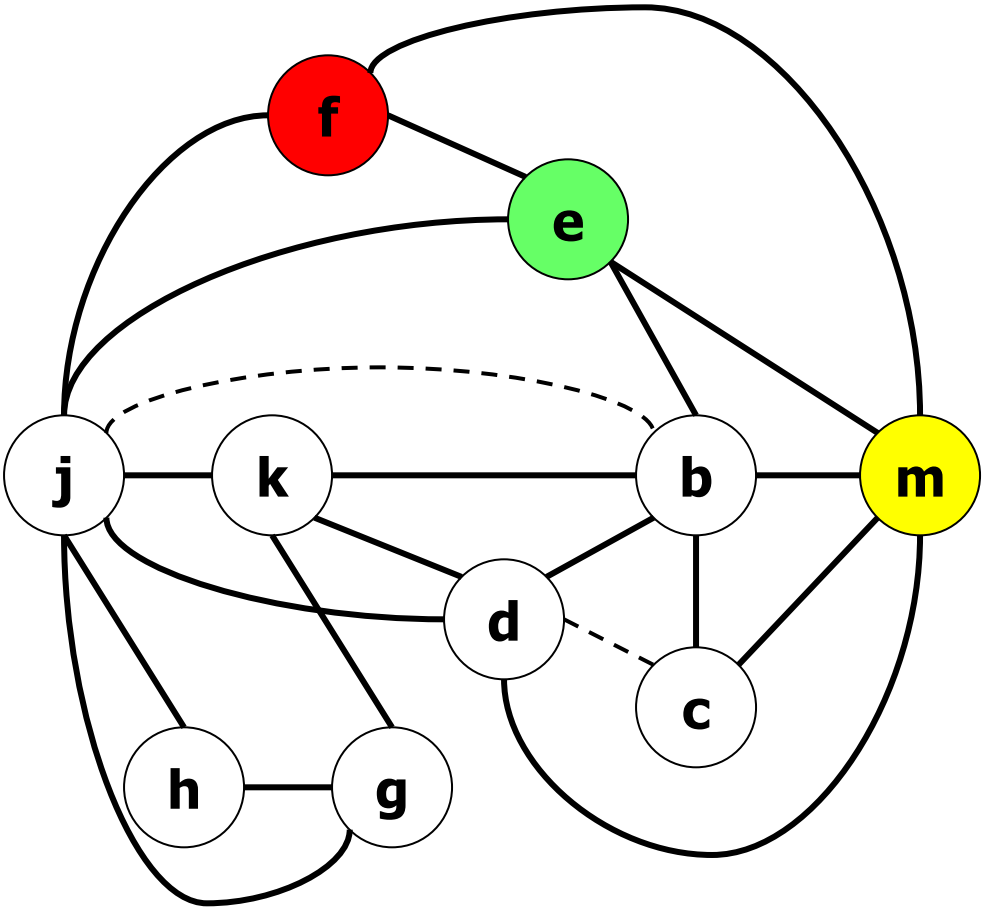
R1

R2

R3

R4

■⑤选择：依次弹栈，指派颜色



f
j&b
c&d
k
g
h

stack

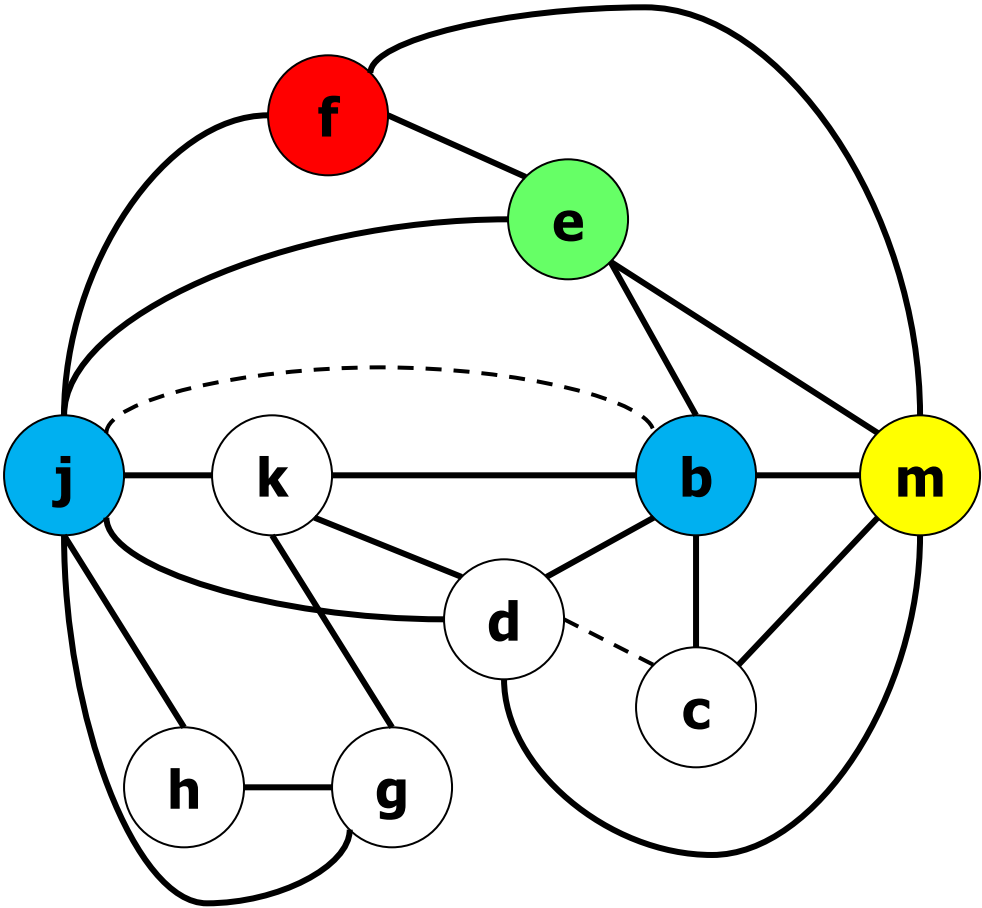
R1

R2

R3

R4

■⑤选择：依次弹栈，指派颜色



j&b
c&d
k
g
h

stack

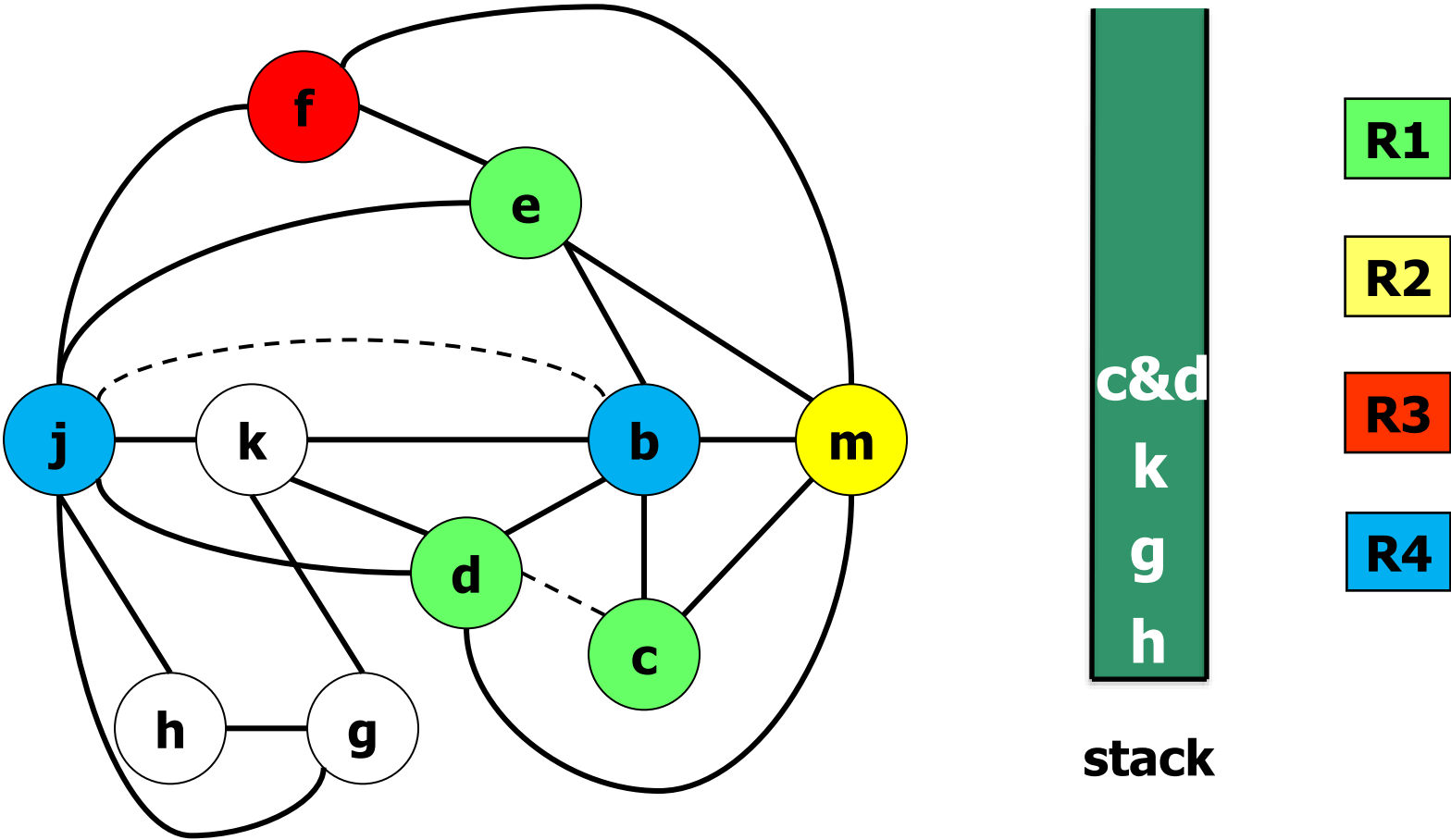
R1

R2

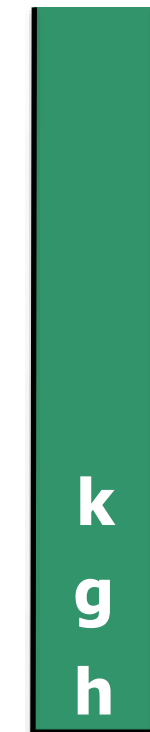
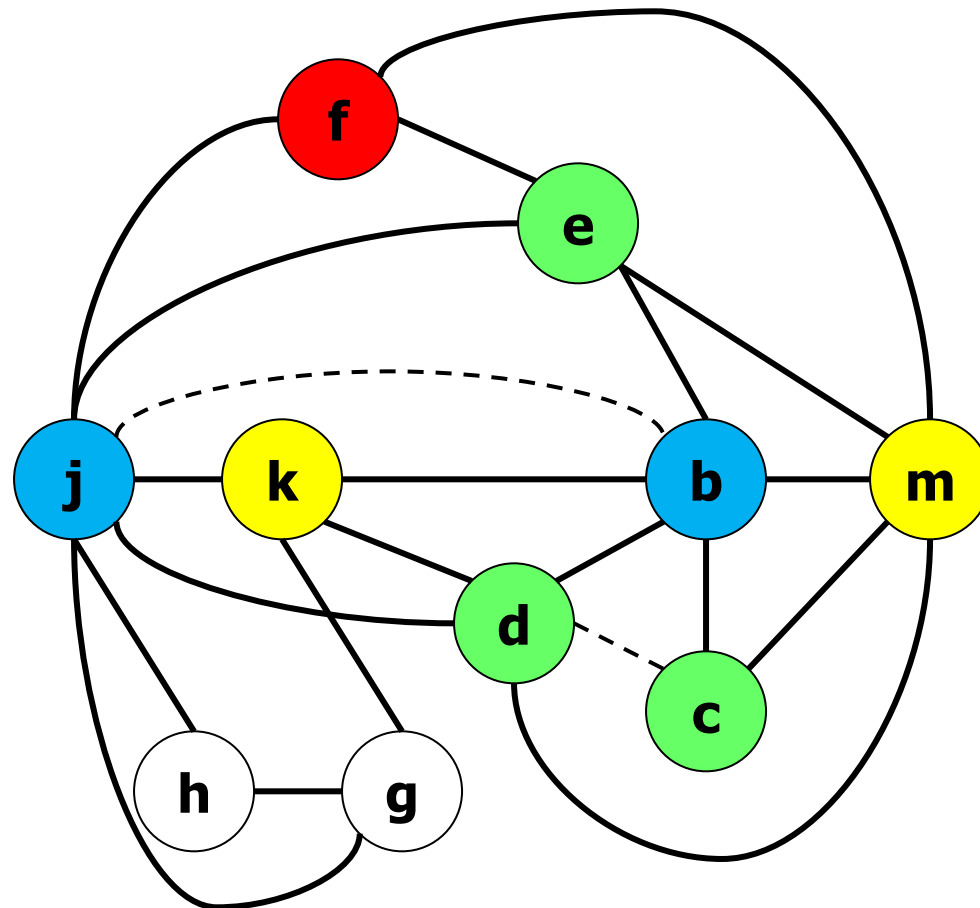
R3

R4

■⑤选择：依次弹栈，指派颜色



⑤选择：依次弹栈，指派颜色



stack

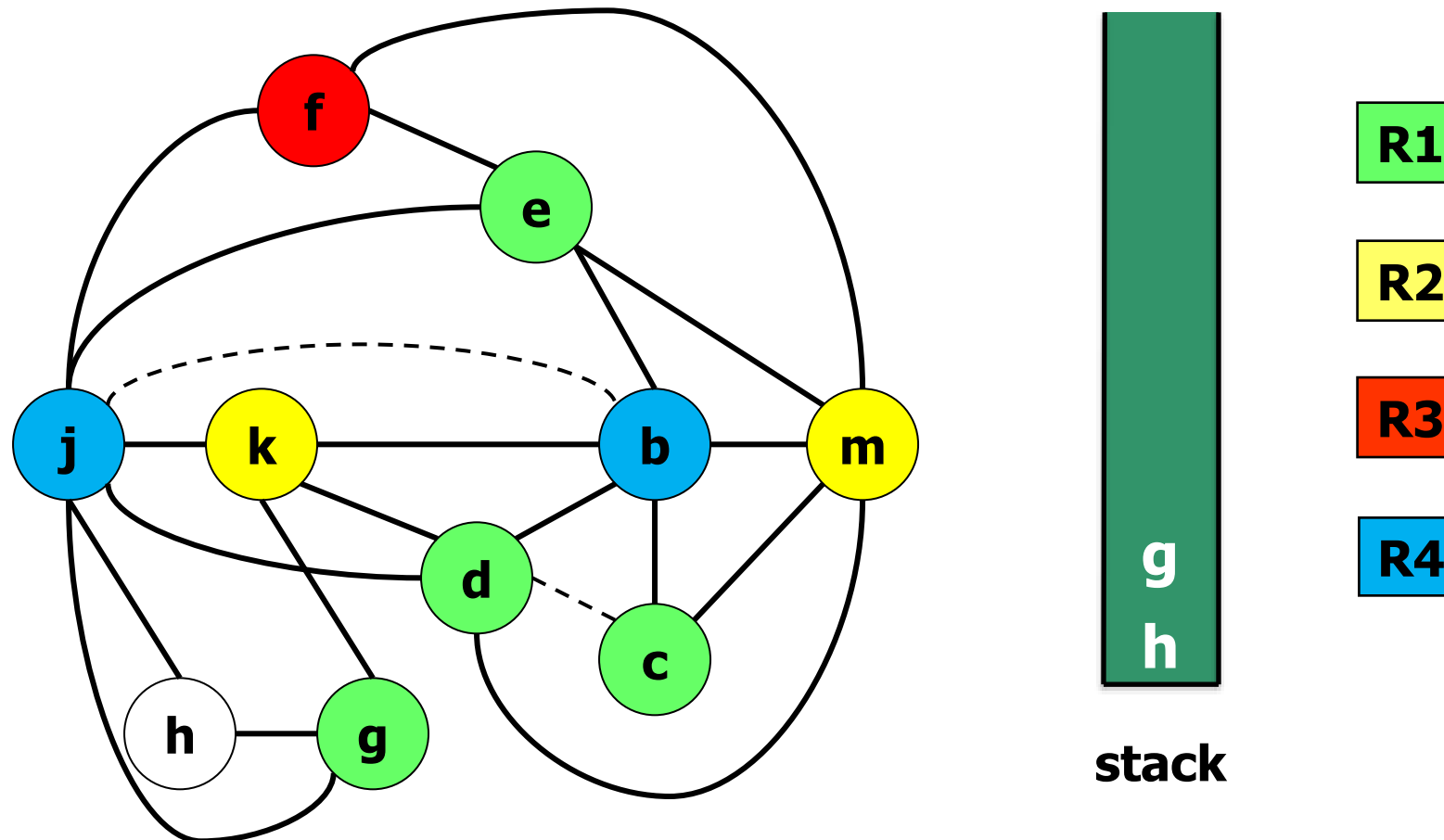
R1

R2

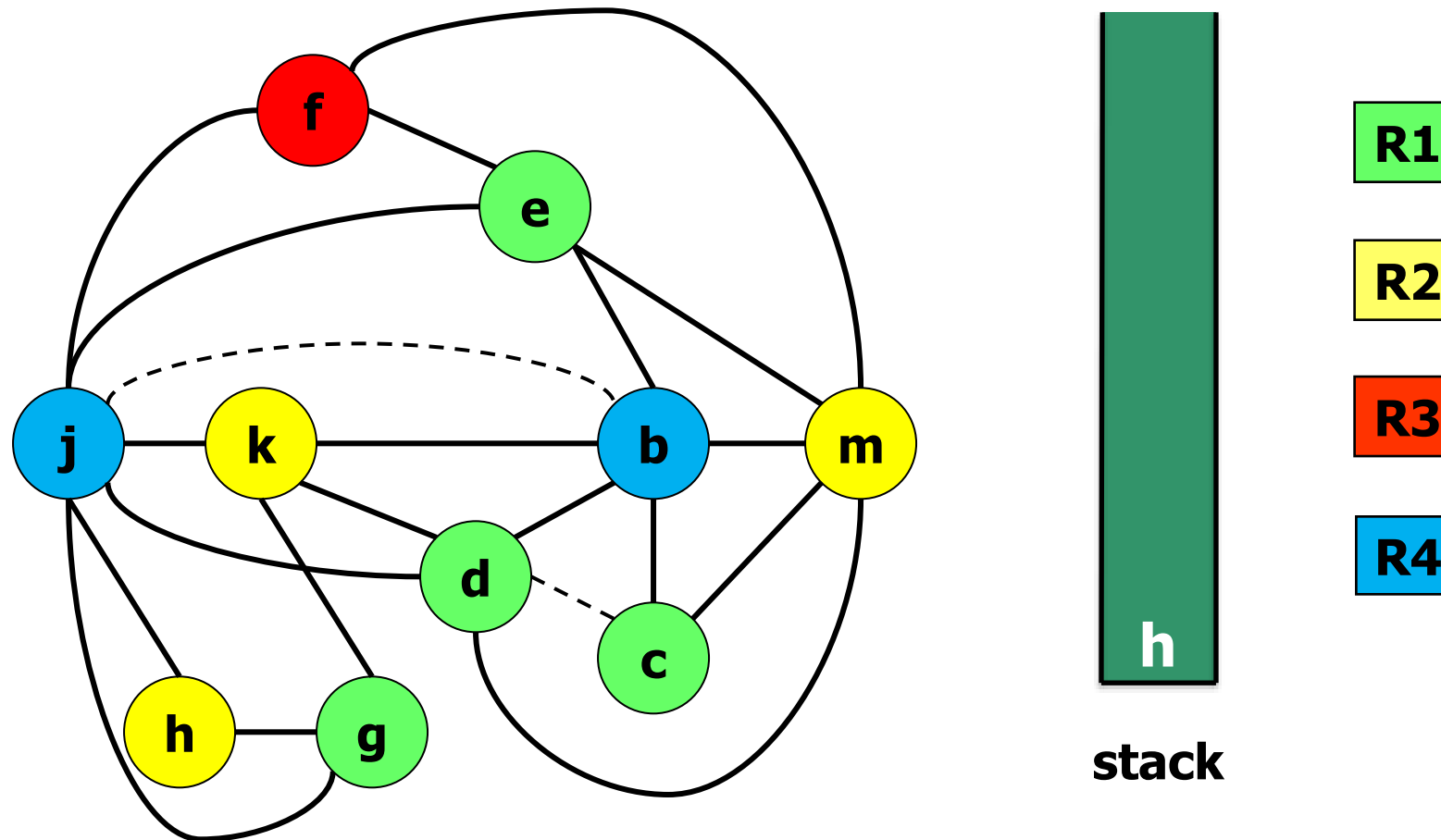
R3

R4

■ ⑤选择：依次弹栈，指派颜色



■ ⑤选择：依次弹栈，指派颜色



没有出现溢出

10.1 寄存器分配概述

10.2 基于使用计数的寄存器分配方法

10.3 基于图着色的寄存器分配方法

10.4 Briggs图着色寄存器分配改进算法

10.5 George图着色寄存器分配改进算法

10.6 基于线性扫描的寄存器分配方法

■ Briggs改进算法过于保守

- ⊕ 错失一些可以合并的结点以及可以删除的传送指令

■ 针对Briggs算法，George提出两项改进

- ⊕ 放松合并条件
- ⊕ 引入迭代的合并过程和冻结结点

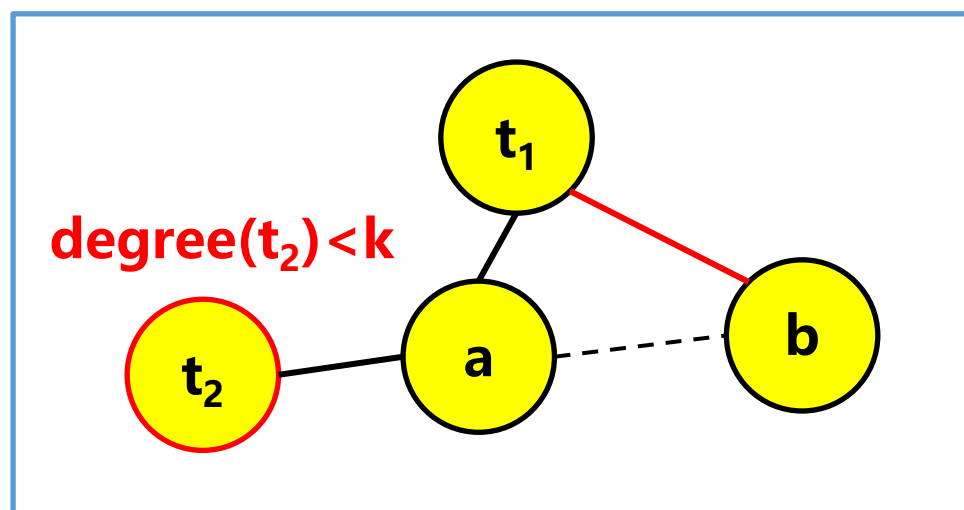
■ George合并条件

合并后的新结点a&b的高度数(度 $\geq k$)邻居结点数小于k (Briggs)

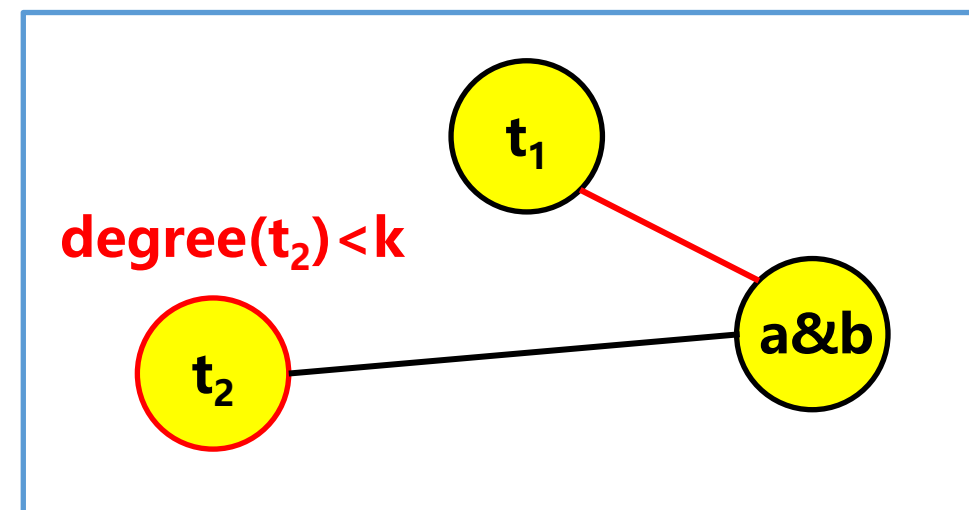


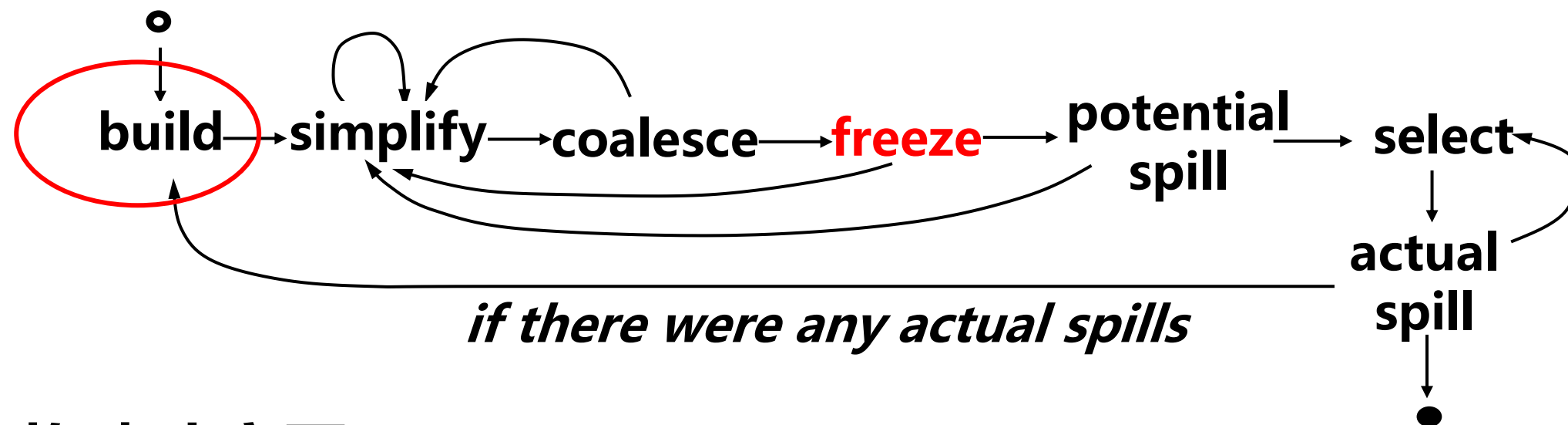
对a的每个邻居结点t, 要么t是b的邻居, 要么t是一个低度数(度 $< k$)结点, 则可以合并 (George)

合并不会改变原图色数



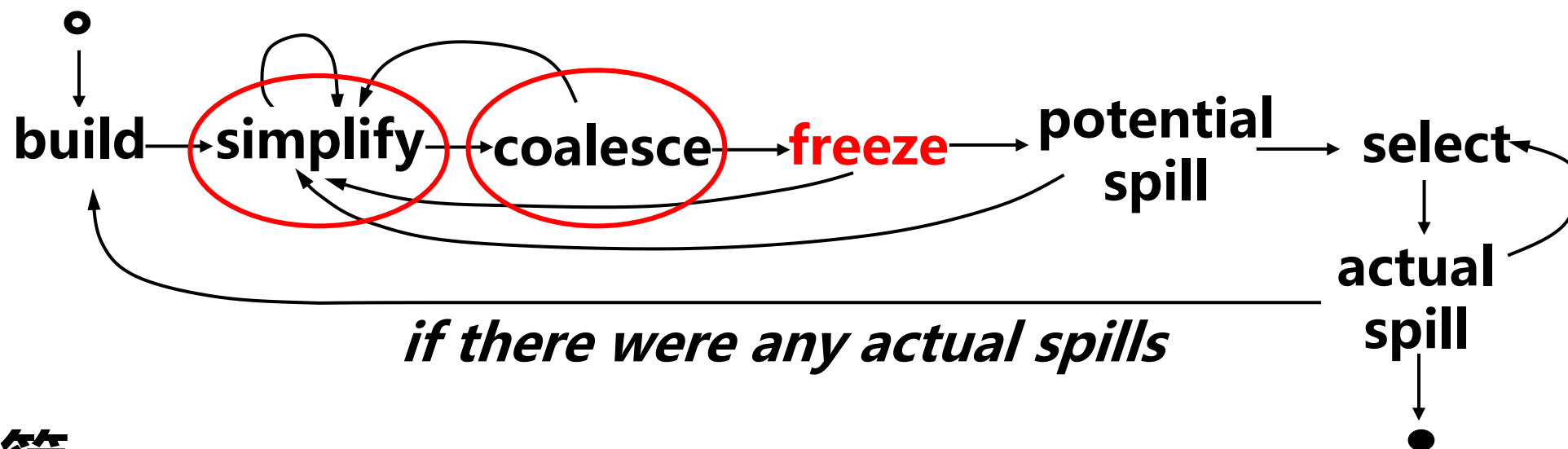
合并
→





①构建冲突图

⊕构建冲突图，将结点分为**传送相关**和**传送无关**结点



②化简

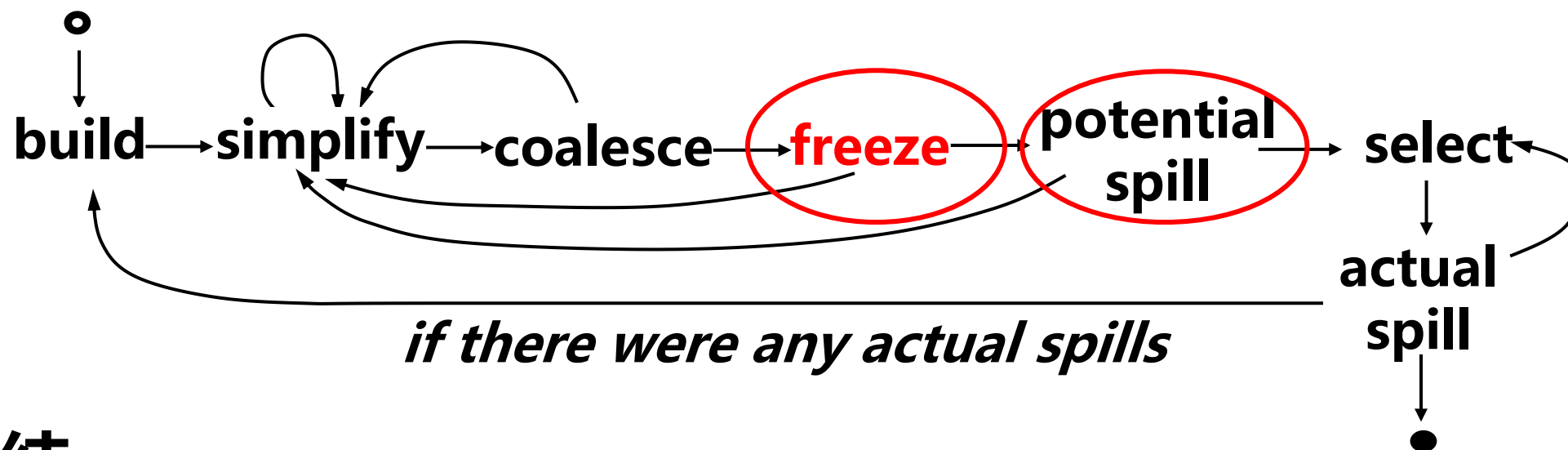
⊕ 删除与**传送无关**的**低度数**结点(**度** $< k$), 直到不能进行化简

③合并

⊕ 根据George合并条件, 保守合并**传送相关**的结点

⊕ 由此产生的结点如果不再是传送相关的, 则用于下一轮化简

⊕ 重复化简和合并过程, 直到只剩下**高度数**(**度** $\geq k$)结点, 或者**传送相关**但不满足合并条件的结点

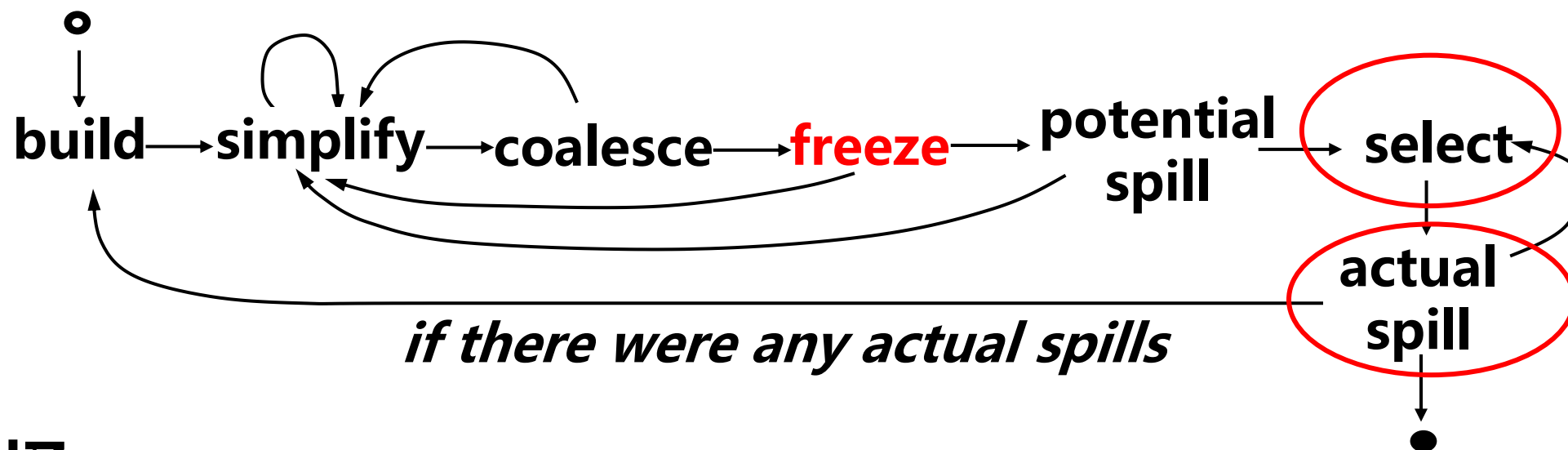


④冻结

✦ 如果化简和合并都不能再进行，则冻结一对**传送相关**的结点，将它们看作**传送无关**的结点，继续化简、合并

⑤ 潜在溢出

- 如果没有低度数结点，则根据启发式方法评估溢出代价，选择一个溢出代价小的结点作为潜在溢出结点，并将其压入栈
- 继续化简、合并、冻结的过程，直到所有结点入栈



⑥选择

⊕将结点依次弹栈，并指派颜色

⑦真正溢出

⊕当指派到一个潜在溢出结点时，如果不能为其指派颜色，则发生**真正溢出**

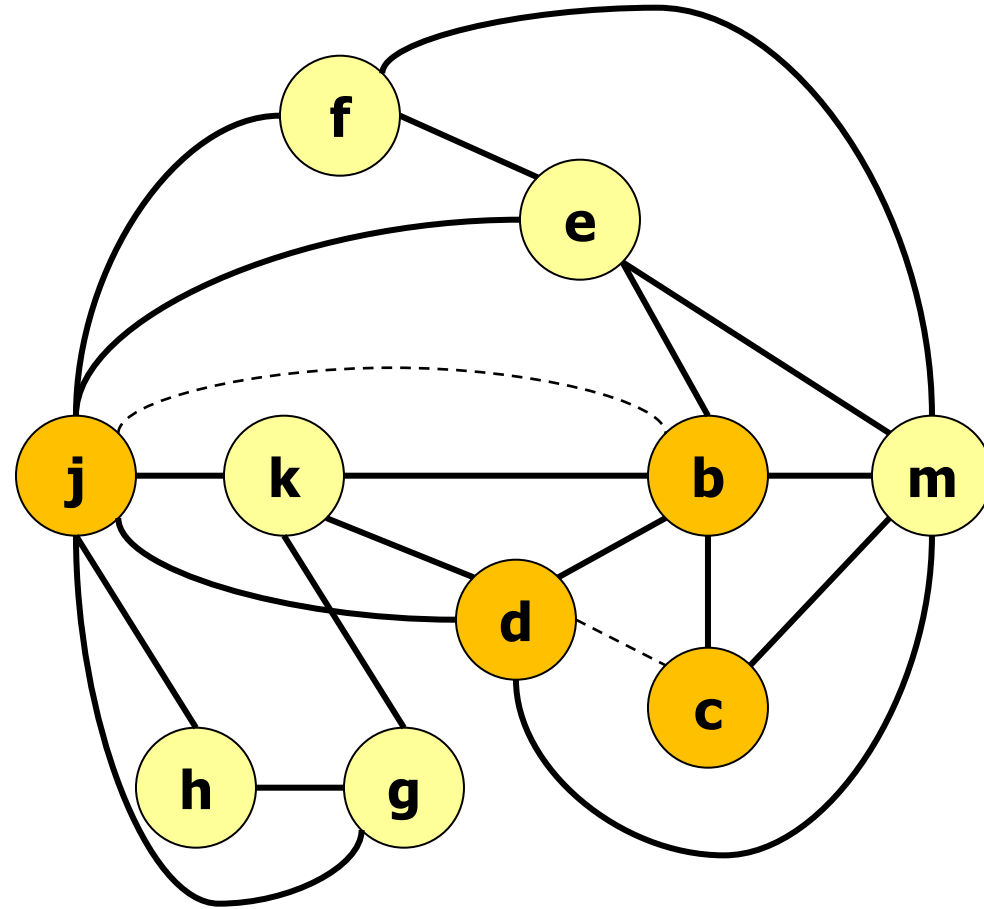
⊕重构冲突图，重新化简和合并

- 回顾前例，运用George改进算法进行寄存器分配，假设寄存器数为3 ($k=3$)

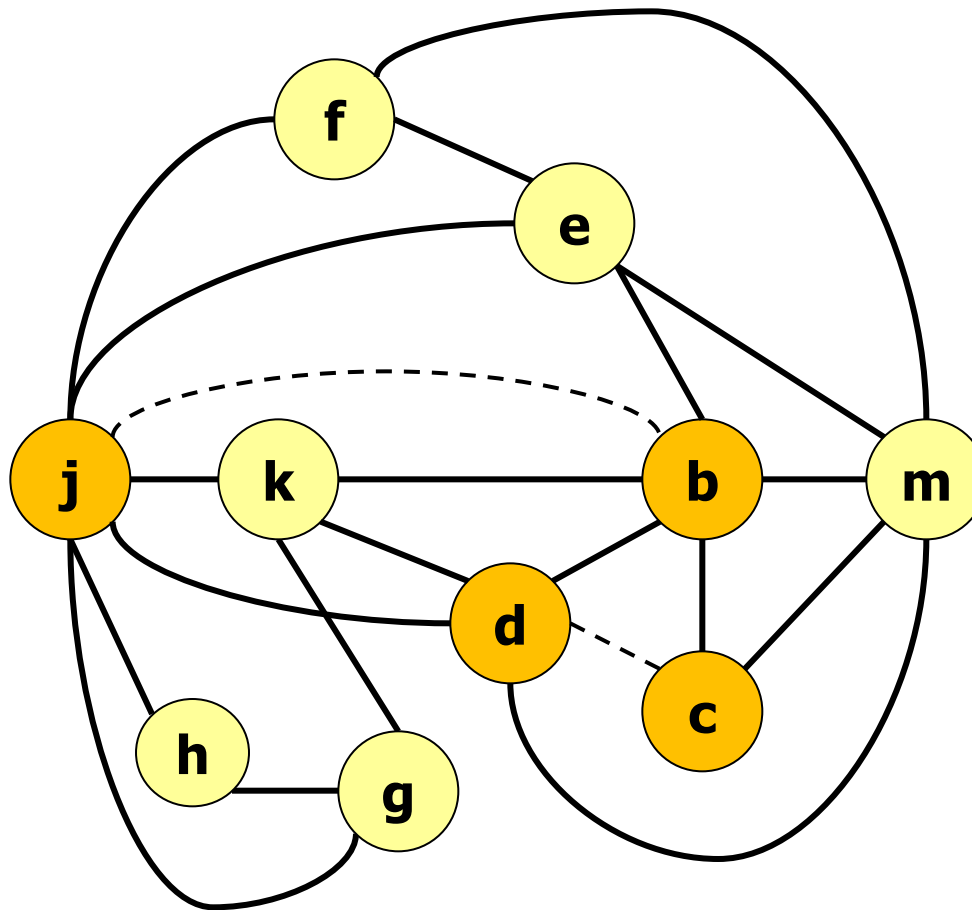
LIVE-IN: k j

```
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

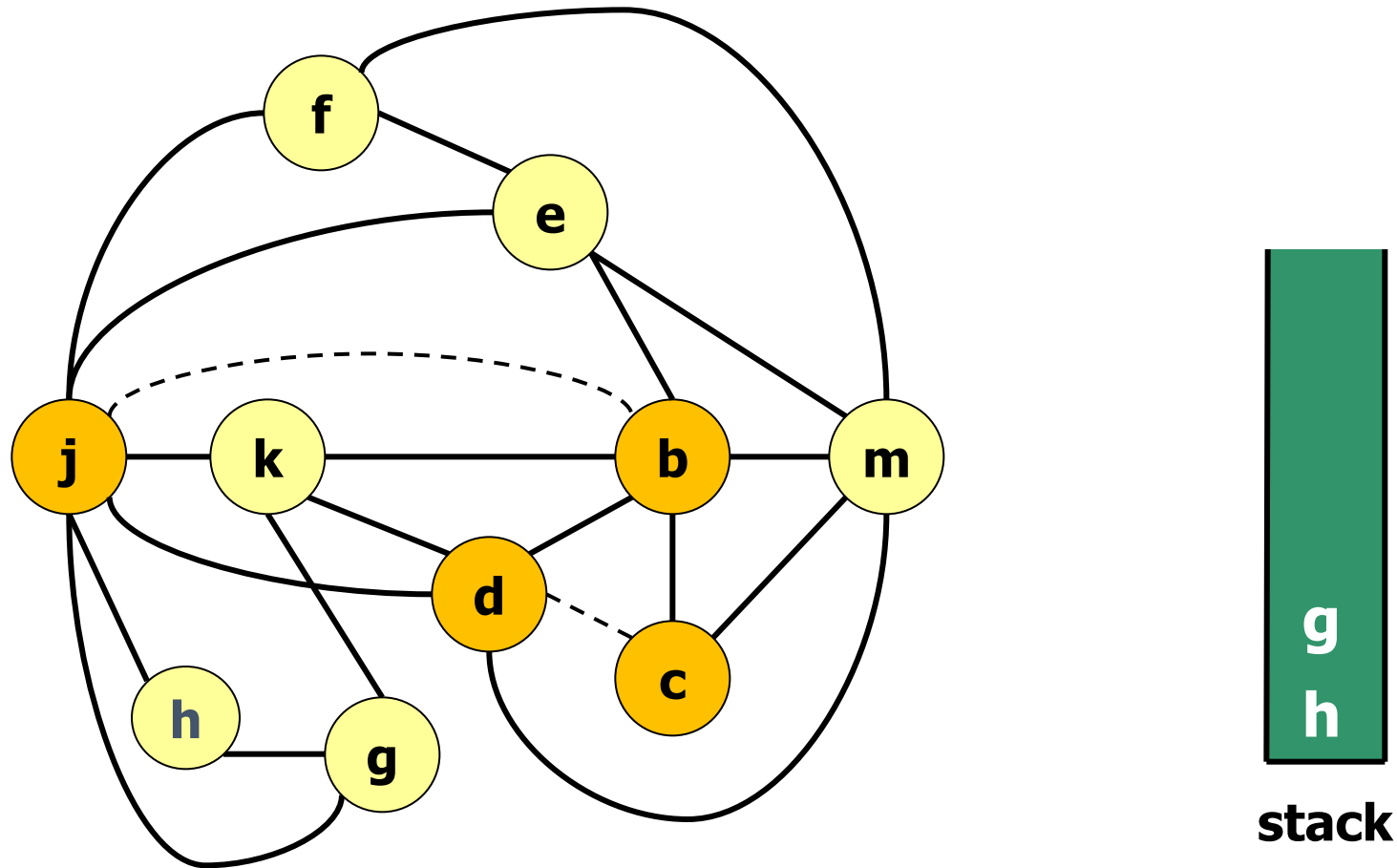


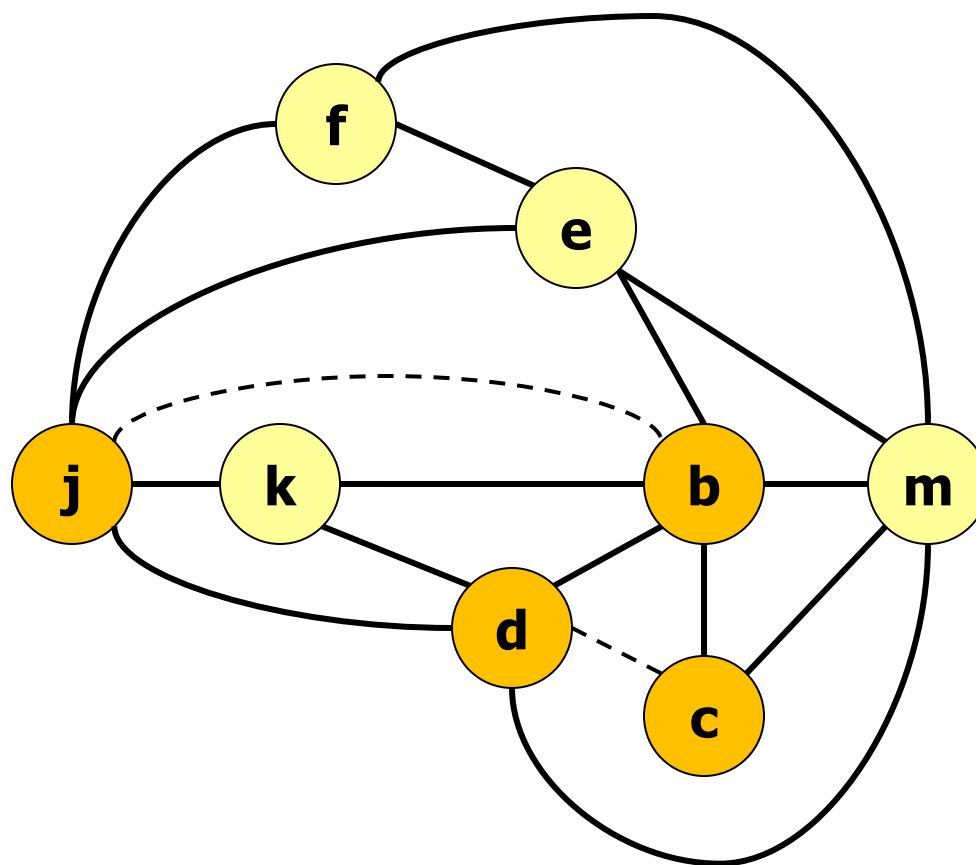
■ ①构建冲突图



传送相关: j-b, c-d

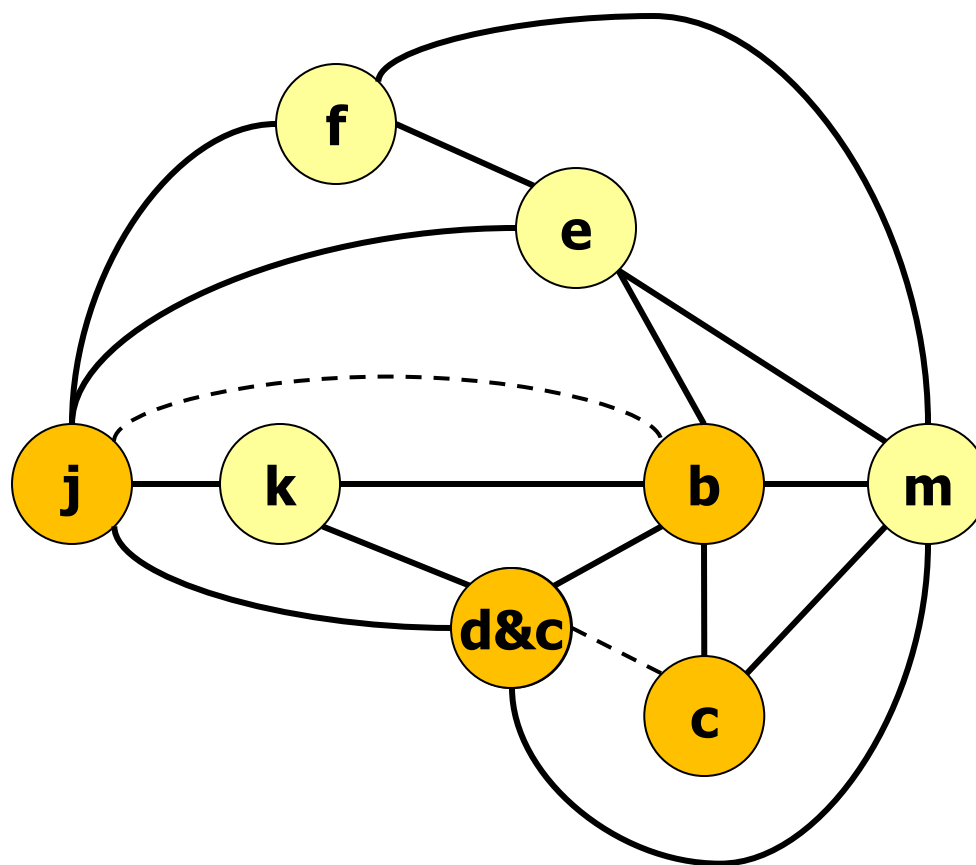
■ ②化简: 删除与**传送无关**的低度数结点(**度k**) , 压栈





b的邻居结点m不是低度数结点，因此j-b不能合并

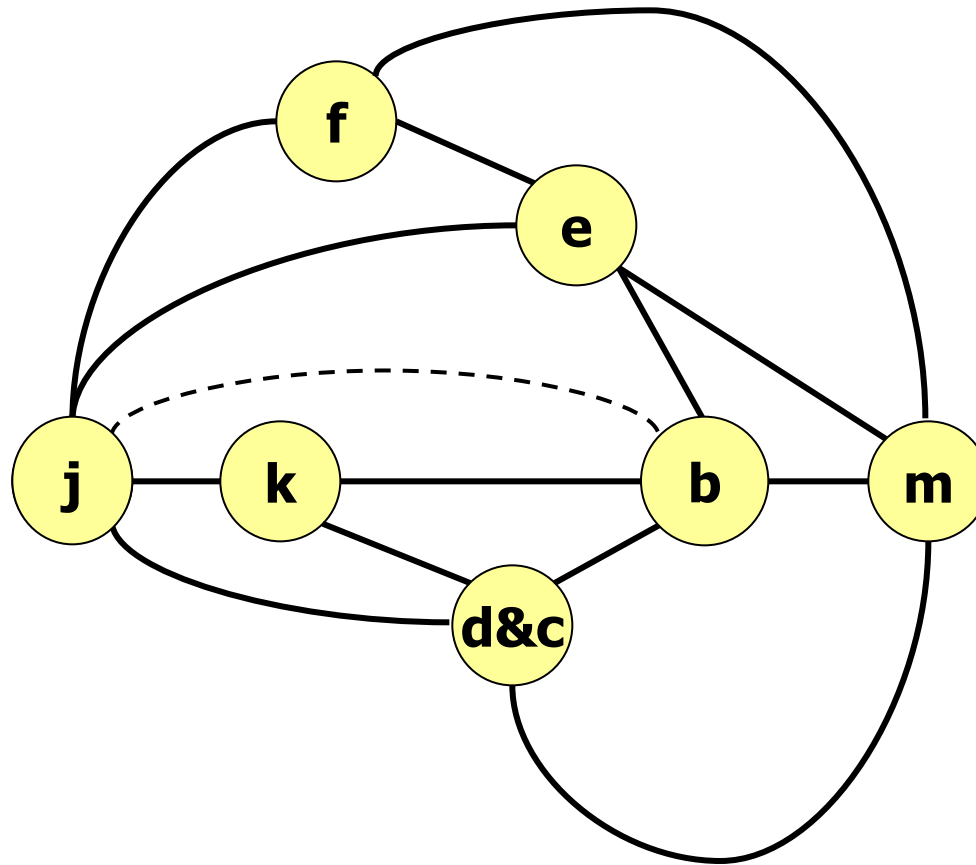
■ ③合并: 根据George合并条件, 保守合并**传送相关**的结点



合并c-d?

c的邻居b和m, 也是d的邻居, 因此c-d可以合并

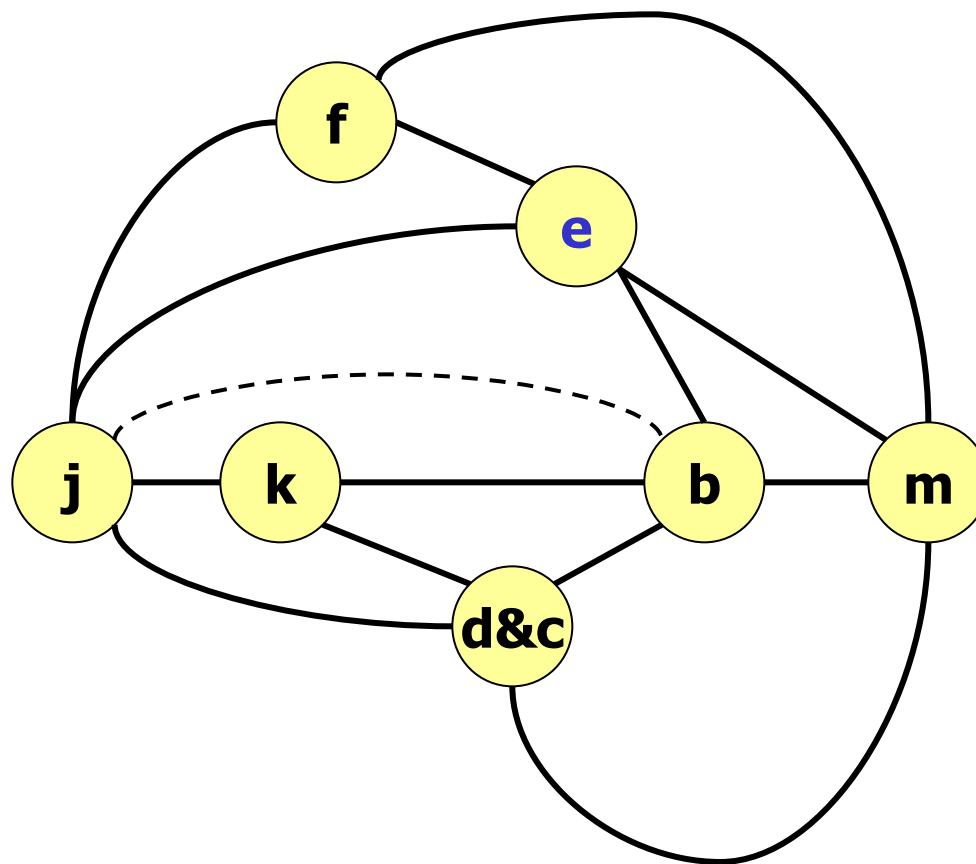
■④冻结一对传递相关的结点



不能继续化简、合并

冻结j-b, 将j和b看作传递无关的结点

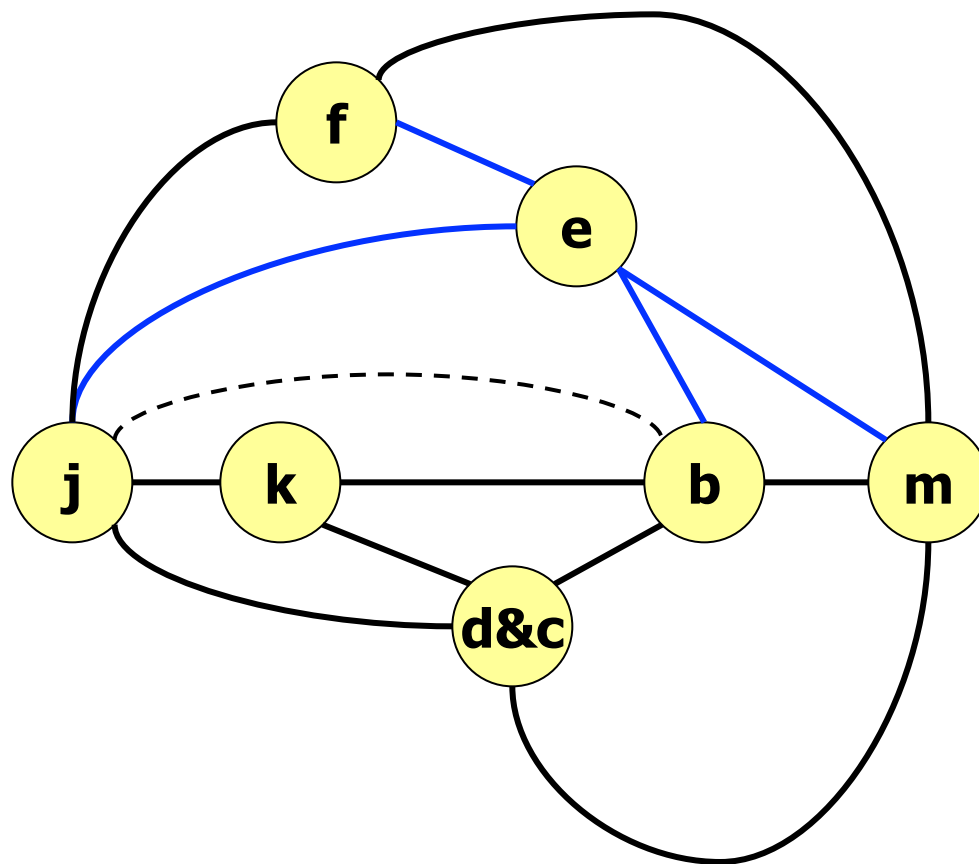
- ⑤潜在溢出: 选在一个结点标记为潜在溢出结点, 将其入栈



合并和冻结都不能使化简继续

使用启发式方法评估溢出代价,
选择e作为潜在溢出结点

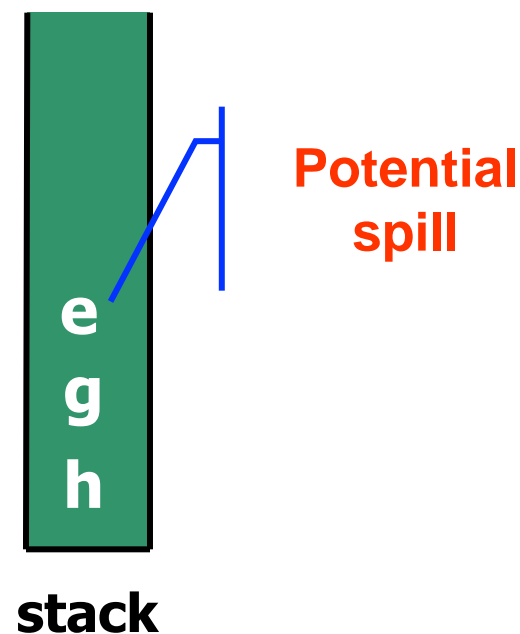
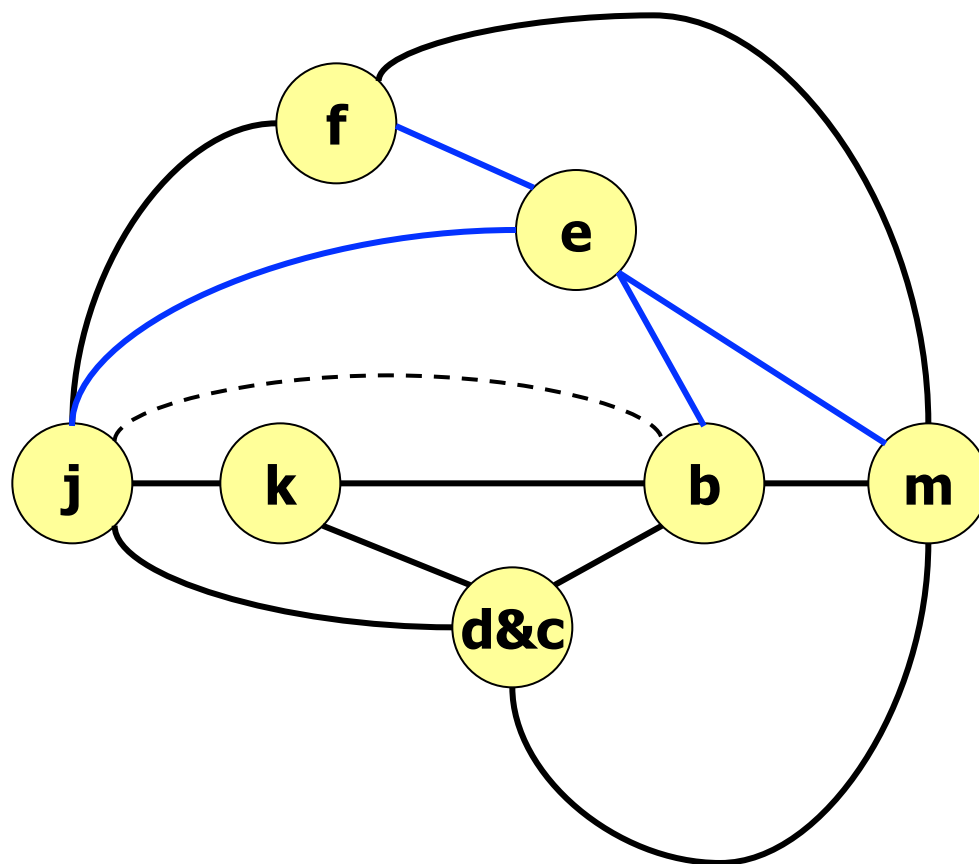
- ⑤潜在溢出: 选在一个结点标记为潜在溢出结点, 将其入栈



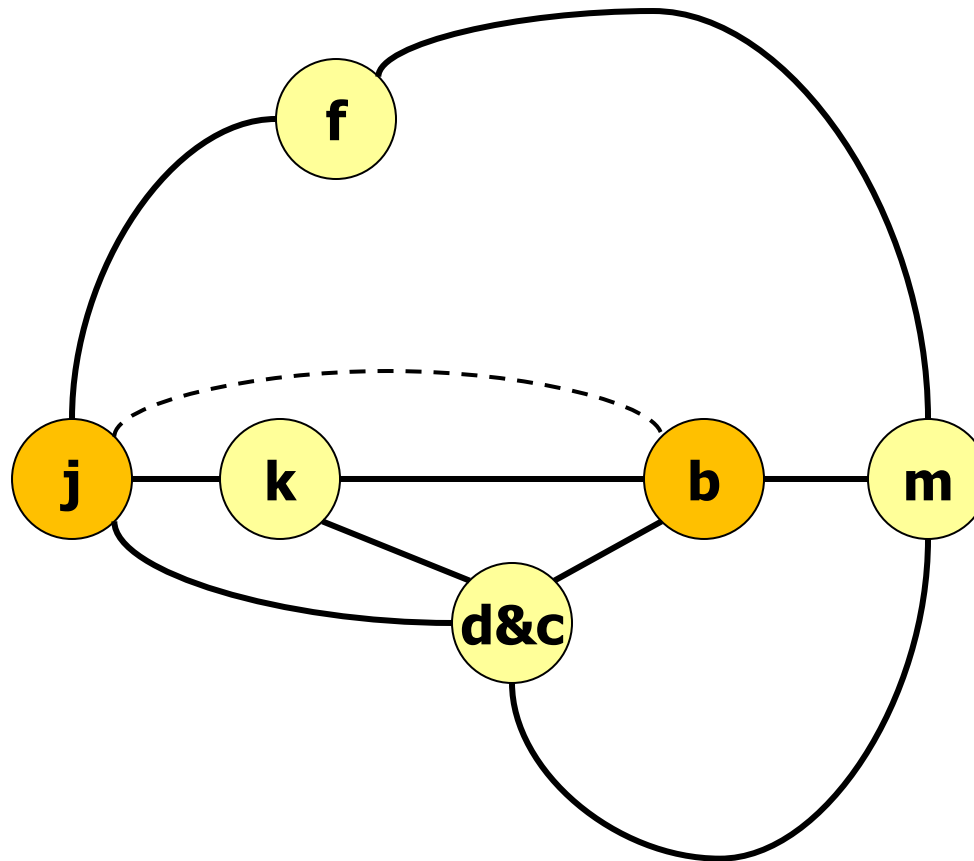
将是否真正需要溢出推迟到
选择阶段判断

如果e的邻居具有相同颜色,
或者有邻居被真正溢出, 则e
可能不需要溢出

- ⑤潜在溢出: 选在一个结点标记为潜在溢出结点, 将其入栈

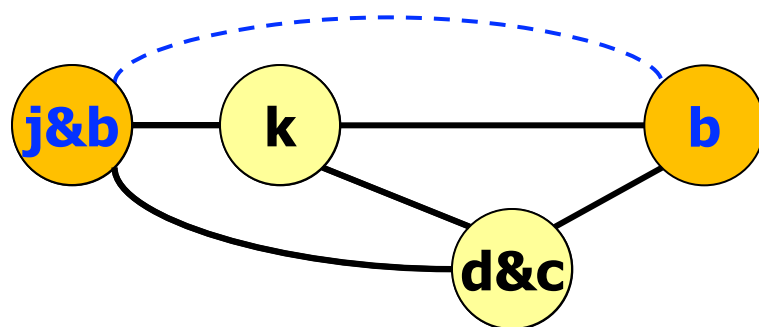


■ ②继续化简: 删除与**传送无关的低度数**结点(**度< k**) , 压栈



stack

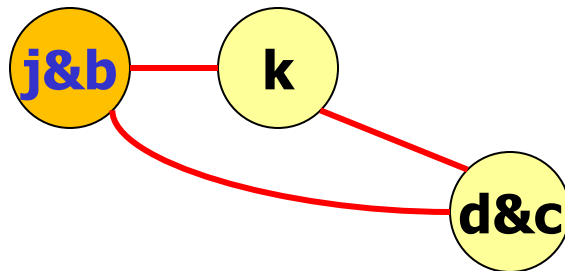
■ ③继续合并: 根据George合并条件, 保守合并**传送相关**的结点



合并 j-b?

b的邻居结点k, d&c也是j的邻居, 因此j-b可以合并

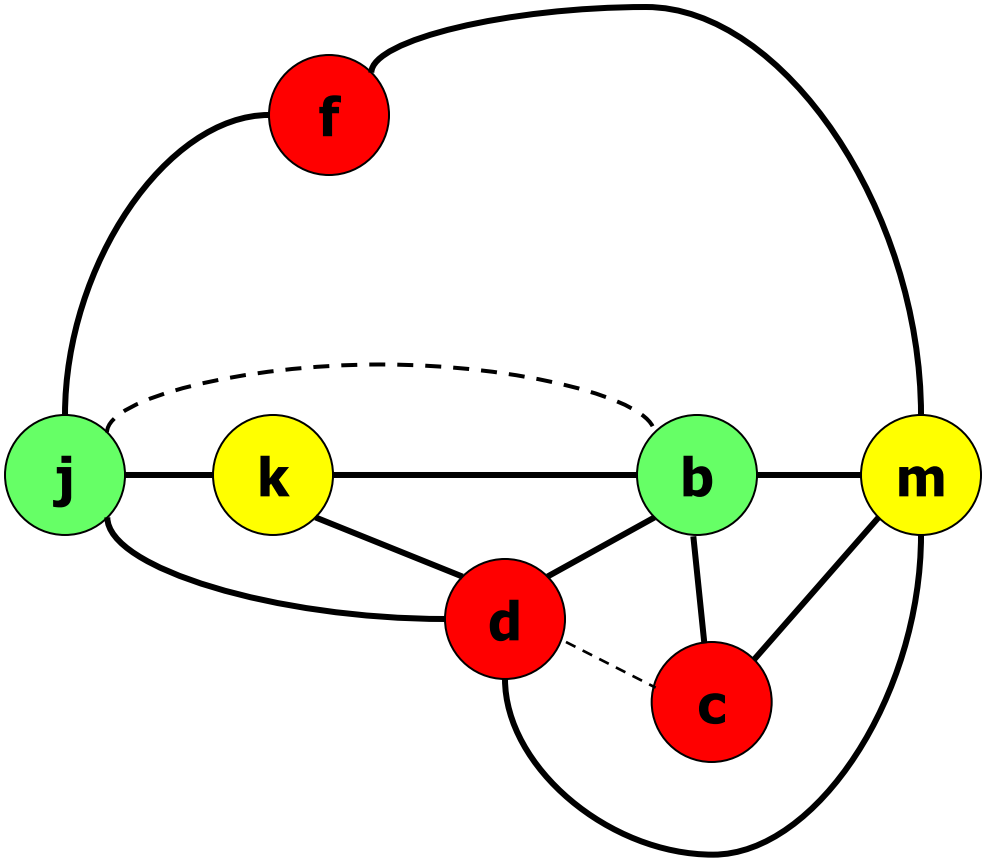
■ ②继续化简: 删除与**传送无关的低度数**结点(**度< k**) , 压栈



j&b
k
d&c
m
f
e
g
h

stack

⑥选择: 将结点依次弹栈, 并指派颜色



j&b
k
d&c
m
f
e
g
h

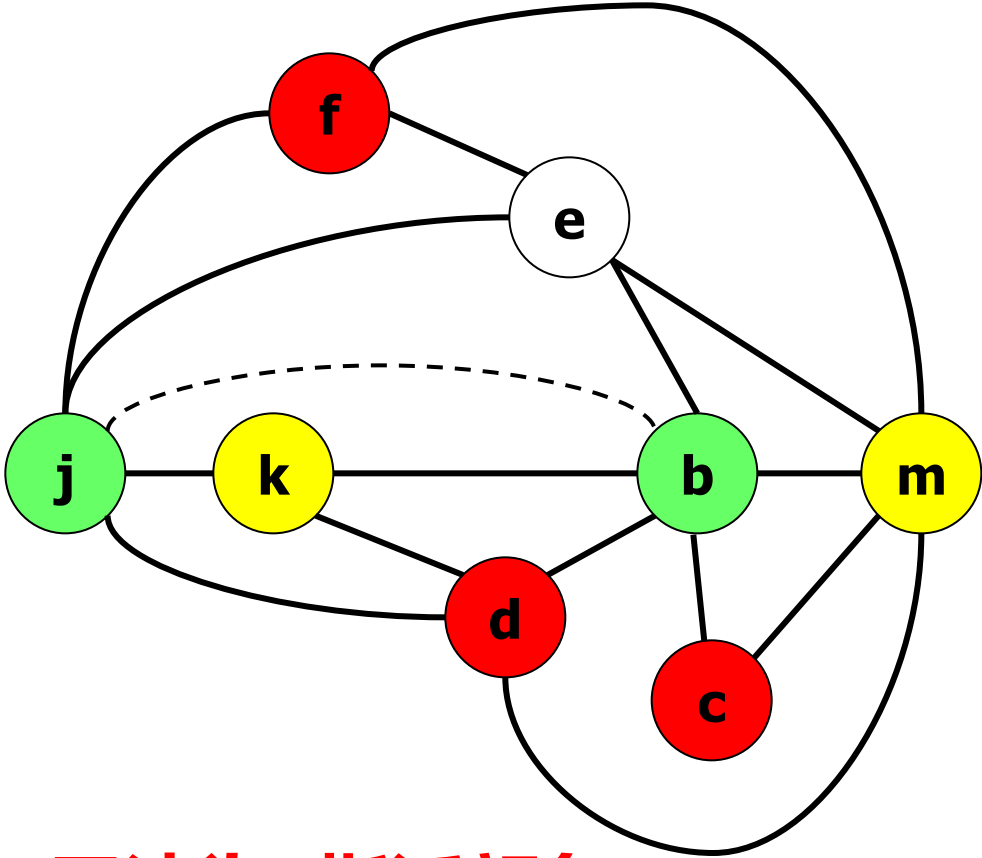
stack

R1

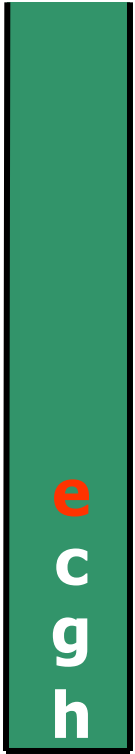
R2

R3

⑥选择: 将结点依次弹栈, 并指派颜色



无法为e指派颜色
e不得不真正溢出



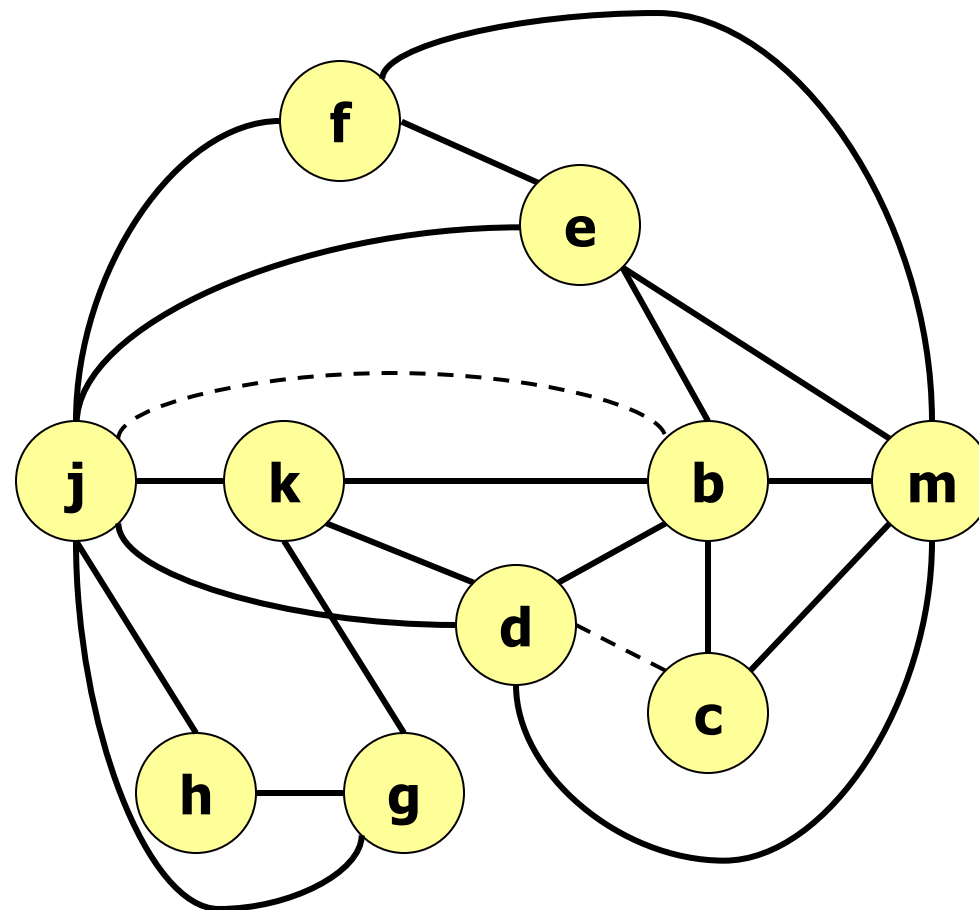
stack



⑦真正溢出: 插入溢出代码, 重构冲突图

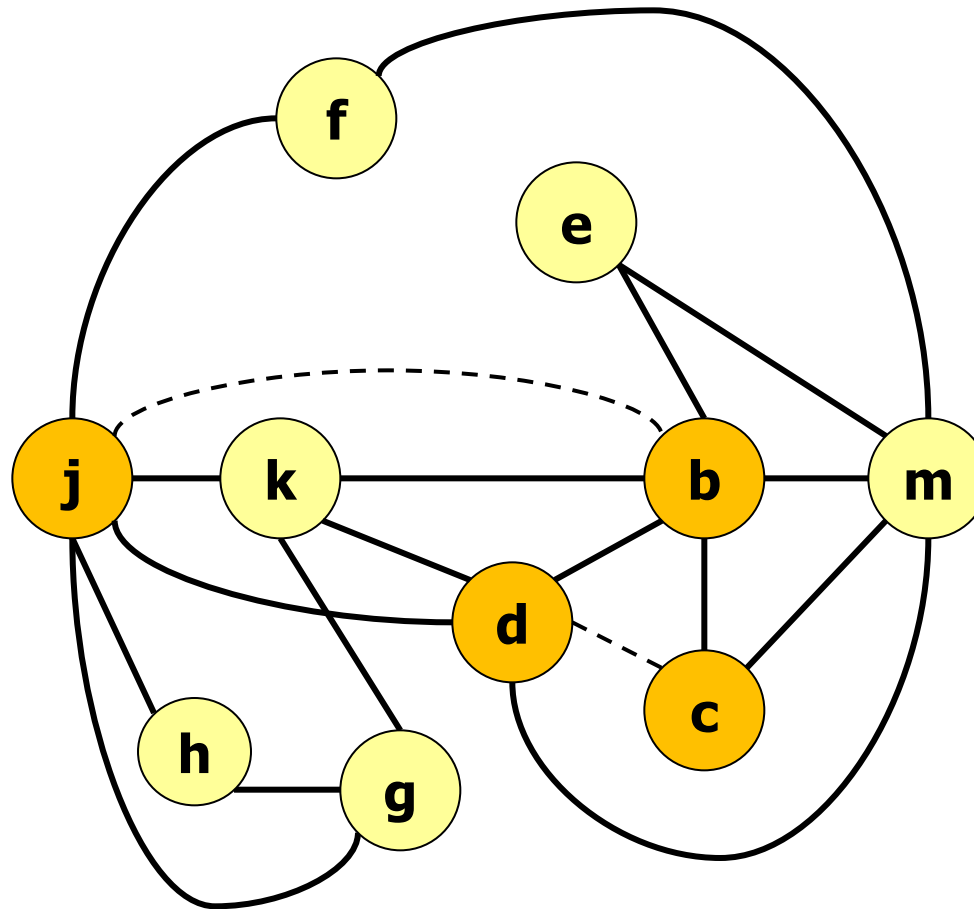
```

live-in: k j
  g ← mem[j+12]
  h ← k - 1
  f ← g + h
  e ← mem[j+8]
  store e, mem[j+8]
  m ← mem[j+16]
  b ← mem[f]
  e ← mem[j+8]
  c ← e + 8
  d ← c
  k ← m + 4
  j ← b
live-out: d k j
    
```



重构后的冲突图

■ ②化简: 删除与**传送无关**的**低度数**结点(**度<k**) , 压栈

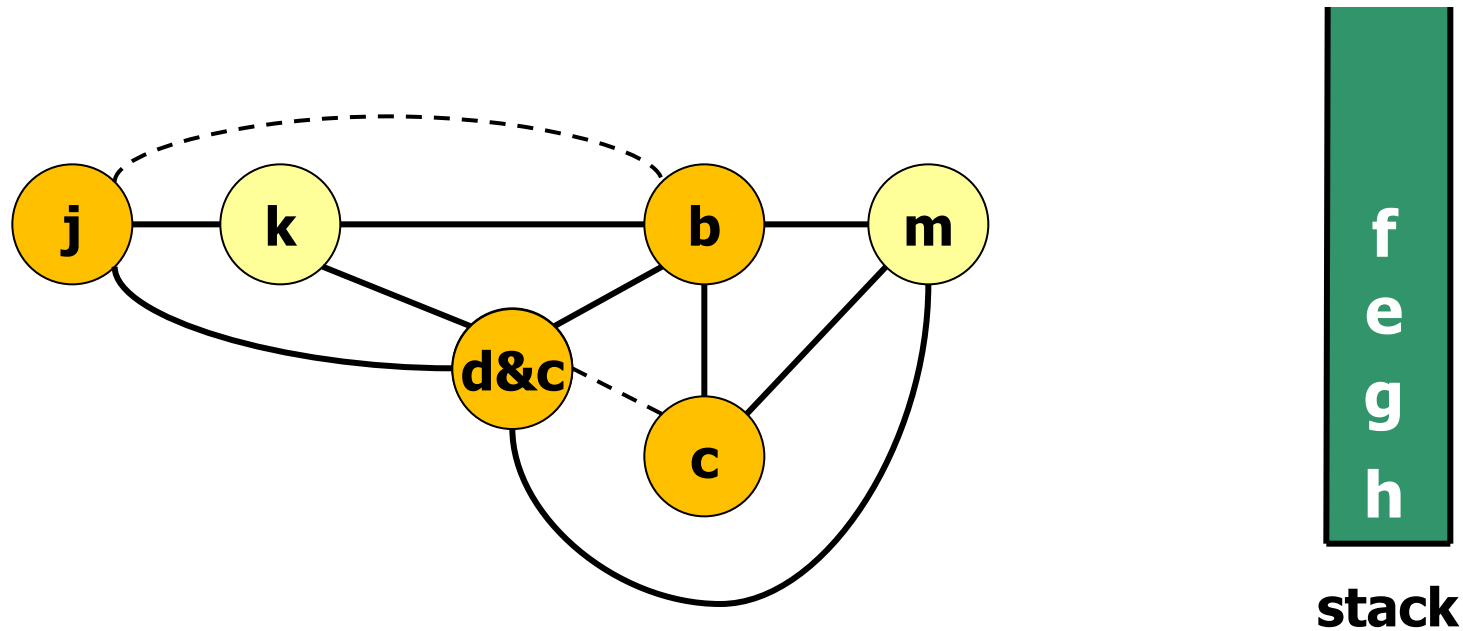


f
e
g
h

stack

■ ③合并: 根据George合并条件, 保守合并**传送相关**的结点

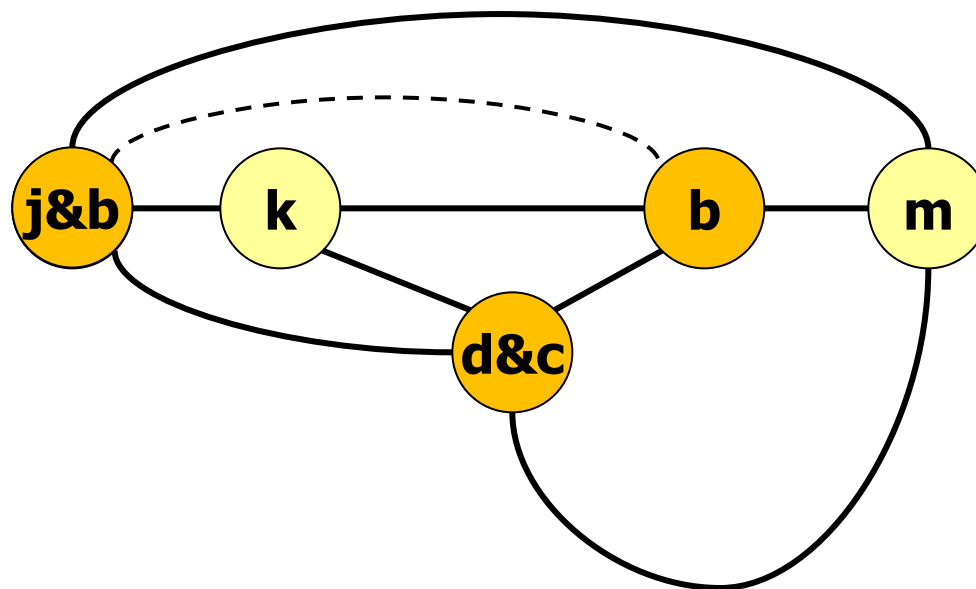
合并c-d? c的邻居b和m, 也是d的邻居, 因此c-d可以合并



■ ③合并: 根据George合并条件, 保守合并**传送相关**的结点

合并j-b?

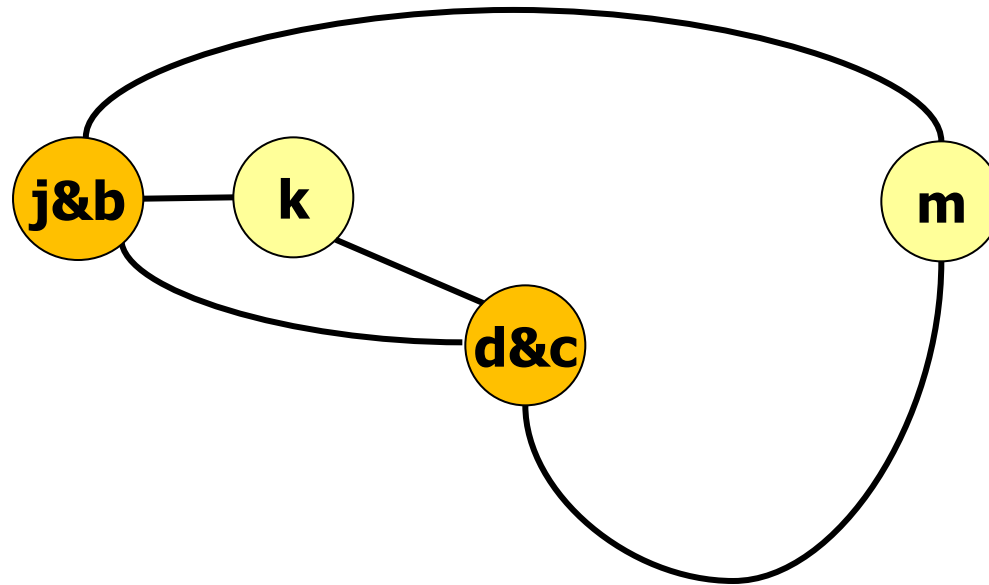
b的邻居k和d&c, 也是j的邻居
b的邻居m是低度数结点, 因此j-b
可以合并



f
e
g
h

stack

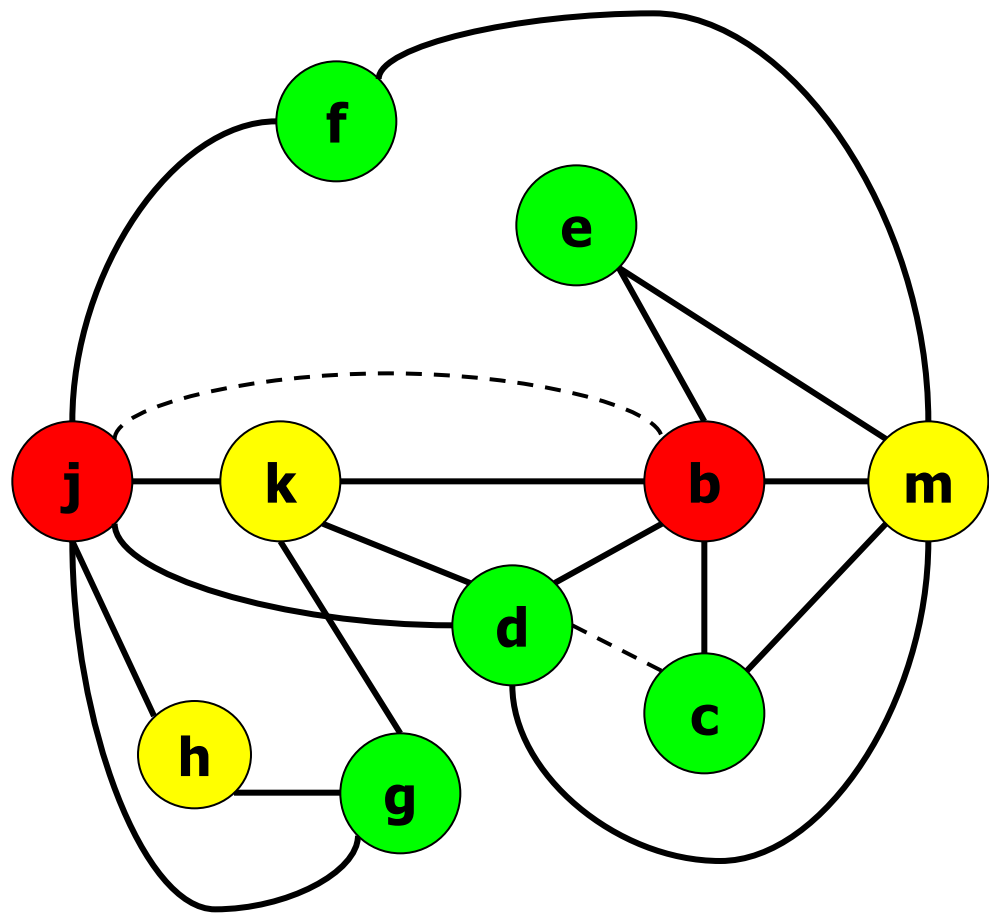
■ ②继续化简: 删除与**传送无关的低度数**结点(**度< k**) , 压栈



d&c
k
j&b
m
f
e
g
h

stack

⑥选择: 将结点依次弹栈, 并指派颜色



d&c
k
j&b
m
f
e
g
h

stack

R1

R2

R3

■图着色寄存器分配方法

- ⊕Chaitin算法 (1981)
- ⊕Briggs改进算法 (1994)
- ⊕George改进算法 (1996)



在继承中创新
在创新中发展

■优点: 全局近似最优解

■缺点: 效率不高, 比较耗时

- ⊕构建冲突图的时间复杂度, 最坏情况是 $O(n^2)$, n 为可分配对象数目
- ⊕需要迭代进行构建冲突图、化简、合并、冻结、选择、溢出的过程

10.1 寄存器分配概述

10.2 基于使用计数的寄存器分配方法

10.3 基于图着色的寄存器分配方法

10.4 Briggs图着色寄存器分配改进算法

10.5 George图着色寄存器分配改进算法

10.6 基于线性扫描的寄存器分配方法

- Poletto和Sarkar于1999提出
- 在一个遍(pass)中扫描所有变量的活跃区间(live intervals)
- 以贪心的方式为变量分配寄存器
- 优点: 实现简单, 算法高效, 生成的代码质量相对较高

全局寄存器分配



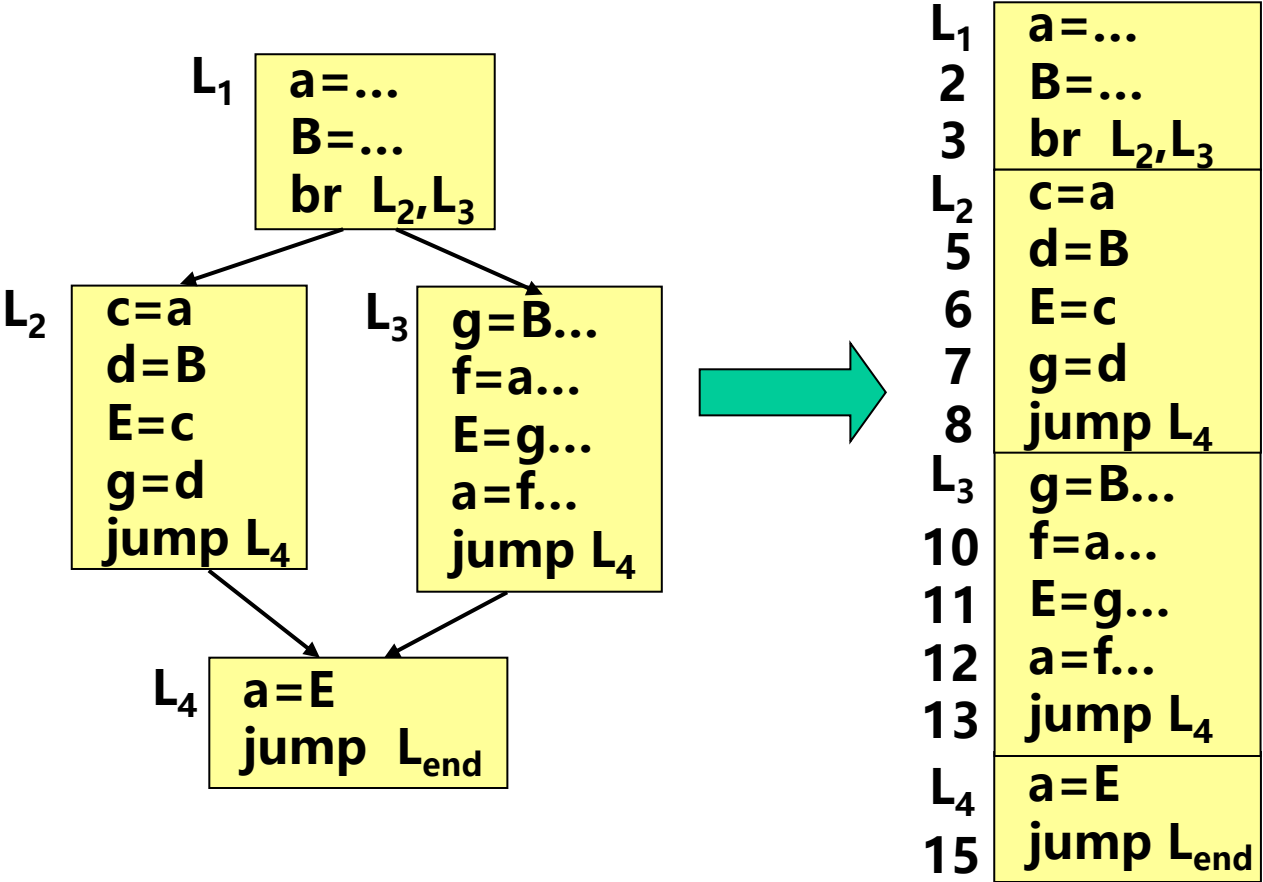
给一个由**区间**组成的**有序序列**指派颜色的问题

- 已经证明对有序区间可以采用贪心算法获得最优着色
- 线性扫描虽然不是最优算法，但可以使用它近似解决寄存器分配问题

■对程序模式的要求

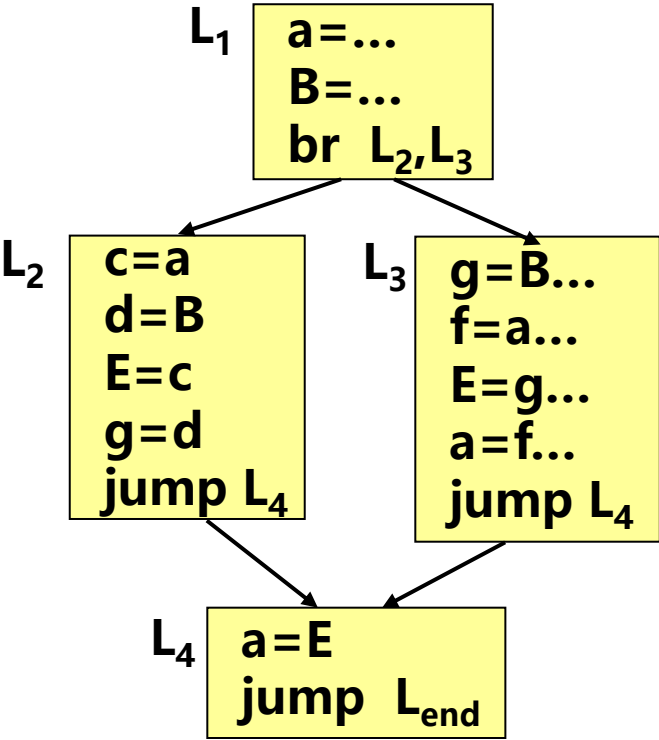
- ⊕中间表示由三地址码(如四元式)或伪指令组成
- ⊕指令操作数用虚寄存器(符号寄存器)表示, 虚寄存器个数无限
- ⊕基本块按某种顺序**线性排列**, 使得指令序列可以依次编号
 - 基本块可以按照控制流图的深度为主的顺序线性排列
- ⊕上述要求对大多数编译器的中间表示而言, 都是具备的

■ 示例



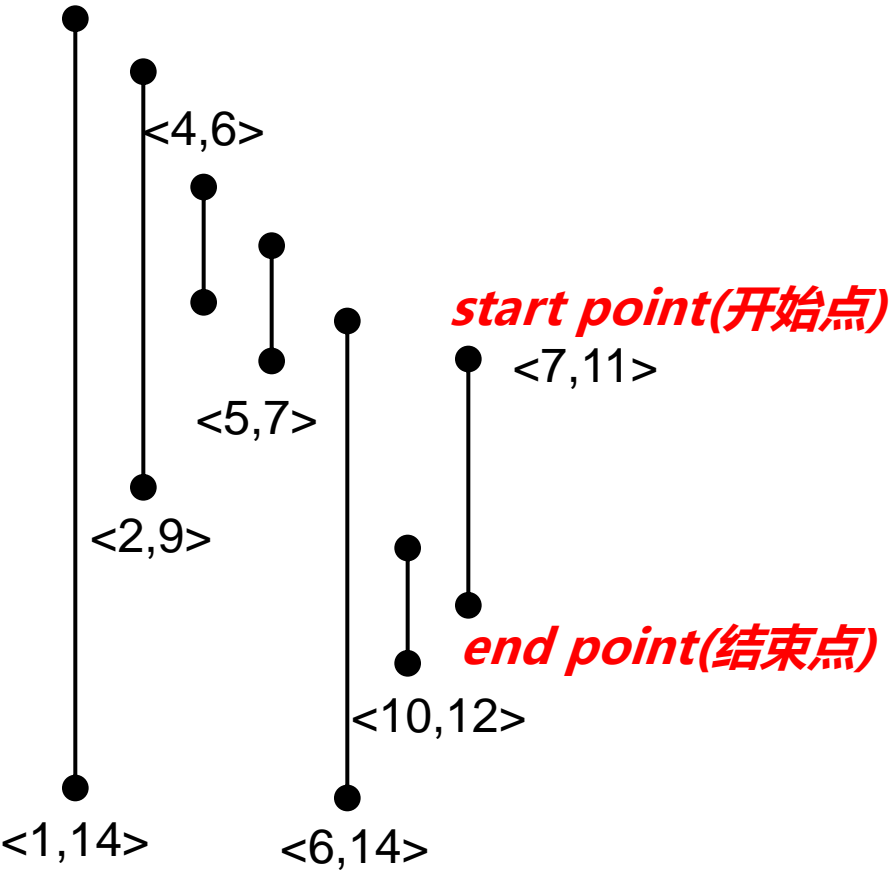
- 对于变量 v ，如果指令序列中存在一段区间 $[i, j]$ ，其中 $1 \leq i \leq j \leq N$ ，使得变量 v 只在此区间内是活跃的，就称 $[i, j]$ 是变量 v 的活跃区间 (live interval)
- ⊕ 变量 v 不在 i 之前的任何一条指令 i_k 处活跃 ($i_k < i$)，也不在 j 之后的任何一指令 j_k 处活跃 ($j_k > j$)
- ⊕ 不存在编号小于 i 的指令 i_k ($i_k < i$) 定值或使用变量 v ，也不存在编号大于 j 的指令 j_k ($j_k > j$) 定值或使用变量 v
- 活跃区间可以包含多次定值，多次使用

■ 示例



L ₁	a=...
2	B=...
3	br L ₂ ,L ₃
L ₂	c=a
5	d=B
6	E=c
7	g=d
8	jump L ₄
L ₃	g=B...
10	f=a...
11	E=g...
12	a=f...
13	jump L ₄
L ₄	a=E
15	jump L _{end}

intervals of
a B c d E f g



■ 基本思想

- ⊕ 如果两个活跃区间重叠，则称它们存在**冲突**
- ⊕ 分配寄存器给尽可能多的活跃区间，保证存在冲突的区间不分配相同的寄存器
- ⊕ 如果在程序某点重叠的活跃区间数 n 大于可分配的寄存器数 R ，则至少有 $n-R$ 个活跃区间要分配到存储器中

■ 维护两张表

- ⊕ 一张表存放**待分配**的活跃区间，按**开始点增加**的顺序存放
- ⊕ 一张表存放**已分配寄存器但还未到达其结束点**的活跃区间，按**结束点增加**的顺序存放(**active表**)

■ “已到期”的区间

- ⊕ 已分配寄存器的区间的结束点**早于**待分配的区间的开始点
- ⊕ 已到期的区间可以释放所占用的寄存器资源

■ “未到期”的区间

- ⊕ 已分配寄存器的区间的结束点**晚于**待分配的新区间的开始点

■按**开始点增加**的顺序依次扫描待分配的活跃区间

■在每一步

- ⊕给**待分配的区间**指派一个颜色
- ⊕如果没有可用的颜色，则尝试释放“**已到期**”的区间
- ⊕如果没有已到期的区间，则从所有已分配寄存器的区间和待分配的新区间中选择一个区间溢出(溢出**结束点最远**的区间)

LinearScanRegisterAllocation

$active \leftarrow \emptyset$

foreach live interval i , in order of **increasing start point**

ExpireOldIntervals(i) //释放已到期的区间

if $\text{length}(active) = R$ **then** //无可可用寄存器，则溢出一个区间

SpillAtInterval(i)

else //有可用寄存器，则将其分配给区间 i ，并将区间 i 插入 $active$ 表

$register[i] \leftarrow$ a register removed from pool of free registers

add i to $active$, sorted by **increasing end point**

ExpireOldIntervals(*i*)

foreach interval *j* in *active*, in order of **increasing end point**

if *endpoint[j]* > *startpoint[i]* **then**

return //没有“已到期”的区间

else

 remove *j* from *active* //从**active**表中删除*j*

 add *register[j]* to pool of free registers //释放*j*占用的寄存器

SpillAtInterval(i)

last = last interval in *active*

if $endpoint[last] \geq endpoint[i]$ **then**

$register[i] \leftarrow register[last]$

$location[last] \leftarrow$ new stack location //溢出*last*

remove *last* from *active*

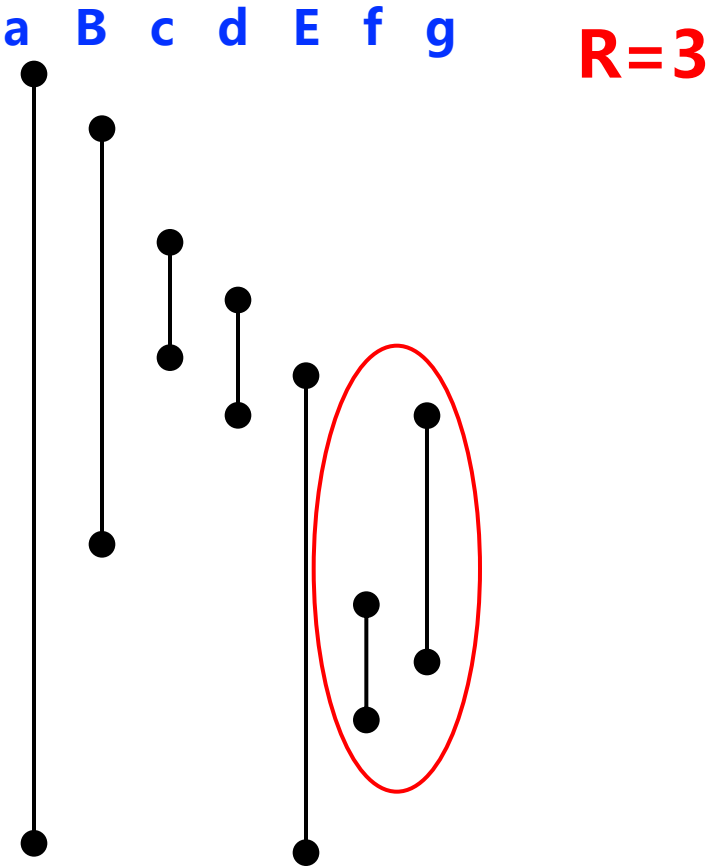
add *i* to *active*, sorted by increasing end point

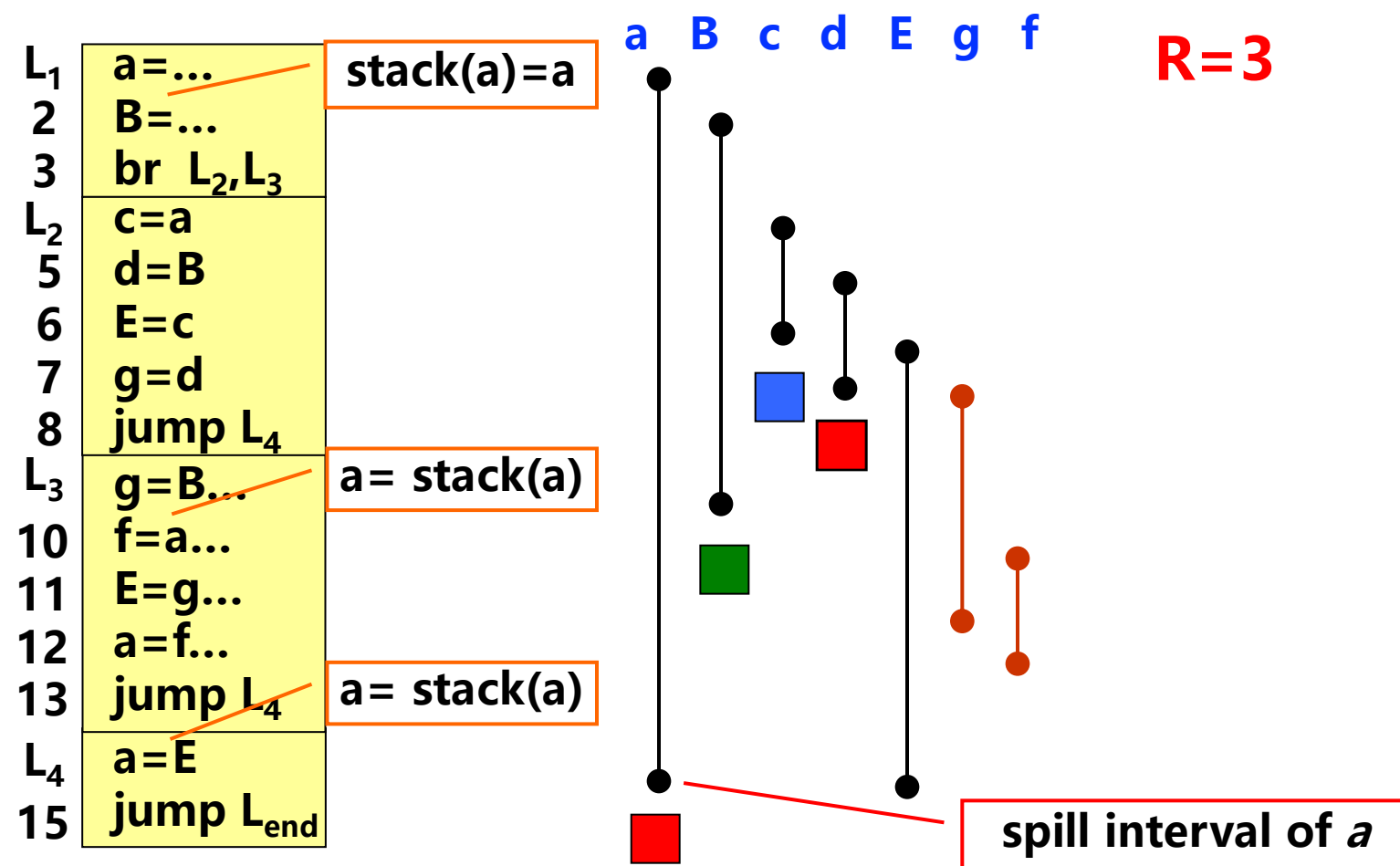
else

$location[i] \leftarrow$ new stack location //溢出*i*

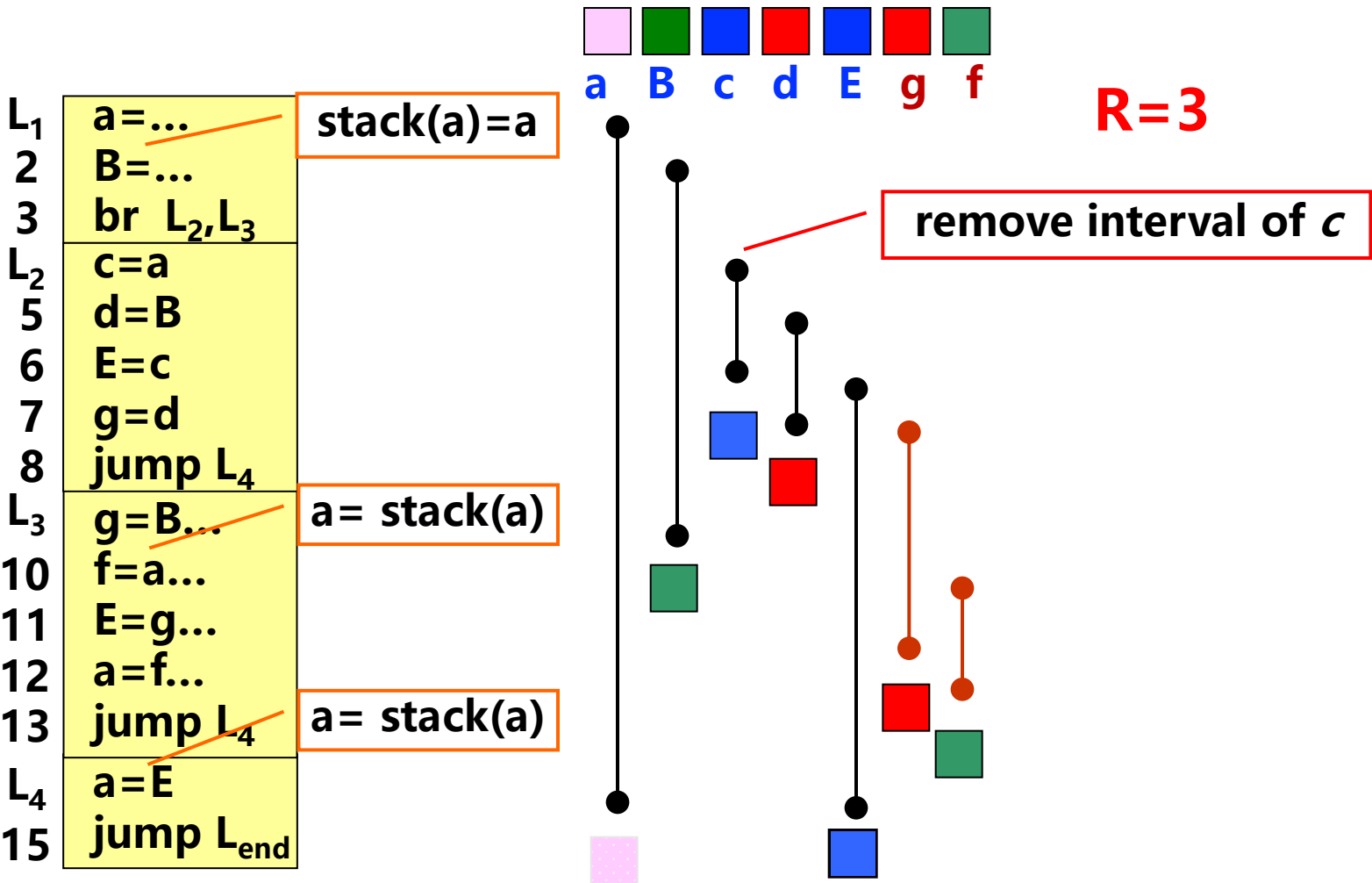
10.6.4 线性扫描算法示例

L ₁	a=...
2	B=...
3	br L ₂ ,L ₃
L ₂	c=a
5	d=B
6	E=c
7	g=d
8	jump L ₄
L ₃	g=B...
10	f=a...
11	E=g...
12	a=f...
13	jump L ₄
L ₄	a=E
15	jump L _{end}





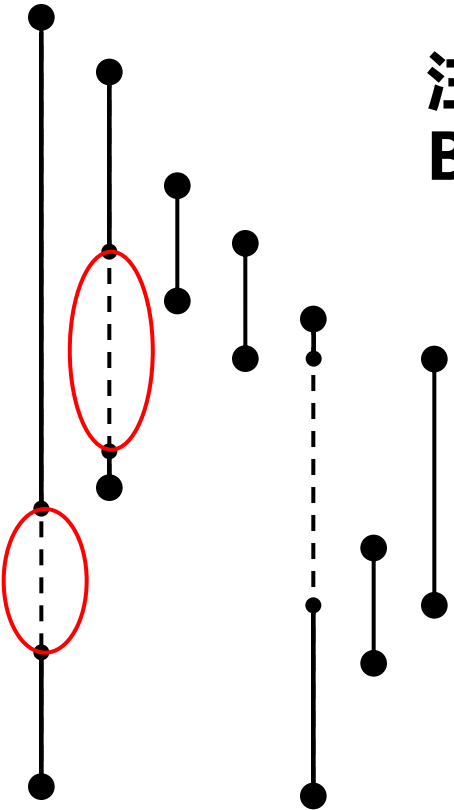
10.6.4 线性扫描算法示例



10.6.4 线性扫描算法示例

L ₁	a=...
2	B=...
	br L ₂ ,L ₃
L ₂	c=a
5	d=B
6	E=c
7	g=d
8	jump L ₄
L ₃	g=B...
10	f=a...
11	E=g...
12	a=f...
13	jump L ₄
L ₄	a=E
15	jump L _{end}

intervals of
a B c d E f g



注意到intervals存在的洞(holes),
B和E可以分配相同的寄存器

■ 处理活跃区间中的洞(holes)

⊕ Pinpacking model

Omri Traub, *et al.*, *Quality and speed in linear-scan register allocation*. In Conference on Programming Language Design and Implementation, 1998.

⊕ Optimal interval splitting

Christian Wimmer, *et al.*, *Optimized interval splitting in a linear scan register allocator*. In VEE, ACM, 2005.

⊕ Extended Linear scan

V. Sarkar, *et al.*, *Extended linear scan: an alternate foundation for global register allocation*. In LCTES/CC, ACM, 2007.

■ 优点: 寄存器分配速度

- ⊕ 分配速度是图着色算法的2倍
- ⊕ 适用于编译时间和代码质量同样重要的情况

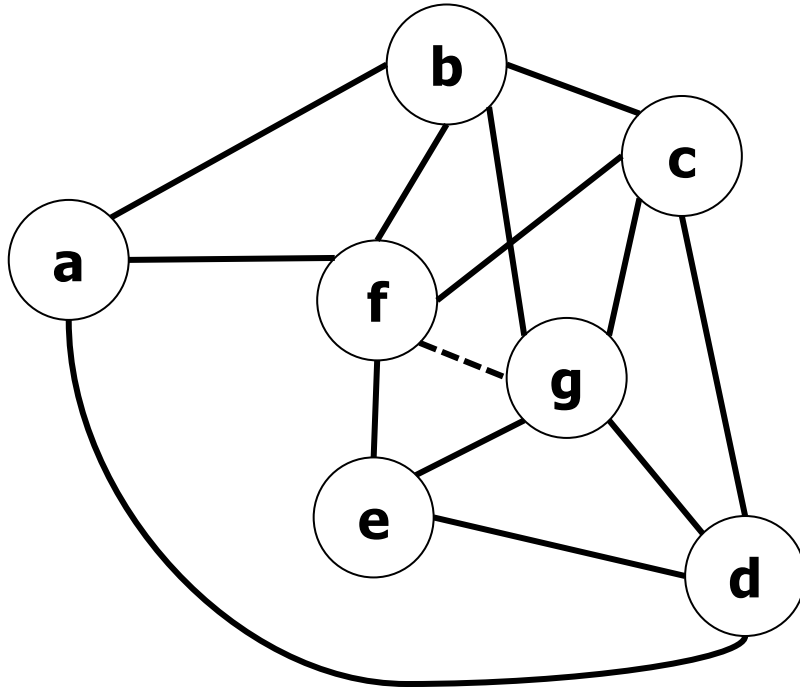
■ 缺点: 分配速度的提高是以一定程度对代码质量的影响为代价的

- ⊕ 基本块排序影响活跃区间的确定, 影响寄存器分配质量, 从而影响代码质量
- ⊕ 好的基本块排序使得变量的活跃区间更短、洞更少

- **寄存器分配是编译器必须具备的优化**
- **基于使用计数的寄存器分配方法**
- **基于图着色的寄存器分配方法**
 - ⊕ Chaitin算法, Briggs改进算法, George改进算法
 - ⊕ 全局近似最优解, 采用迭代方法, 比较耗时
- **基于线性扫描的寄存器分配方法**
 - ⊕ 实现简单, 采用贪心算法, 高效
 - ⊕ 基本块排序影响代码质量

■ 对如下冲突图运用George改进算法进行寄存器分配，假设寄存器数为3 ($k=3$)

⊕ 给出寄存器分配的具体过程(化简、合并、冻结、潜在溢出、选择等)和最终分配方案，如果需真正溢出，溢出度数最高的结点



- 《高级编译器设计与实现》(鲸书) 第16章
- 《现代编译原理C语言描述》(虎书) 第11章
- 论文

- ⊕ Richard A. Freiburghouse. Register Allocation via Usage Counts, *CACM*, vol.17, No.11, Nov. 1974.
- ⊕ G.J. Chaitin. et. al, Register allocation via coloring. *Computer Languages*, 1981.
- ⊕ P. Briggs, et al. Improvement to graph coloring register allocation. *Transactions on Programming Languages and Systems*, 1994.
- ⊕ L. George and Andrew W. Appel. Iterated register coalescing. *Transactions on Programming Languages and Systems (TOPLAS)*, 1996.
- ⊕ A. B. Kempe, On the geographical problem of four colours. *American Journal of Mathematics*.
- ⊕ M. Poletto and V. Sarkar. Linear scan register allocation. *Transactions on Programming Languages and Systems*, 1999.
- ⊕ Fanica Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph, *SIAM Journal on Computing*: 1(2), 1972.