Compiler Techniques

Lecture 12: Register Allocation

Tianwei Zhang

Outline

- **Overview of Register Allocation**
- **Local Register Allocation**
- **Global Register Allocation**

Outline

- **Overview of Register Allocation**
- **Local Register Allocation**
- **Global Register Allocation**

Motivation of Register Allocation

- Registers are valuable
  - Operations with registers are much faster that with main memory
  - We would like to keep as many local variables as possible in registers (as opposed to main memory)
- Registers are limited
  - Code generation can have arbitrary numbers of local variables
  - Actual physical machines have limited numbers of registers.
- We need to reuse the registers as much as possible
  - If the stored variable of a register is no longer needed, as we can use this register to hold other variables
- Register spilling: a variable has to be stored in the main memory if there are not available registers for it.
  - Introducing load and store operation for this variable (longer access time)

Register Allocation

- Register allocation: allocate and assign a fixed number of registers (*k*) to local variables
- Requirements:
  - Generate machine code that only uses *k* registers.
  - Minimize the number of register spilling
  - Minimize the memory space to hold spill variables
  - Efficient to identify the solution

Register Allocation Example

- Consider the following code snippet, where x, y and z are locals variables:
  - Recall liveness: a variable is live if its value is read before it is reassigned

```
                // x, y, z all dead
x = 23          // x live; y, z dead
y = 42          // x, y live; z dead
z = x + y       // y, z live; x dead
y = y + z       // x, y, z all dead
```

- Assume we have only two registers r and s
  - Allocate register r to both x and y?
  - Allocate register r to both x and z?
  - Allocate register r to both y and z?

Liveness and Register Allocation

- Interference: two variables are both live at one same point in the program

- Register allocation principle:
  - Two interfered variables cannot be allocated to the same register.
  - Conversely, variables x and y can share the same register if there is no point in the program where both x and y are live

- We can apply liveness analysis (Lecture 10) for each variable in the given code, and then perform register allocation (as introduced below)

Outline

- Overview of Register Allocation
- **Local Register Allocation**
- Global Register Allocation

Local Register Allocation

- Basic block: a straight-line code sequence with no branches
- Consider each of all the instructions in the given basic block. Let i be the instruction having the most live variables, which is m. Assume there are k physical registers.
  - If $m \leq k$, allocation is easy without the need of register spilling
  - If $m > k$, need to spill some variables to the main memory
- Since $m = 4$, no need for register spiling when $k \geq 4$
- Two allocation algorithms
  - Top-down
  - Bottom-up

```
                              Live variables
a = 1                         // a
b = a                         // a, b
c = a * b                     // a, b, c
d = 2                         // a, b, c, d
e = d - b                     // a, c, d, e
f = d + c                     // a, e, f
g = e * f                     // a, g
h = g - a                     // h
a = h + 1
```

Top-down Approach

- Consider each of all the instructions in the given basic block. Let i be the instruction having the most live variables, which is m. Assume there are k physical registers. If $m > k$:
  - Rank variables by the number of occurrences
  - Allocate the first k variables to registers
  - Spill the other variables.

Top-down Example

- Assume there are $k = 3$ physical registers

Spill c to memory

Restore c from memory

```
                    Live variables
a = 1               // a
b = a               // a, b
c = a * b           // a, b, c
d = 2               // a, b, c, d
e = d - b           // a, c, d, e
f = d + c           // a, e, f
g = e * f           // a, g
h = g - a           // h
a = h + 1
```

```
# of occurrence
a = 5  r1
b = 3  r2
c = 2  Spill !
d = 3  r3
e = 2  r2
f = 2  r3
g = 2  r3
h = 2  r1
```

Bottom-up Approach

- Consider each of all the instructions in the given basic block. Let i be the instruction having the most live variables, which is m. Assume there are k physical registers. If $m > k$:
  - Start with an empty register set
  - Load on demand
  - When no register is available, free one
    - Spill the variable whose next use is farthest in the future

Bottom-up Example

- Assume there are $k = 3$ physical registers

Spill a to memory

Restore a from memory

```
                    Live variables
a = 1               // a
b = a               // a, b
c = a * b           // a, b, c
d = 2               // a, b, c, d
e = d - b           // a, c, d, e
f = d + c           // a, e, f
g = e * f           // a, g
h = g - a           // h
a = h + 1
```

a	r1
b	r2
c	r3
d	r1 (Spill a)
e	r2
f	r3
g	r1
h	r1

Outline

- Overview of Register Allocation
- Local Register Allocation
- **Global Register Allocation**

Global Register Allocation

- Local allocation only works in a single basic block

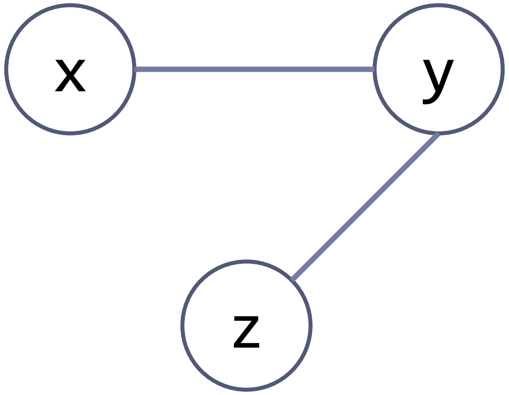
- Global register allocation across multiple blocks with branches

- Modern global allocator adopts a graph-colouring strategy
  - Construct an interference graph
  - Find a k-colouring for this interference graph
  - Map colours to registers

Interference Graph

- Interference graph:
  - One node for each local variable
  - Undirected edge between nodes for x and y if they are both live at some point in the program

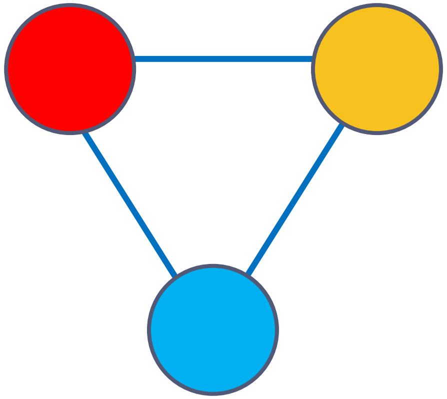
```
x = 23
y = 42
z = x + y
y = y + z
```



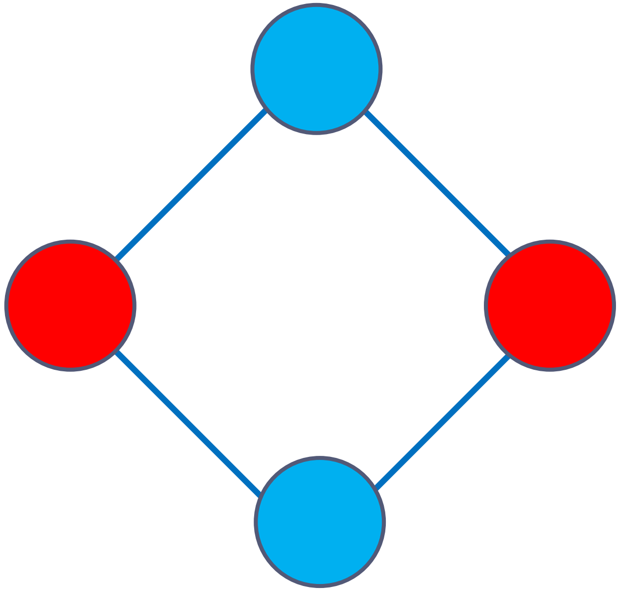
- x and z can be allocated to the same register, but not x and y, or y and z

Graph Colouring

- Graph colouring is an assignment of colours to the nodes of an interference graph such that there is no edge between nodes with the same colour
- A graph is said to be k-colourable if it is a graph colouring with k different colours.



3-colourable



2-colourable

Register Allocation with Graph Colouring

- If the interference graph is k-colourable, then there is a register assignment that uses no more than k registers such that no register spilling is necessary

- Degree of a node is a loose upper bound on the colourability
  - Degree: the number of neighbors
  - If the degree of each node is smaller than k, then this graph is always k-colourable.
  - k may not be the smallest number that makes the graph colourable.

- NP-hard problem
  - Need heuristic solution

Chaitin's Algorithm

- A good heuristic algorithm for finding k-colouring (register allocation) solution. It is based on the following two insights:

- Consider a graph G. If there exists a node n with fewer than k neighbours, and the graph without n is k-colourable, then G is also k-colourable.
  - We can simplify the graph by disregarding n

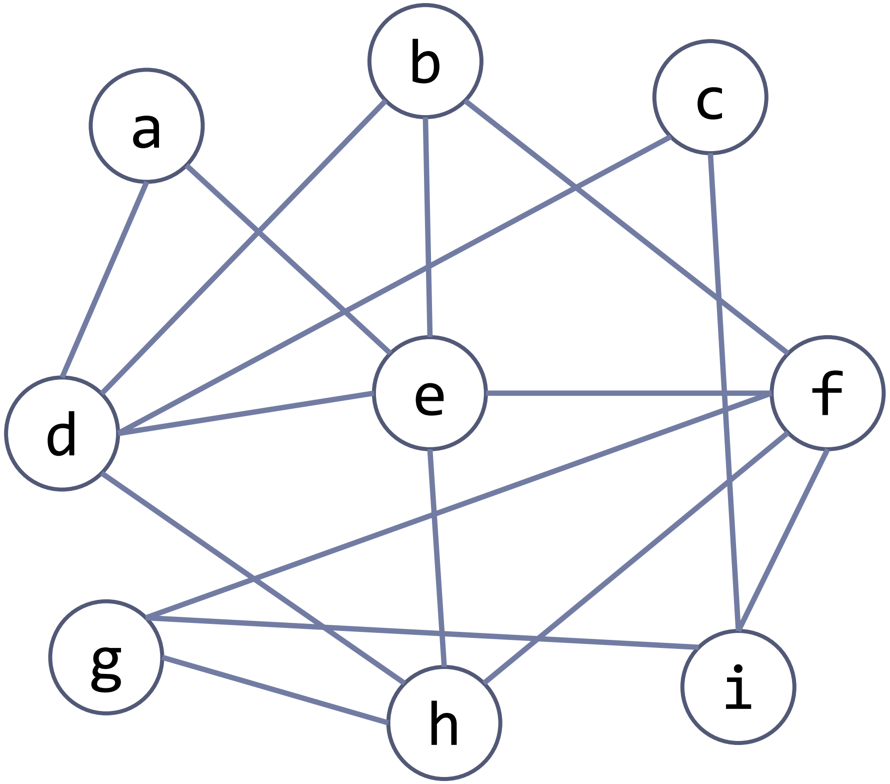
- If every node in the graph has at least k neighbours, we need to select a spill candidate
  - It is generally a good idea to choose a spill candidate that has the maximum number of neighbours: throwing it out will help most to simplify the remaining colouring problem

Chaitin’s Algorithm: Pseudocode

```
INPUT: Interference graph $IG$, number $k$ of registers

$s$ = empty stack
$p$ = empty list

while $IG$ not empty do
    if there is node $n$ with neighbours($n$) < $k$ then
        remove $n$ from $IG$ and push it onto $s$
    else
        let $d$ be node with maximum number of neighbours
        remove $d$ from $IG$ and push it into $p$

For each node $n$ in $p$ do
    do not allocate a register for $n$. Instead, insert the store/load code for $n$.

while $s$ not empty do
    pop node $n$ off $s$
    allocate $n$ a register not allocated to any of neighbours($n$)
```

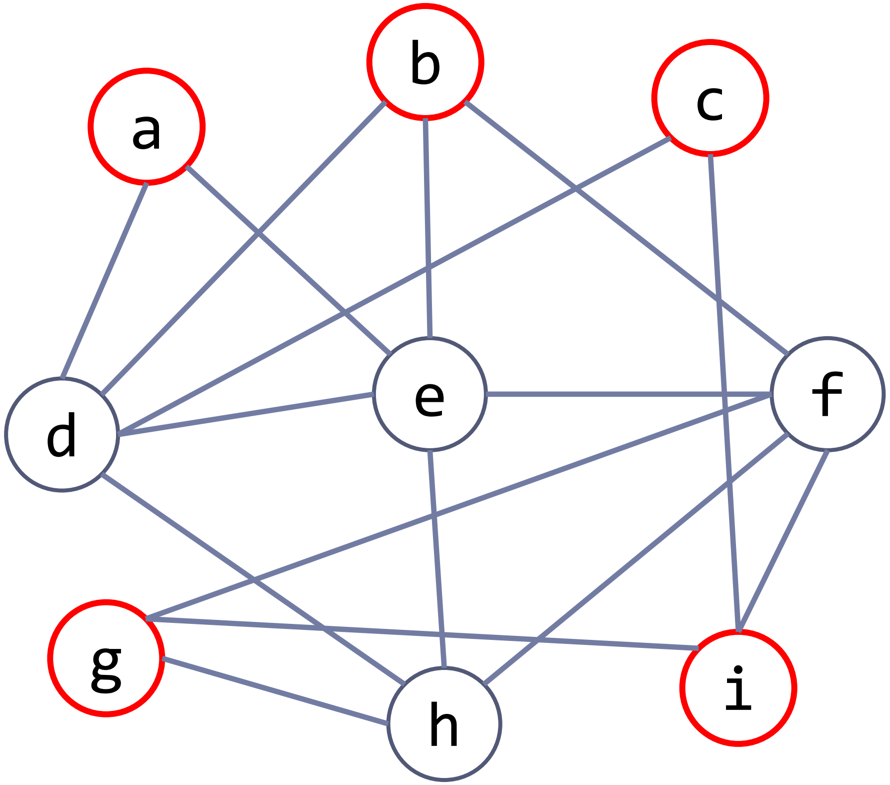
Example

- Assume we have the following interference graph. Can we allocate registers to the variables using four registers r, s, t, u?



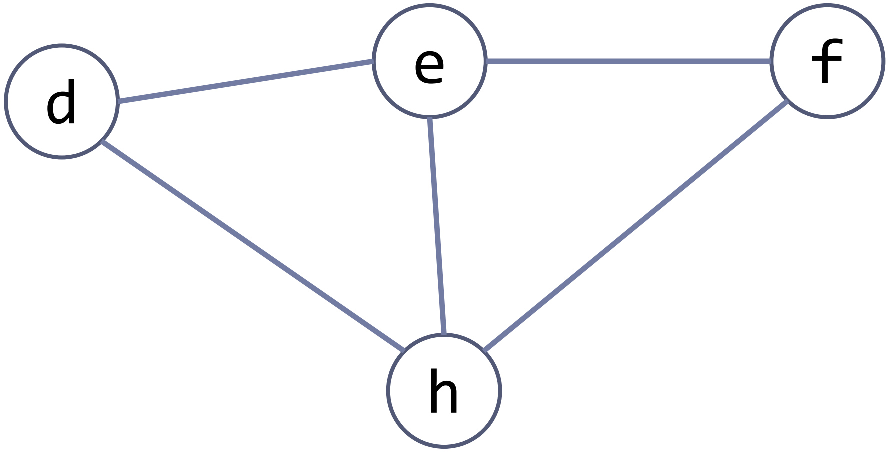
Example

- Find nodes with fewer than four neighbours



Example

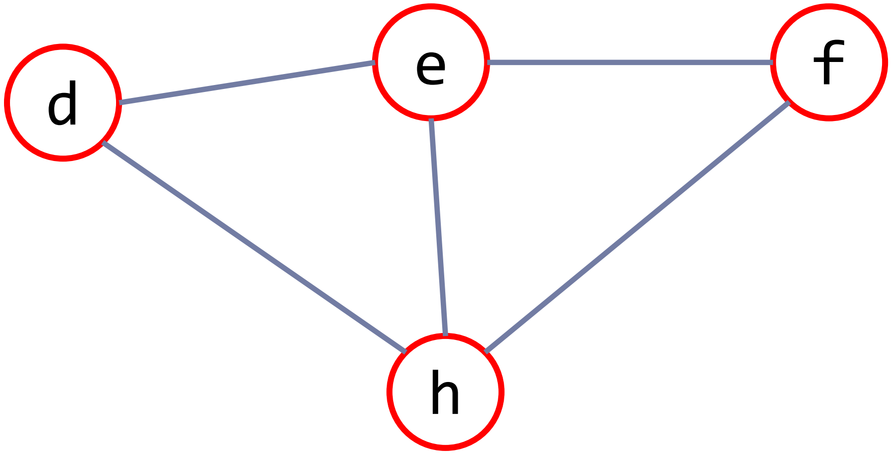
- Remove these nodes and push them onto stack



i
g
c
b
a

Example

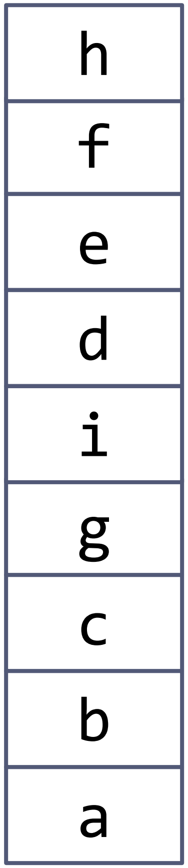
- Find nodes with fewer than four neighbours



i
g
c
b
a

Example

- Remove these nodes and push them onto stack



h

f

e

d

i

g

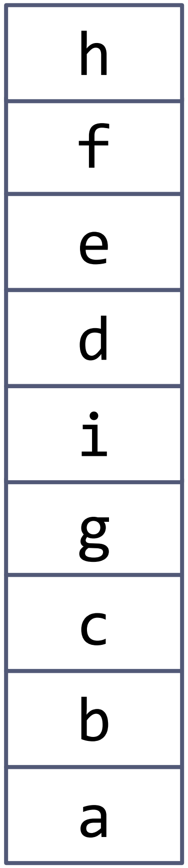
c

b

a

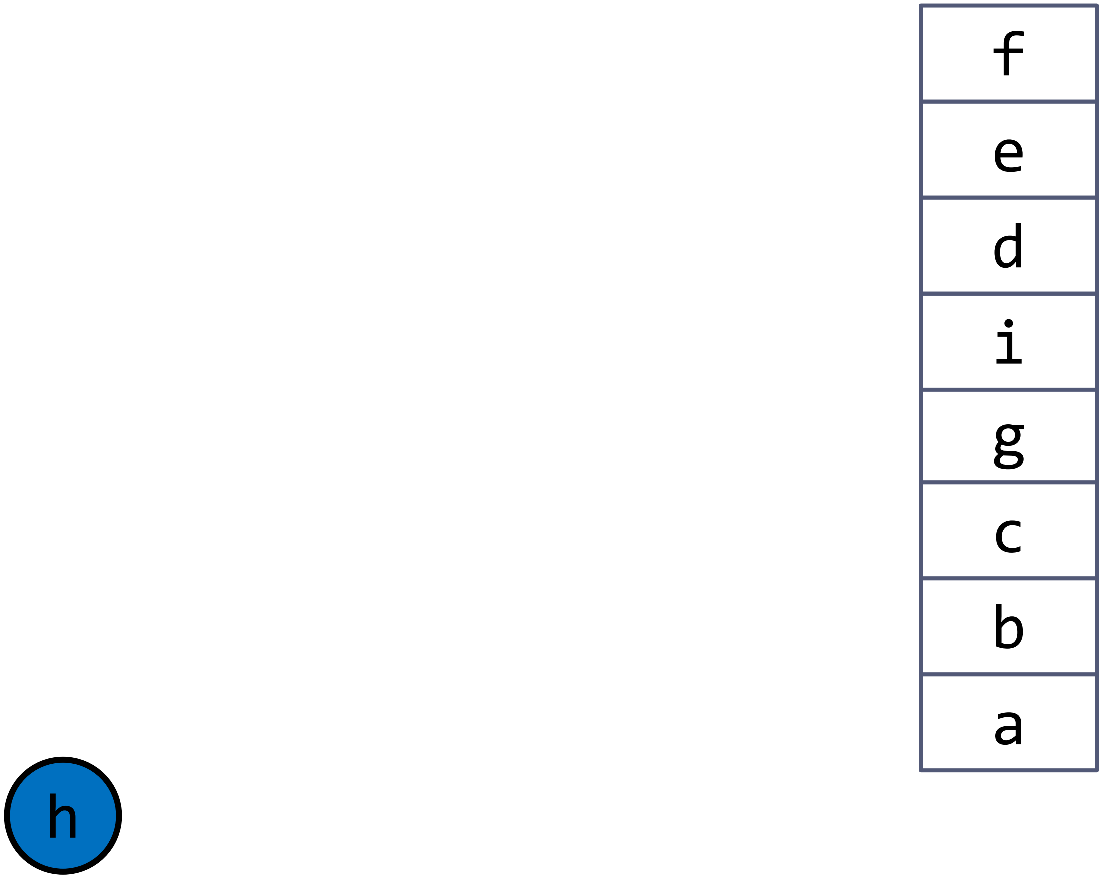
Example

- No nodes left, so pop nodes off the stack one by one, and assign a “colour” (register)



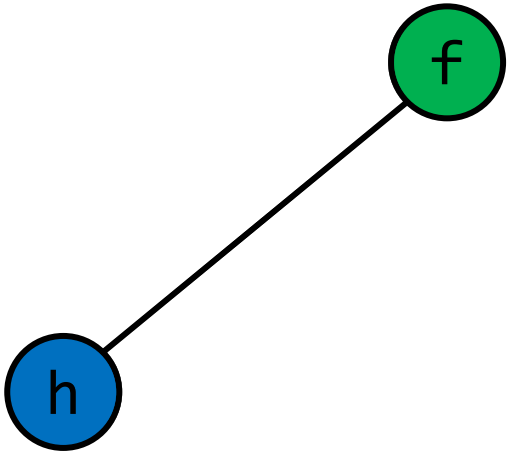
Example

- Pop off h, assign a register



Example

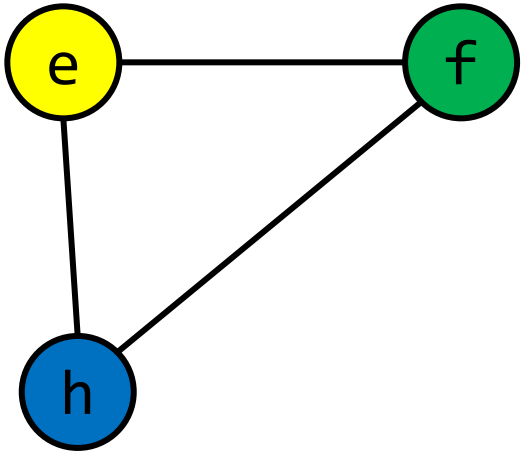
- Pop off f, assign a register



e
d
i
g
c
b
a

Example

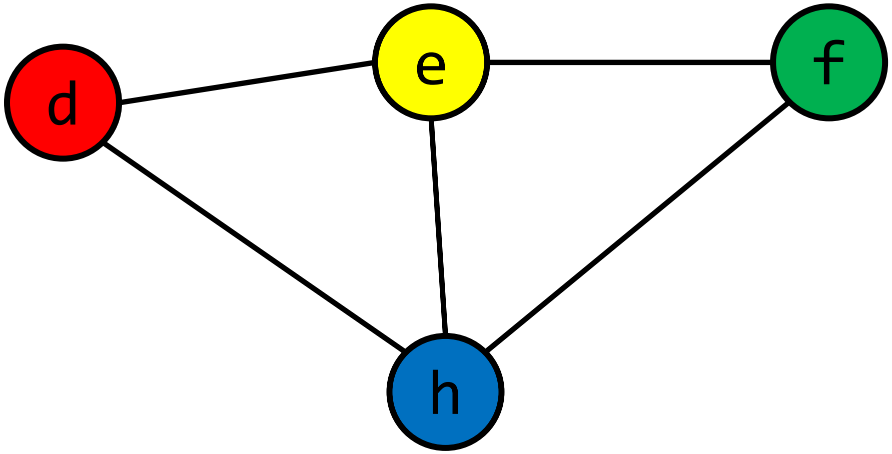
- Pop off d, assign a register



d
i
g
c
b
a

Example

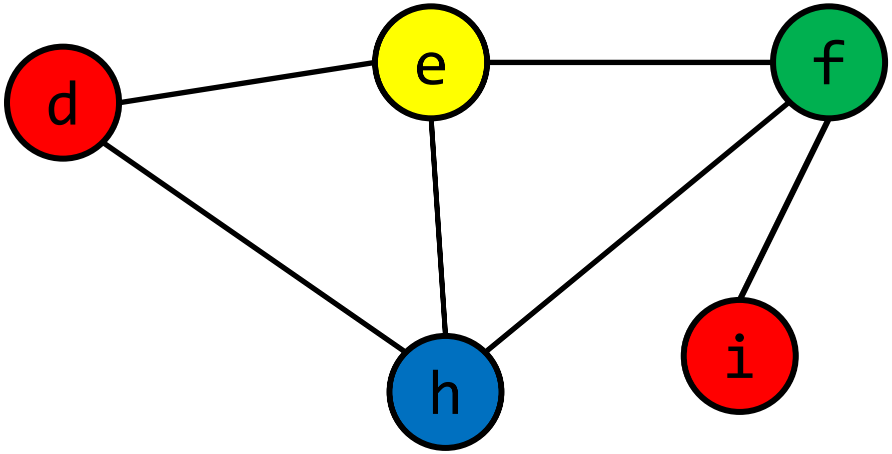
- Pop off e, assign a register



i
g
c
b
a

Example

- Pop off i, assign a register



g

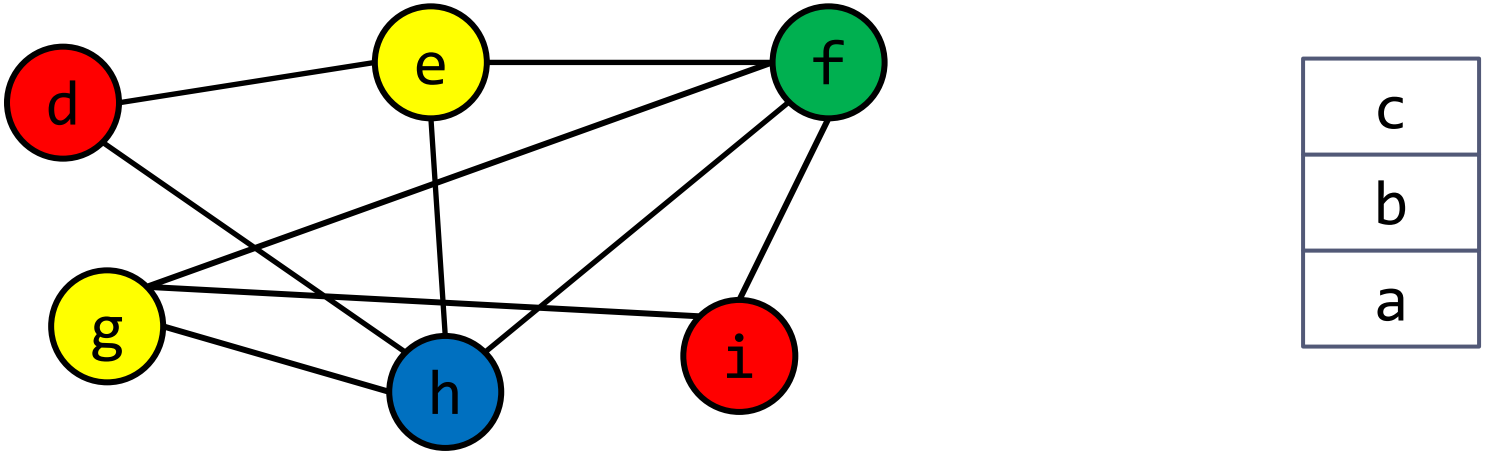
c

b

a

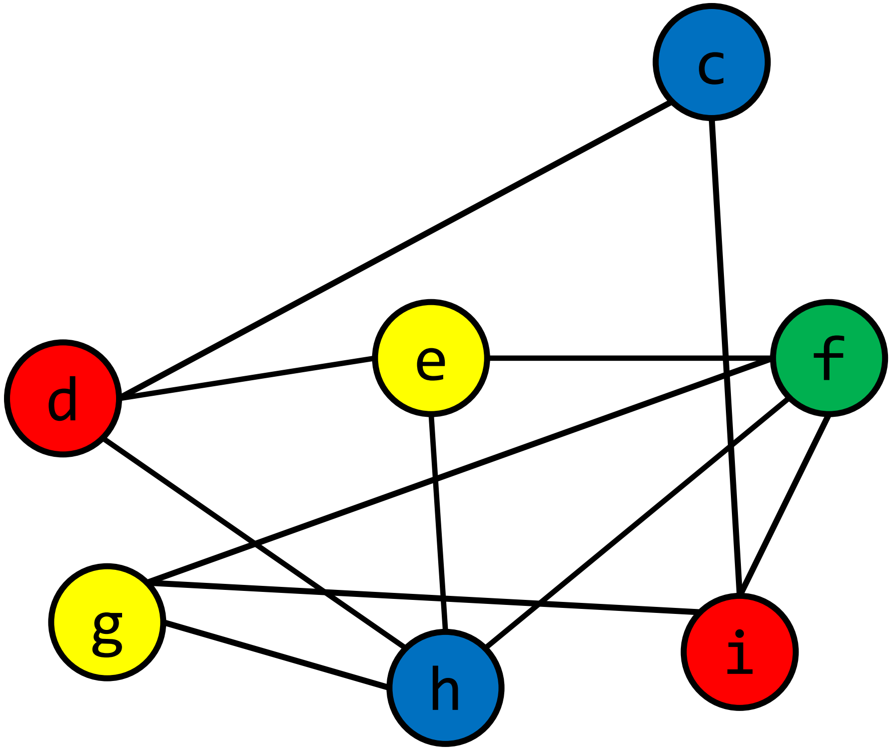
Example

- Pop off g, assign a register



Example

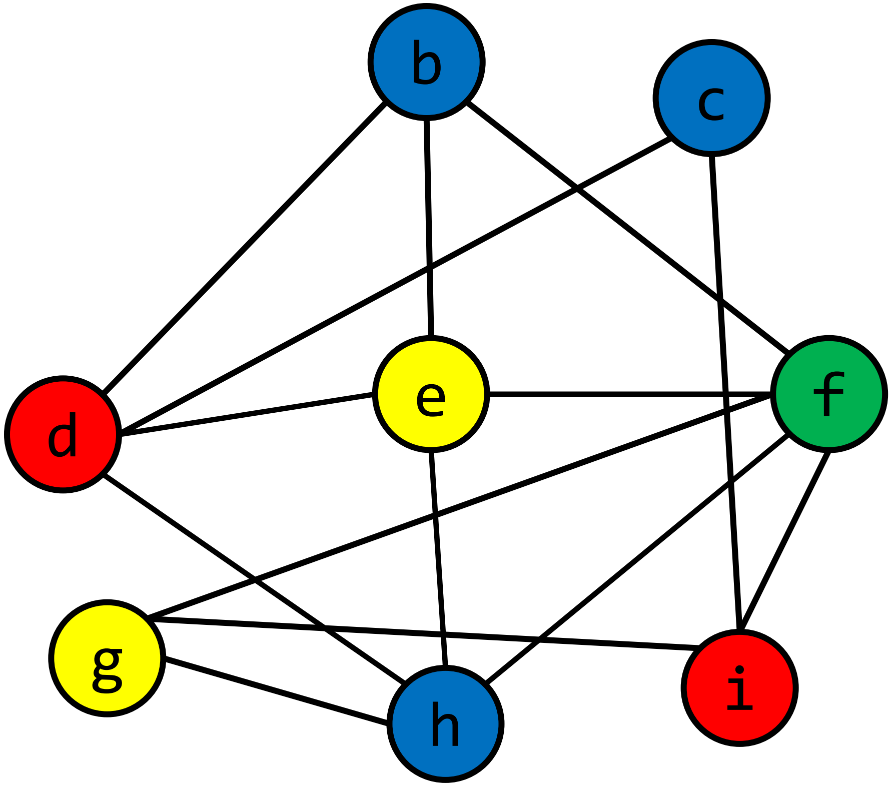
- Pop off c, assign a register

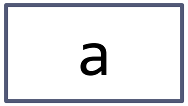


b
a

Example

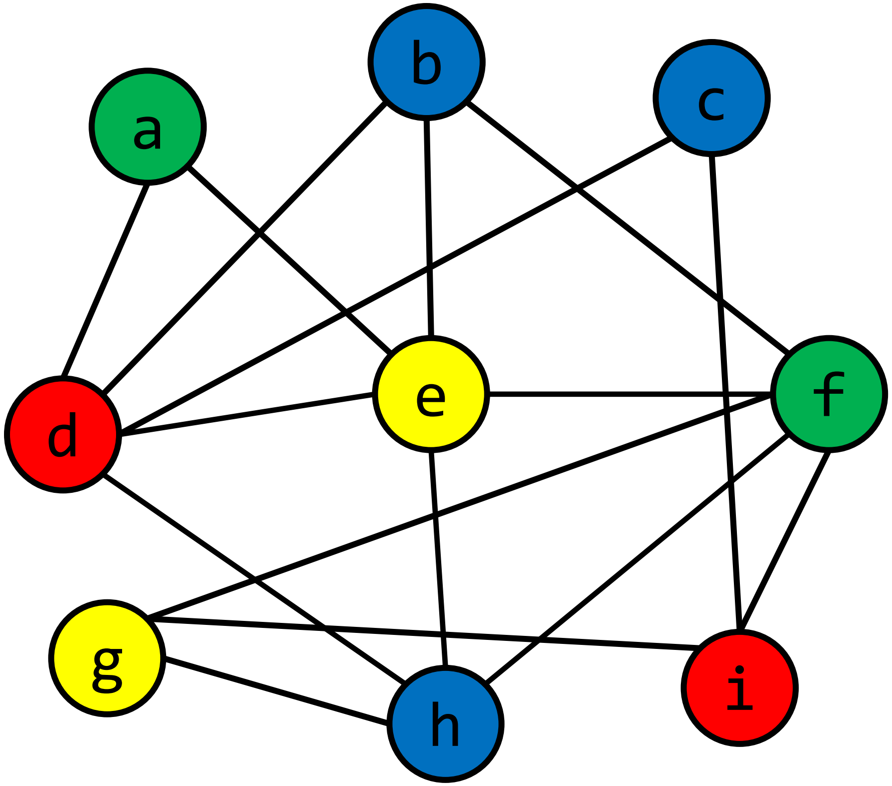
- Pop off b, assign a register





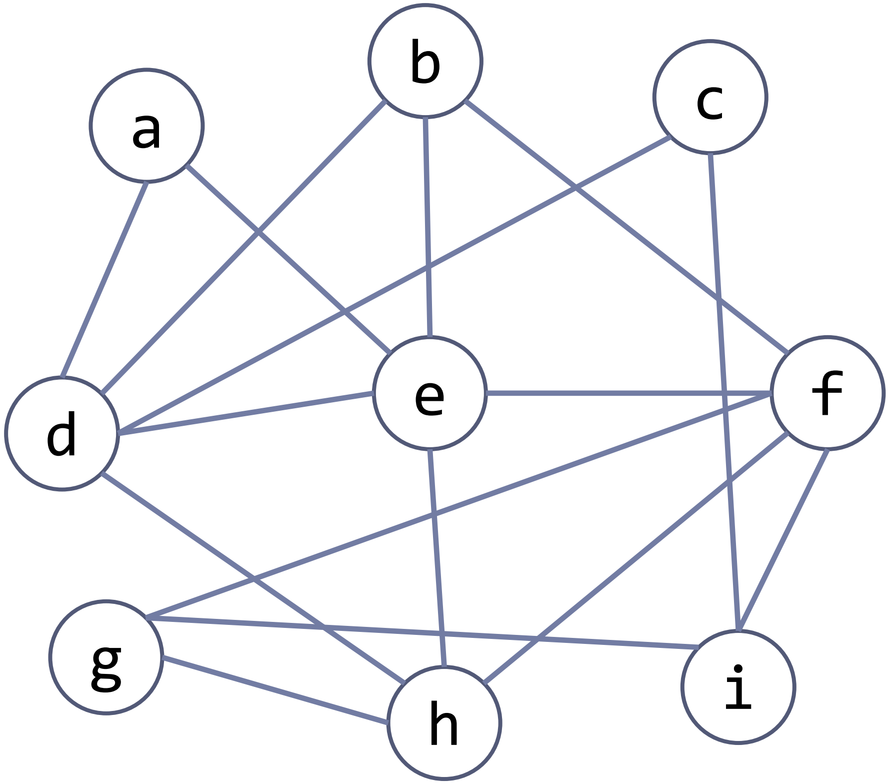
Example

- Pop off a, assign a register
- We have found a 4-colouring of the interference graph



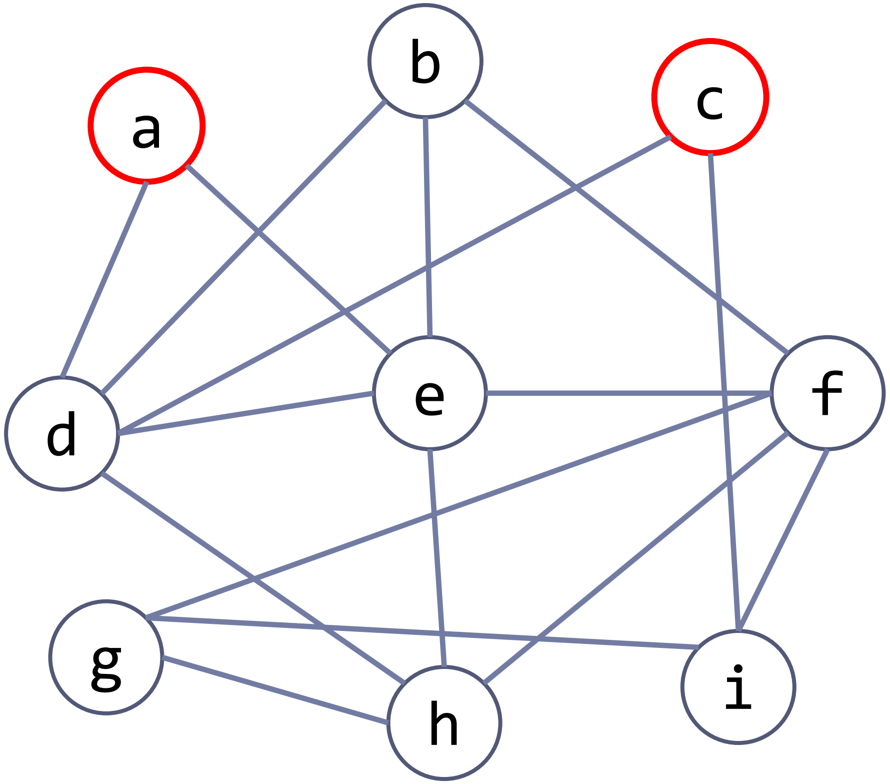
Example

- What if we only have three registers r, s, t?



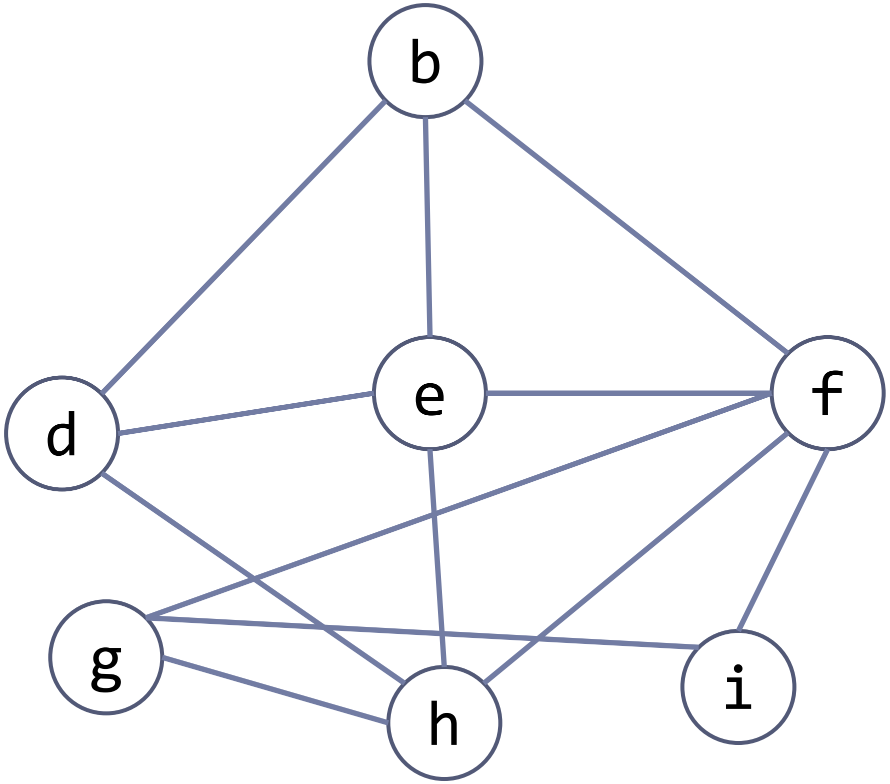
Example

- Find nodes with fewer than three neighbours



Example

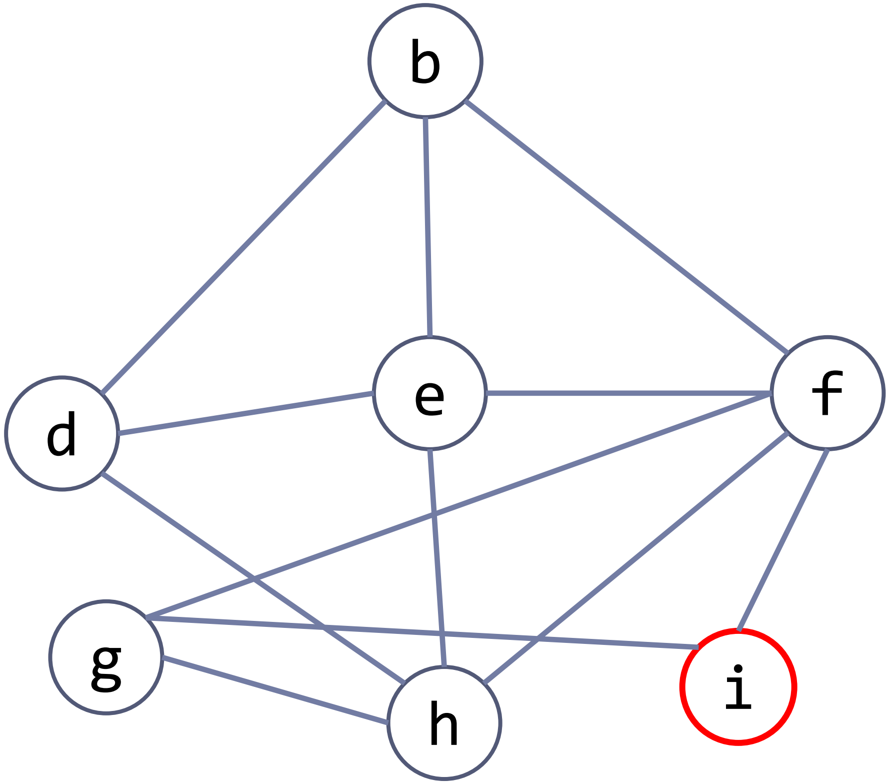
- Remove these nodes and push them onto stack

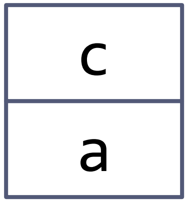


c
a

Example

- Find nodes with fewer than three neighbours

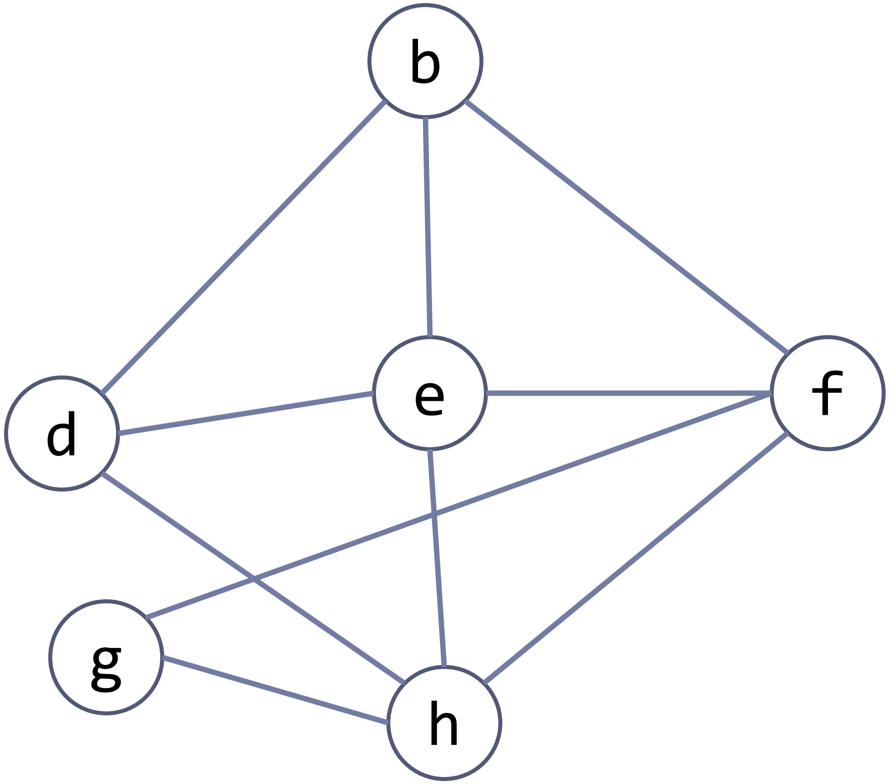



c

a

Example

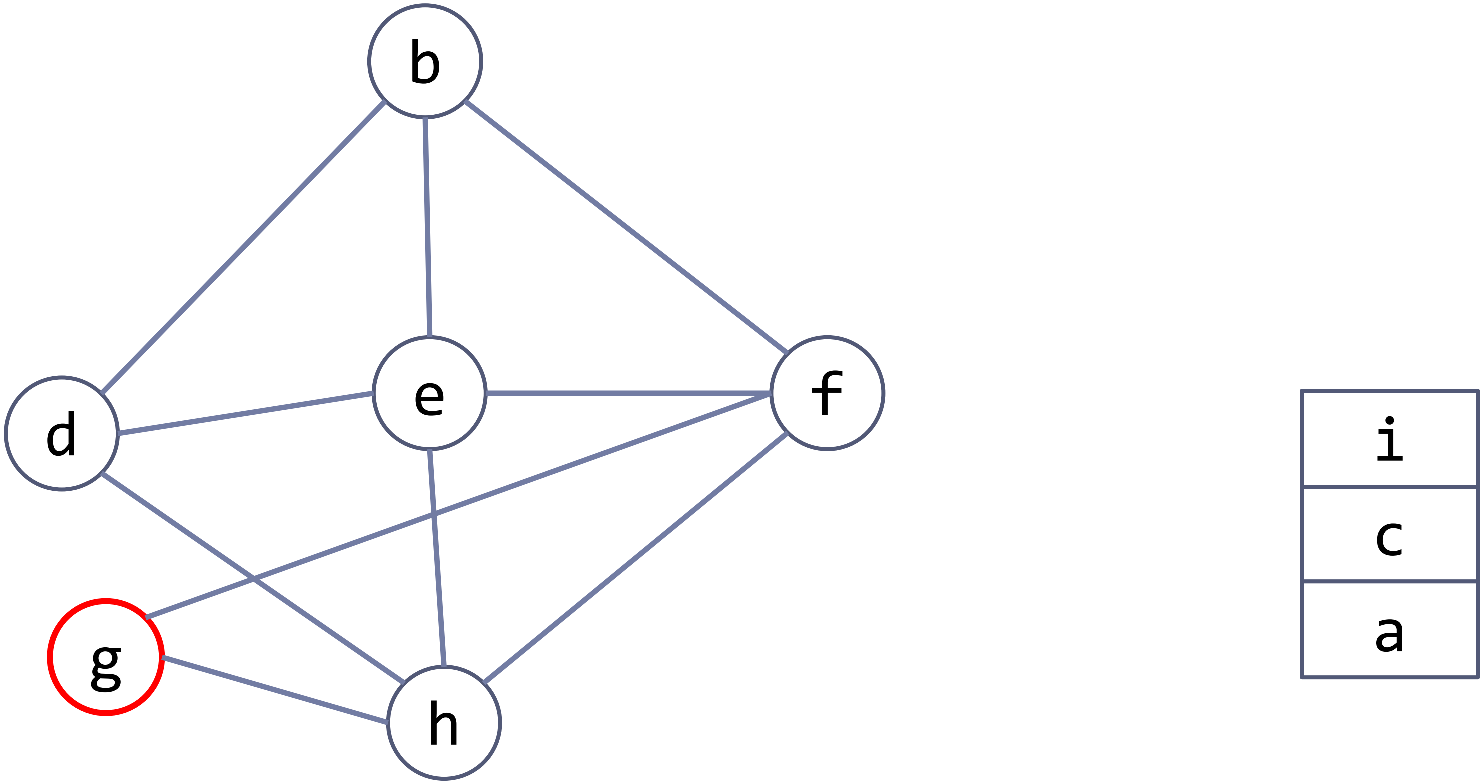
- Remove these nodes and push them onto stack



i
c
a

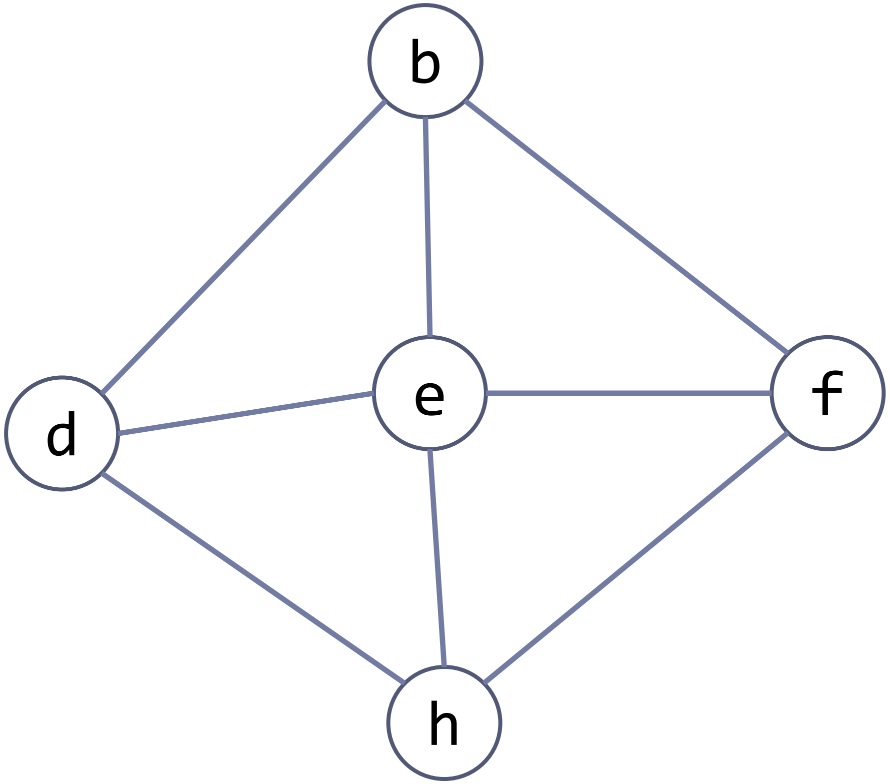
Example

- Find nodes with fewer than three neighbours



Example

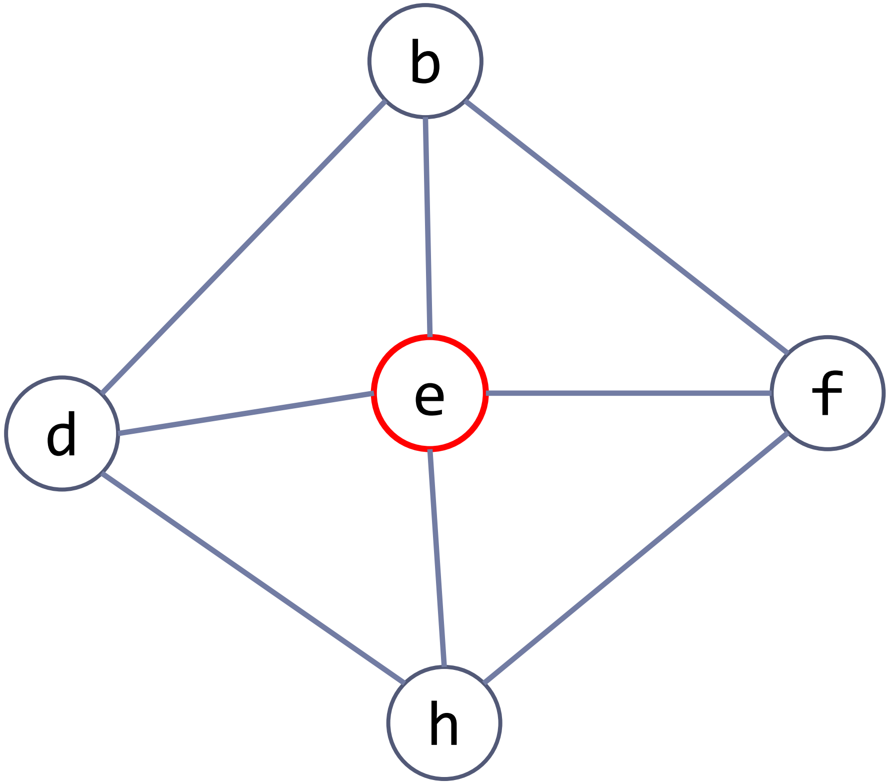
- Remove these nodes and push them onto stack



g
i
c
a

Example

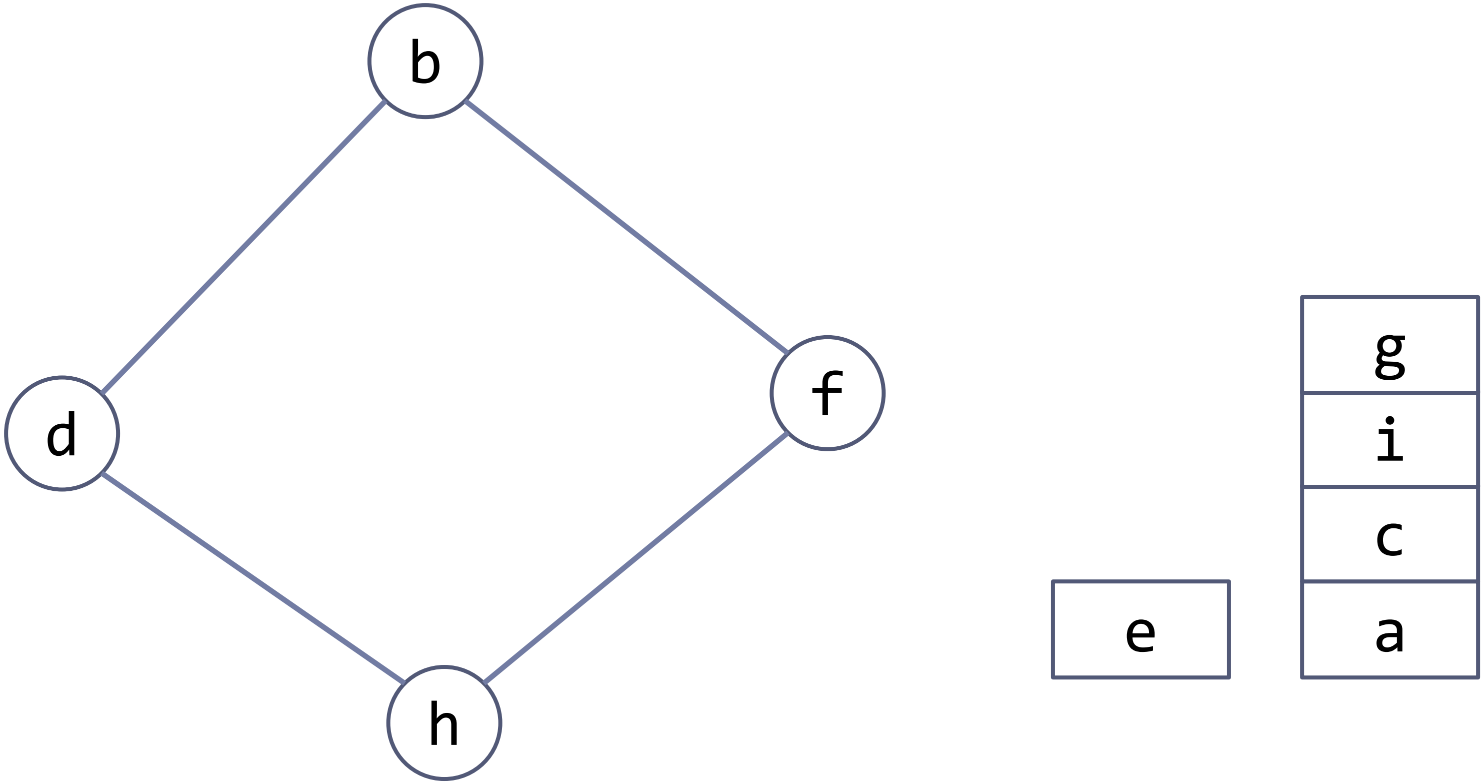
- All remaining nodes have at least three neighbours!
- We select a spill candidate: a node with the greatest number of neighbours



g
i
c
a

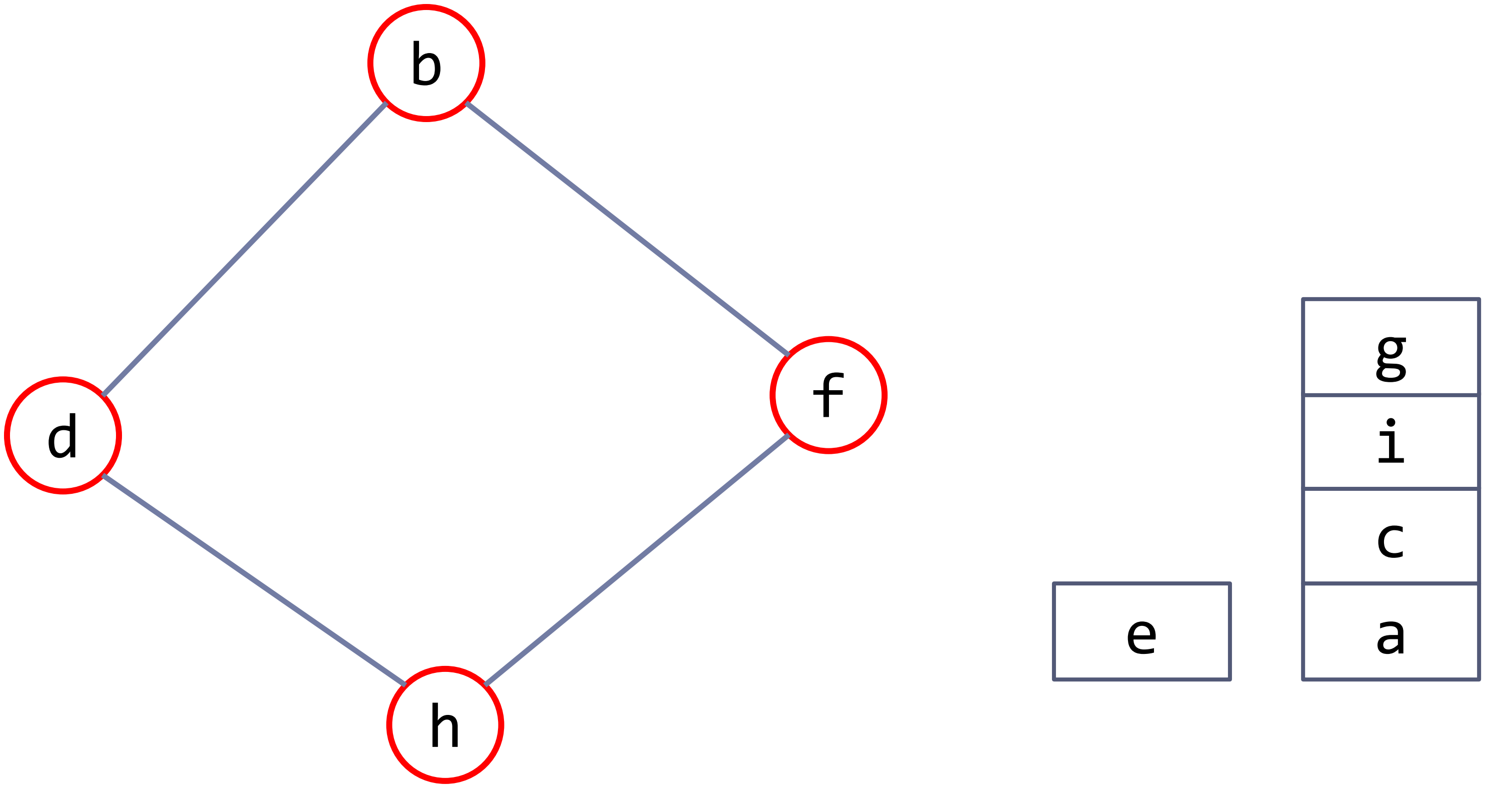
Example

- We remove the spill candidate and push it into the list



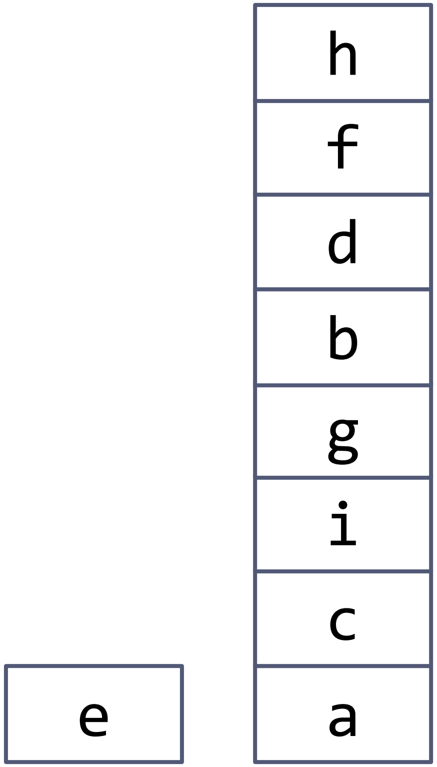
Example

- Find nodes with fewer than three neighbours



Example

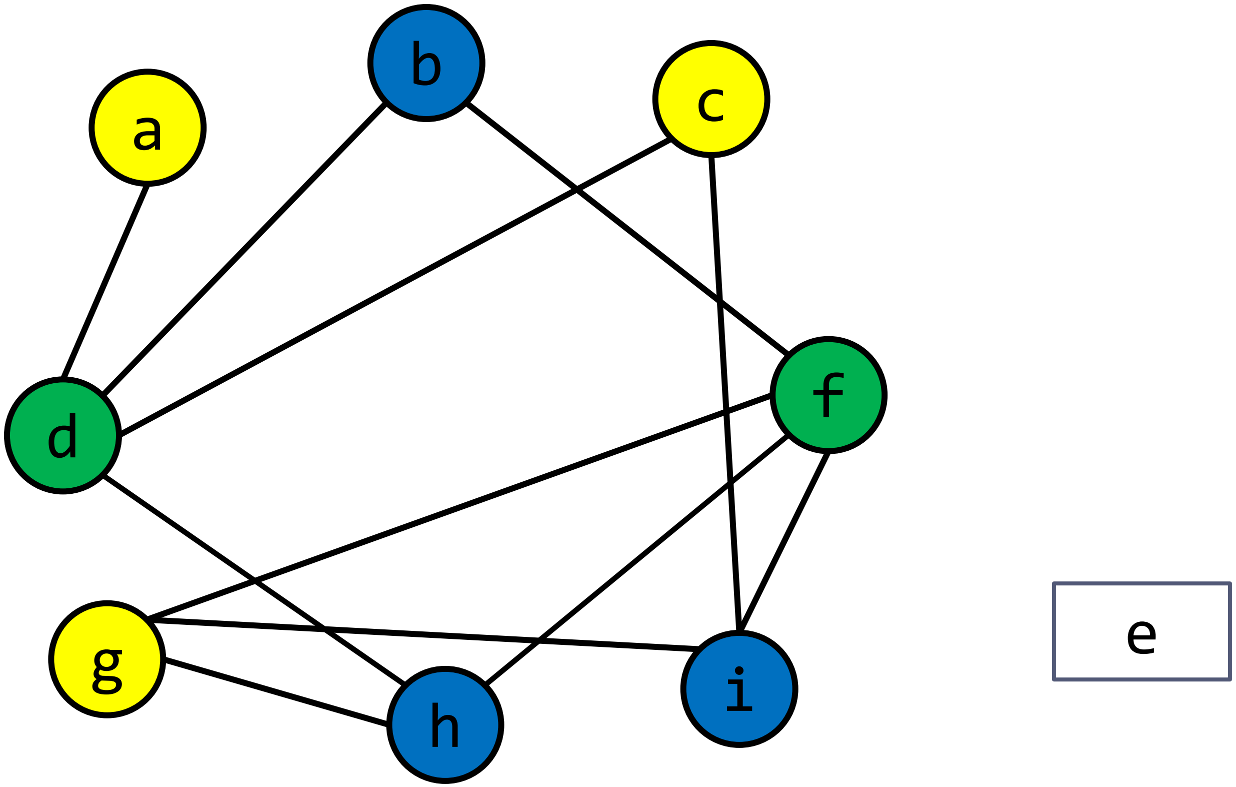
- Remove these nodes and push them onto stack



h
f
d
b
g
i
c
e
a

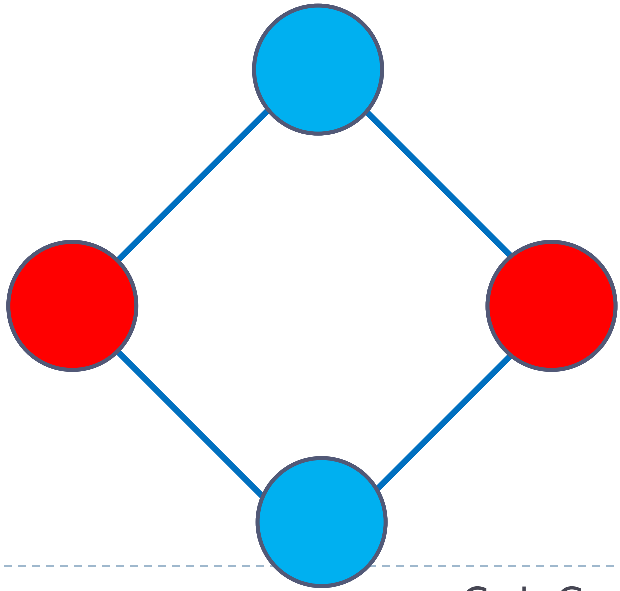
Example

- Pop off each node from the stack and assign a register
- Skip the spill candidate



Optimistic Coloring

- If Chaitins algorithm reaches a state where every node has k or more neighbours, it chooses a node to spill.
  - Chaitines algorithm treats the following graph as 3-colourable
- Briggs: mark the node and push it on the stack
  - When it is popped off from the stack, a colour might be available for it!



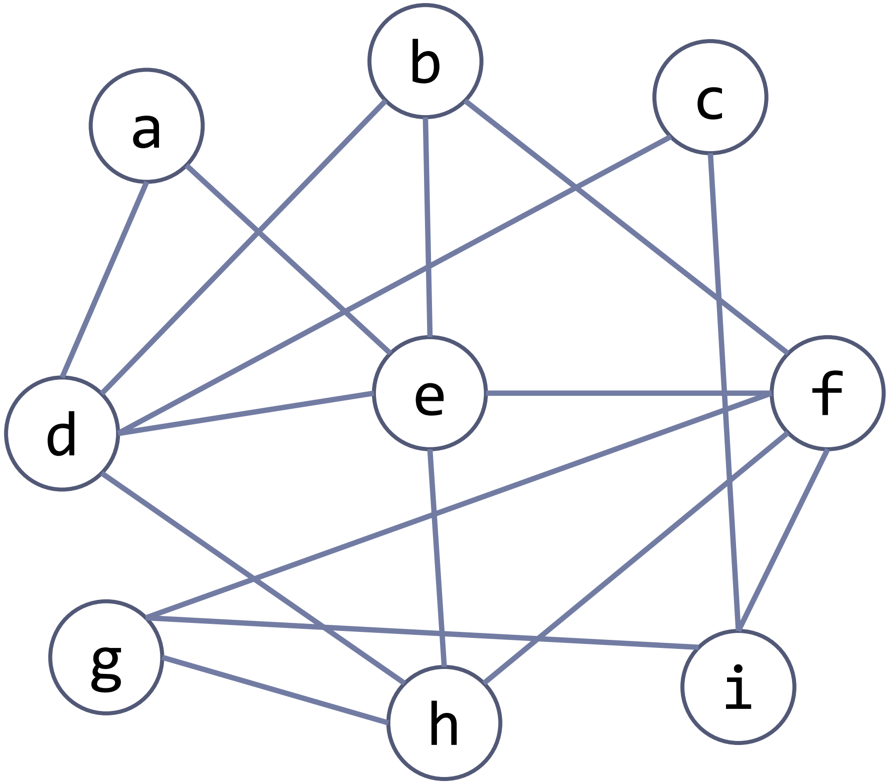
2-colourable

Chaitin-Briggs Algorithm: Pseudocode

```
INPUT: Interference graph $IG$, number $k$ of registers
$s$ = empty stack
while $IG$ not empty do
    if there is node $n$ with neighbours($n$) < $k$ then
        remove $n$ from $IG$ and push it onto $s$
    else
        let $d$ be node with maximum number of neighbours
        remove $d$ from $IG$, mark it as spill candidate and push it onto $s$

while $s$ not empty do
    pop node $n$ off $s$
    if $n$ marked and neighbours($n$) have $k$ different colours then
        do not allocate a register for $n$. Instead, insert the store/load code for $n$.
    else
        allocate $n$ a register not allocated to any of neighbours($n$)
```

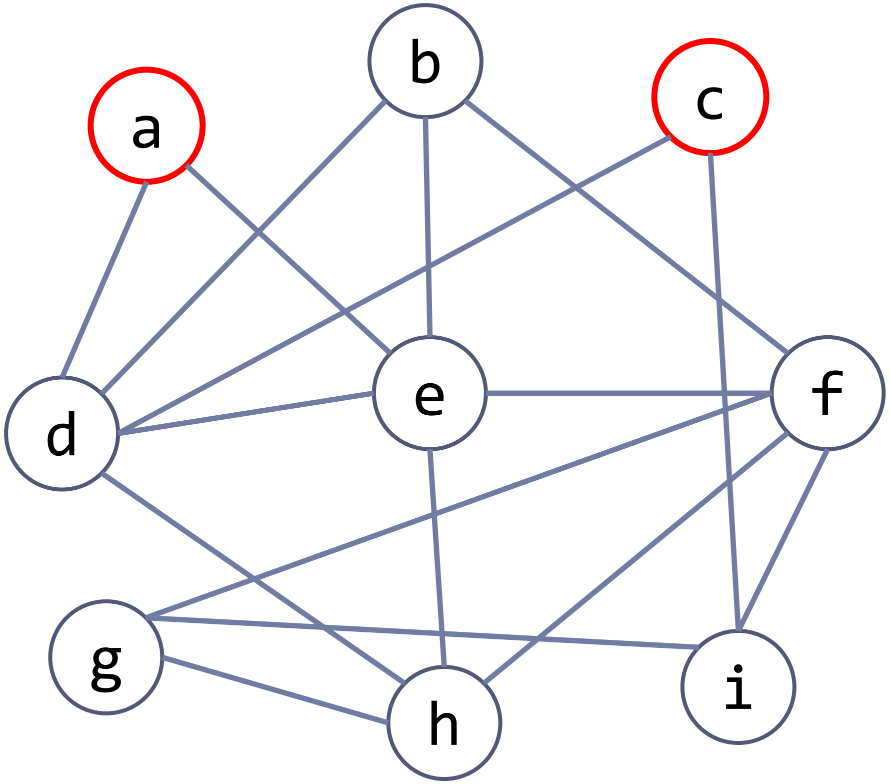
Example

- What if we only have three registers r, s, t?



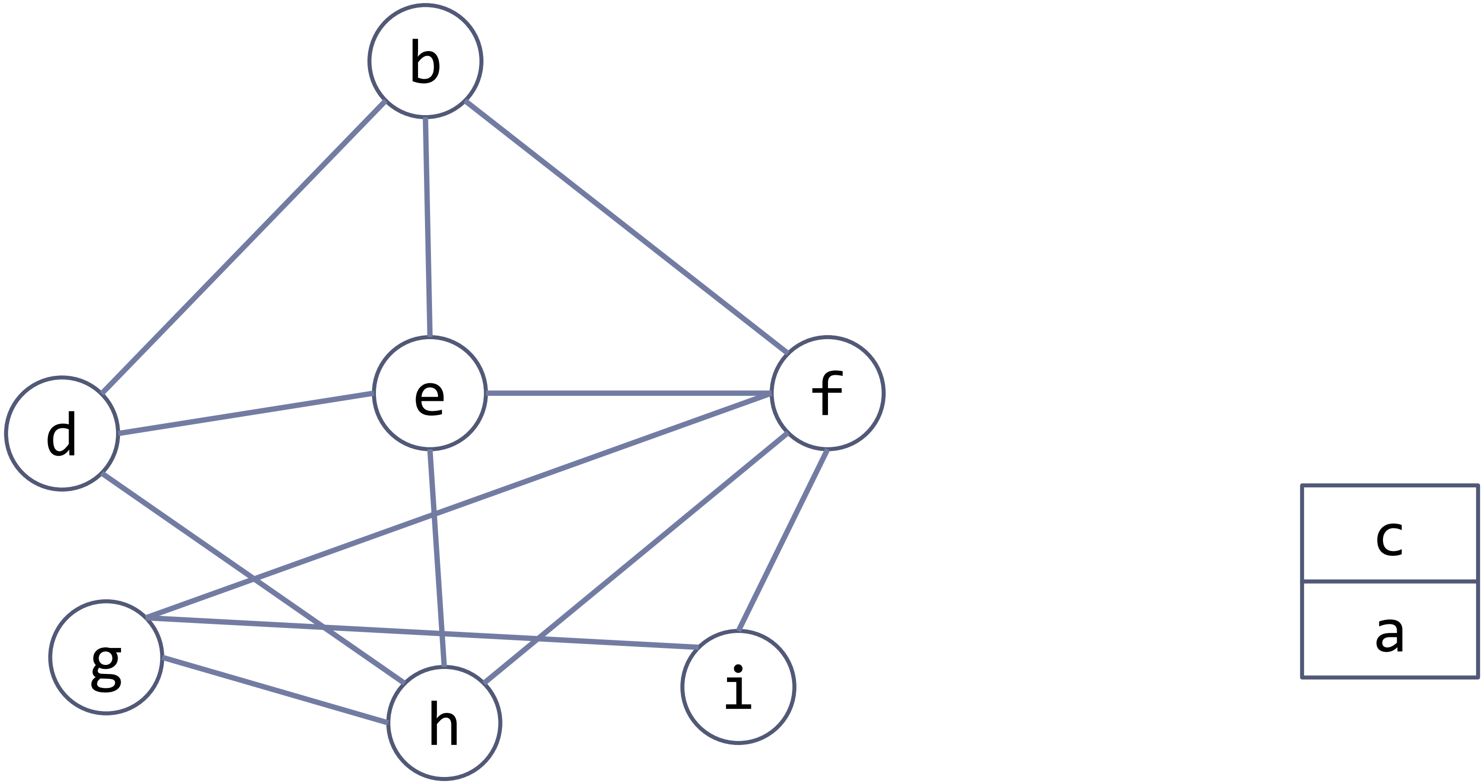
Example

- Find nodes with fewer than three neighbours



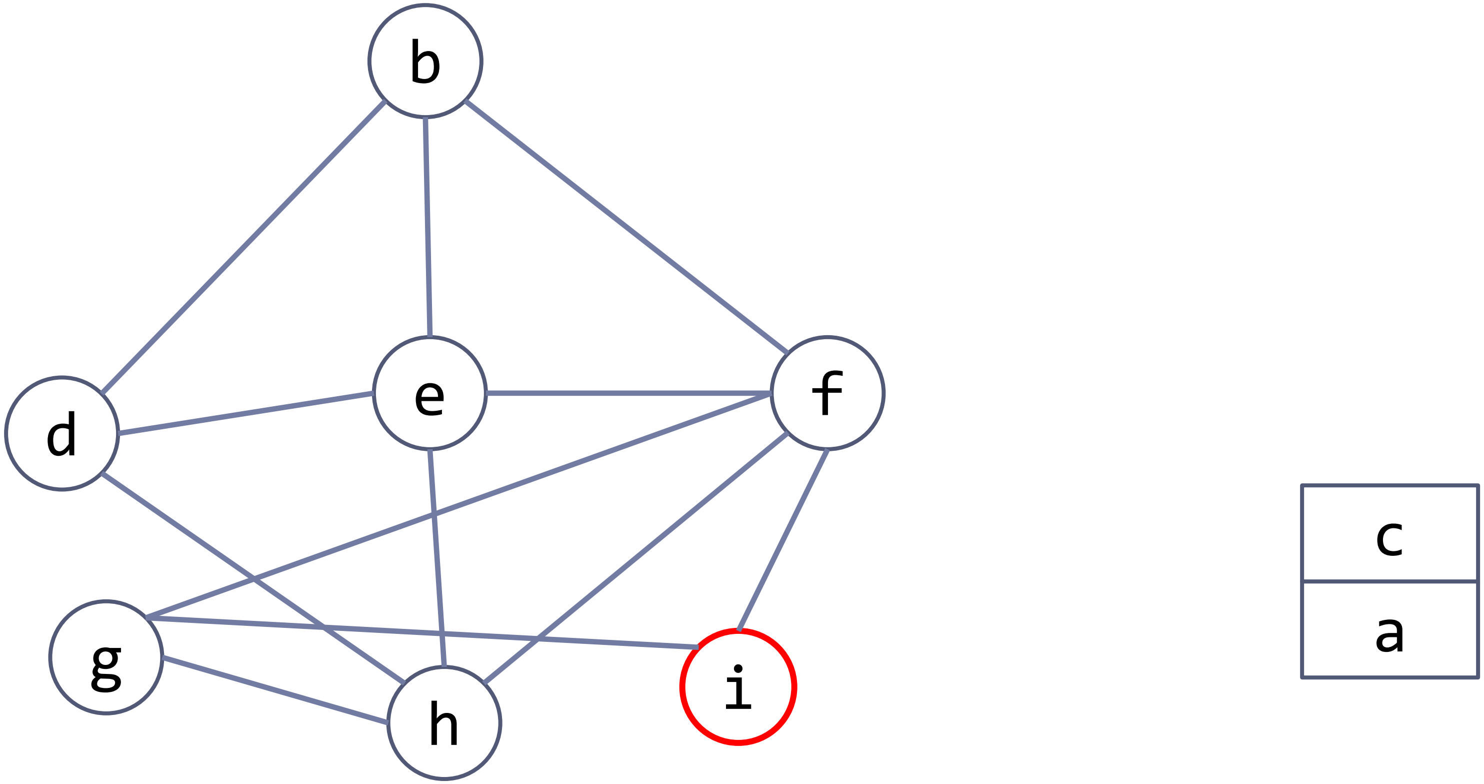
Example

- Remove these nodes and push them onto stack



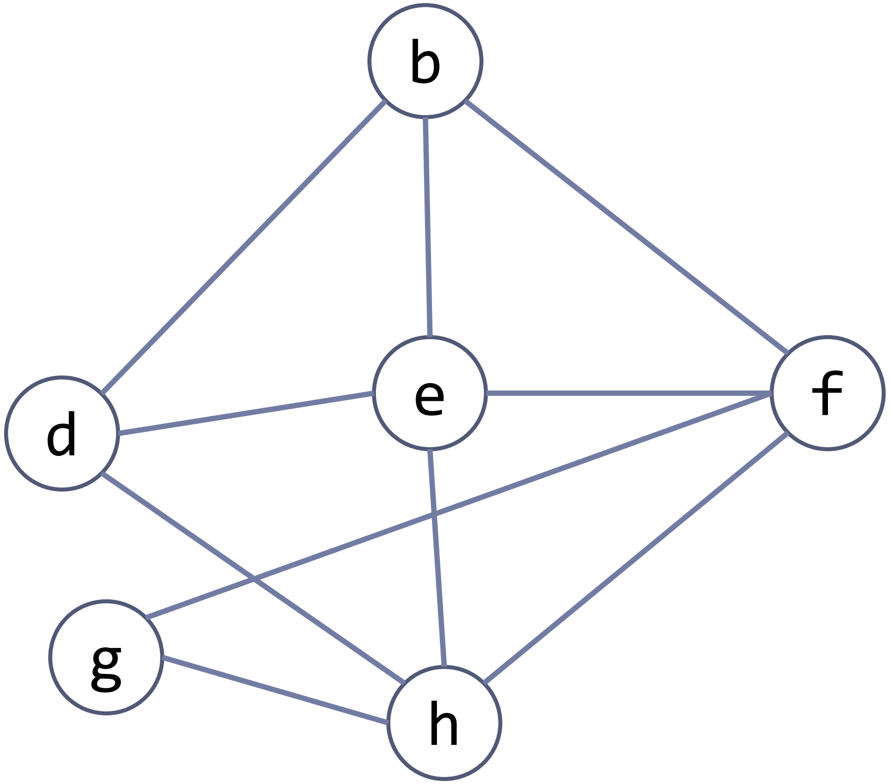
Example

- Find nodes with fewer than three neighbours



Example

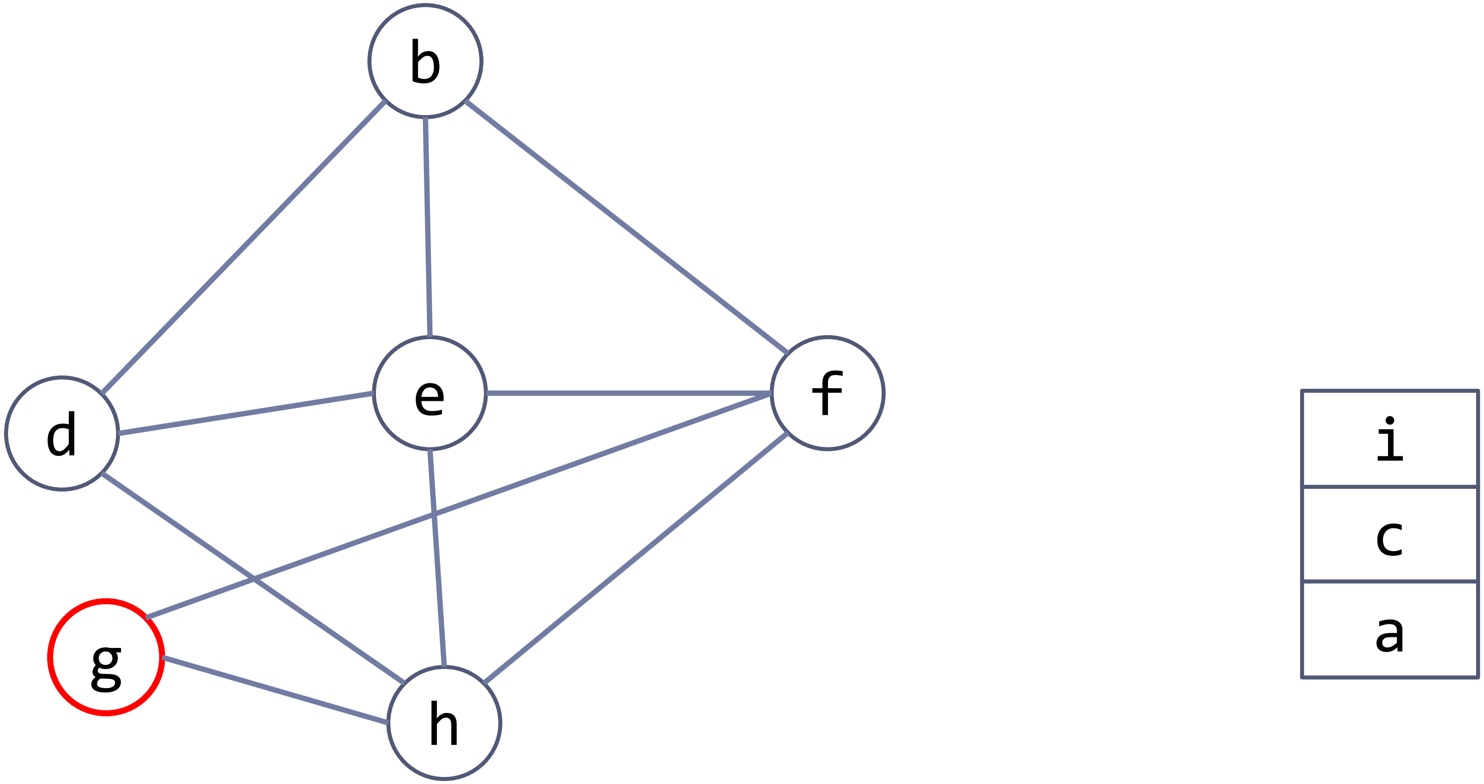
- Remove these nodes and push them onto stack



i
c
a

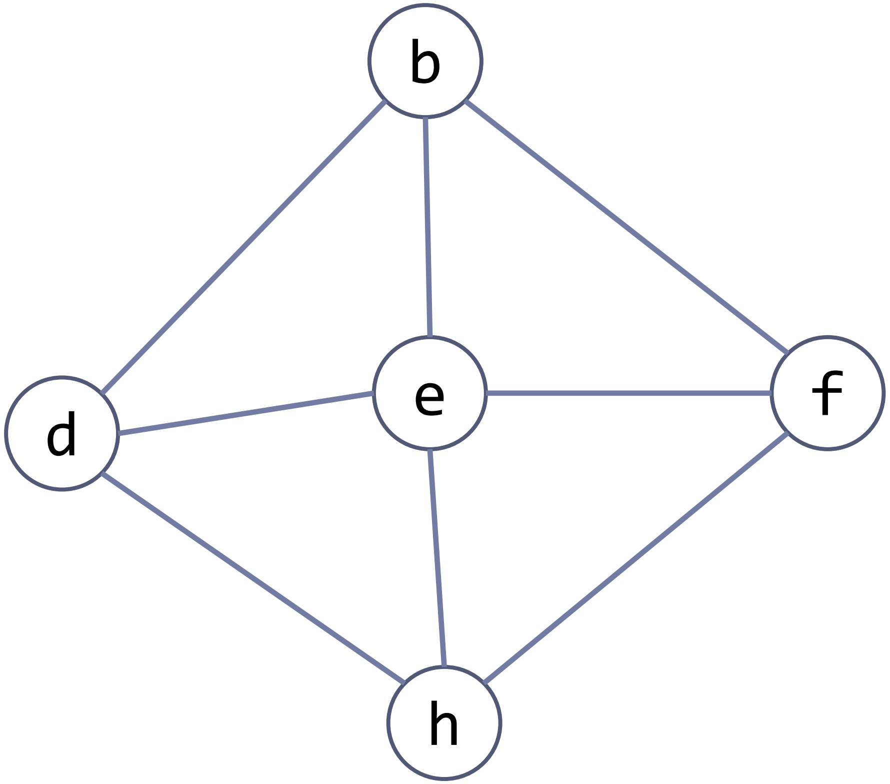
Example

- Find nodes with fewer than three neighbours



Example

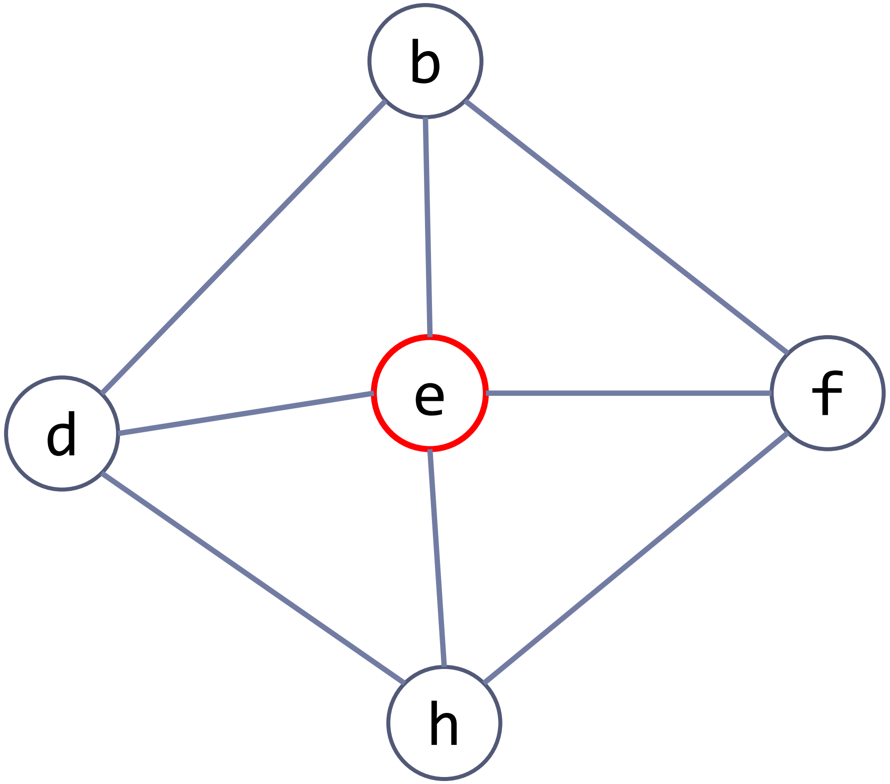
- Remove these nodes and push them onto stack



g
i
c
a

Example

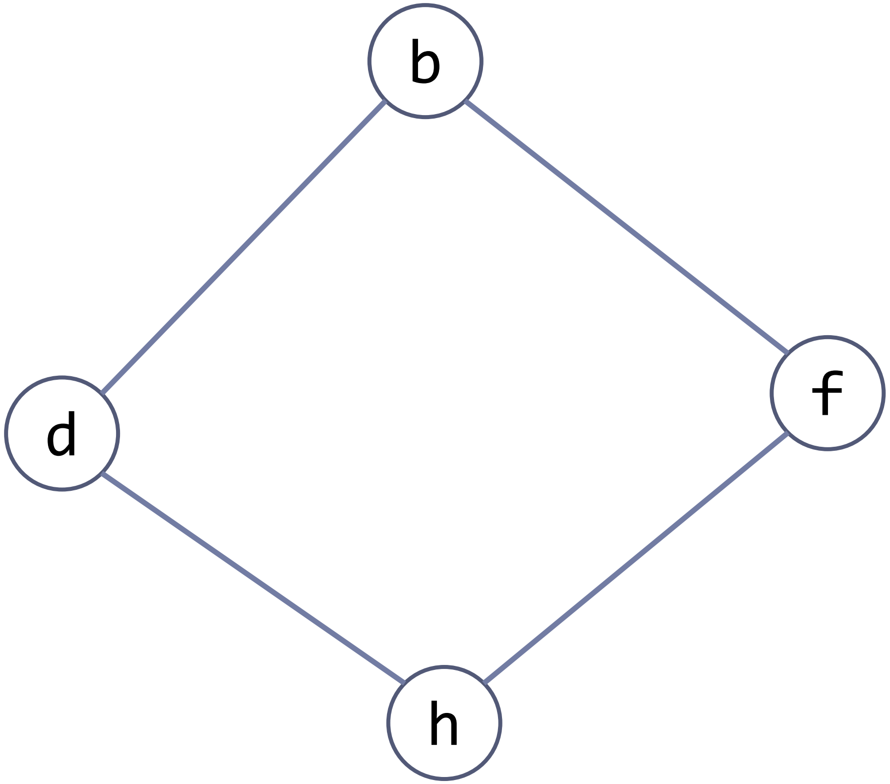
- All remaining nodes have at least three neighbours!
- We select a spill candidate: a node with the greatest number of neighbours



g
i
c
a

Example

- We remove the spill candidate, mark it and push it onto the stack



e*

g

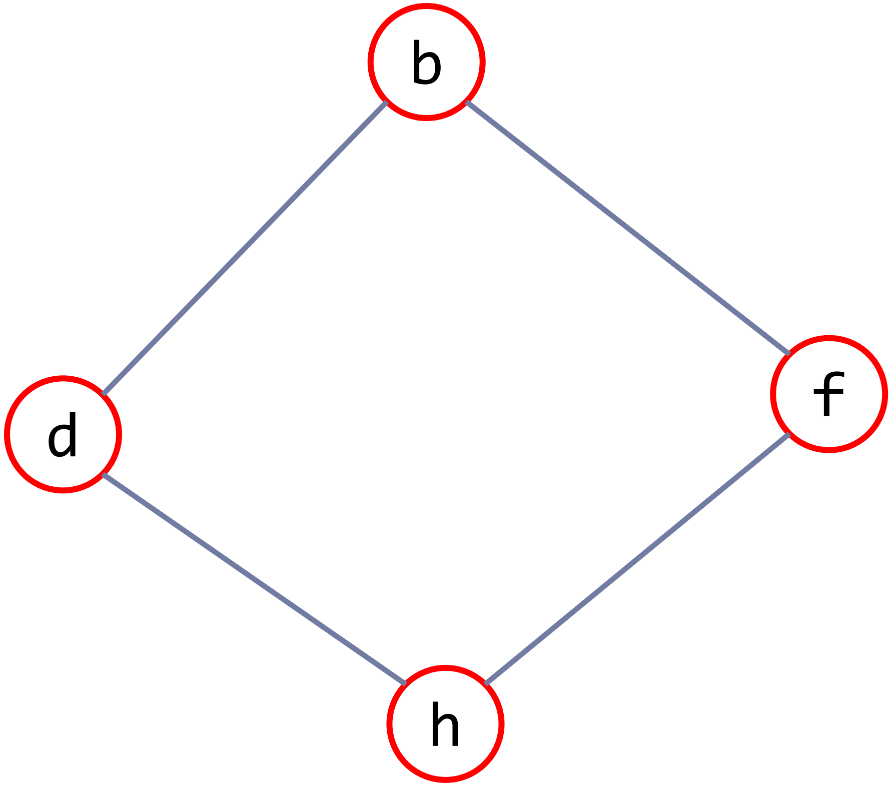
i

c

a

Example

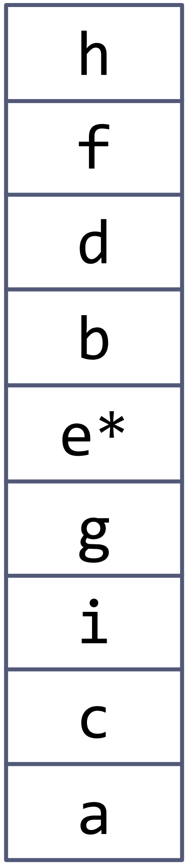
- Find nodes with fewer than three neighbours



e*
g
i
c
a

Example

- Remove these nodes and push them onto stack



h

f

d

b

e*

g

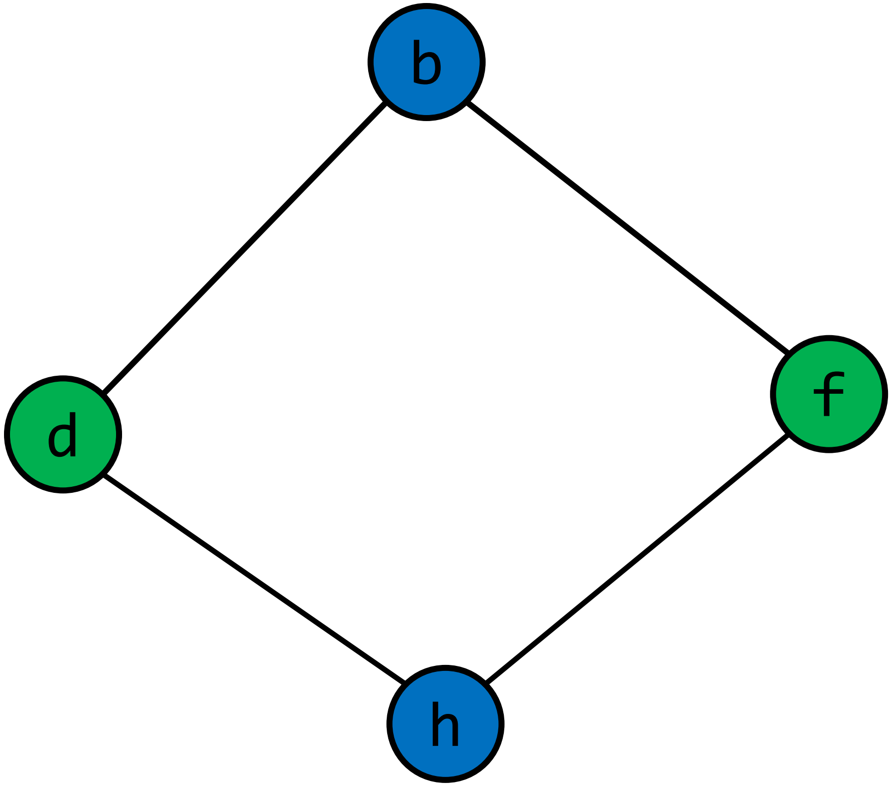
i

c

a

Example

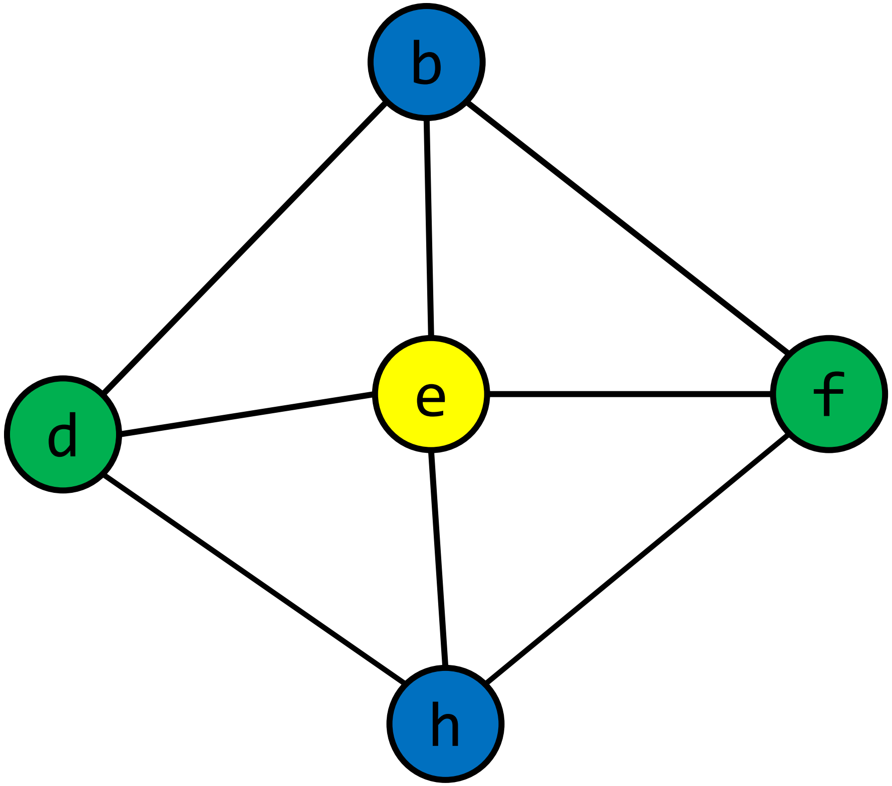
- Pop off each node from the stack and assign a register



e*
g
i
c
a

Example

- We are able to find a colour for the spill candidate
- Pop out e* and allocate a register



g

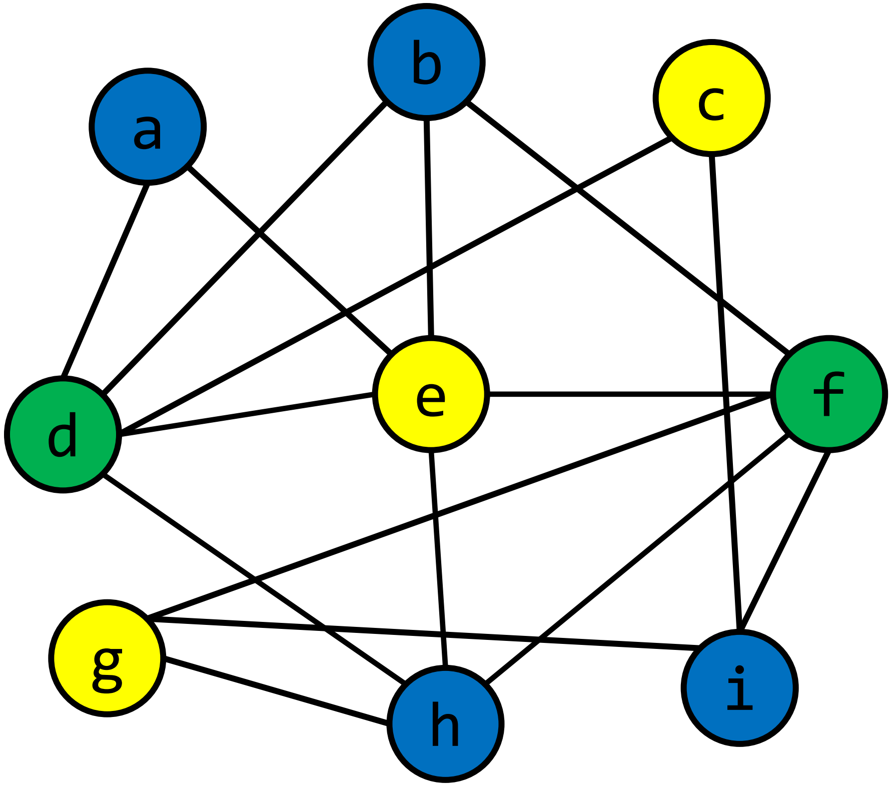
i

c

a

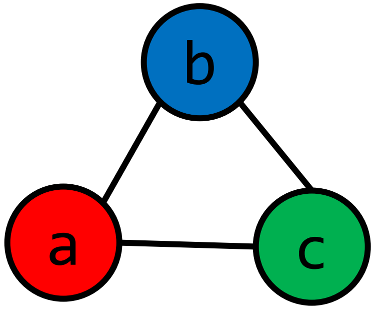
Example

- Pop off the rest nodes from the stack and assign registers

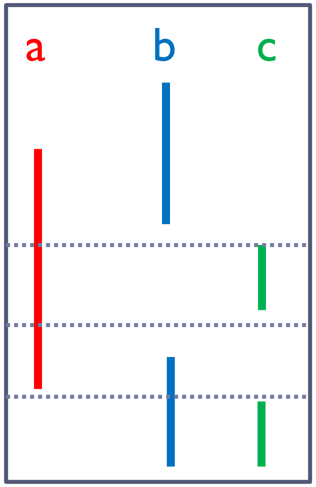


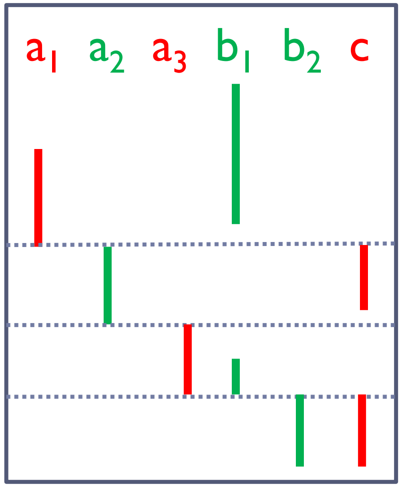
Live Range Splitting

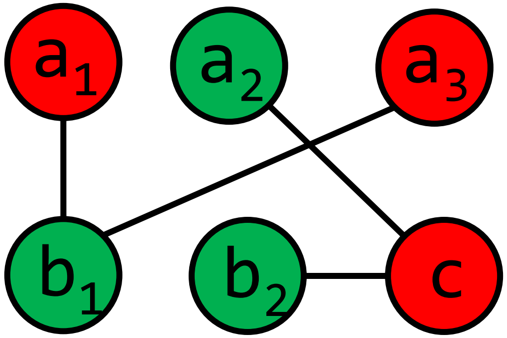
- A variable may have multiple live ranges, with each one having some interferences
- We can split the ranges into multiple variables connected by the copy instruction
  - Possibly reduce the degree of interference graph



3-colourable



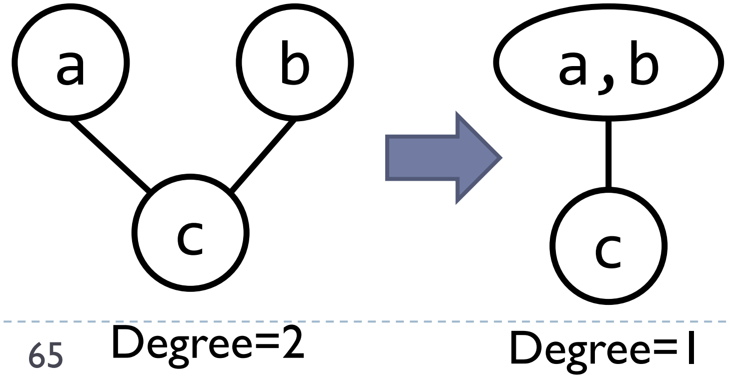


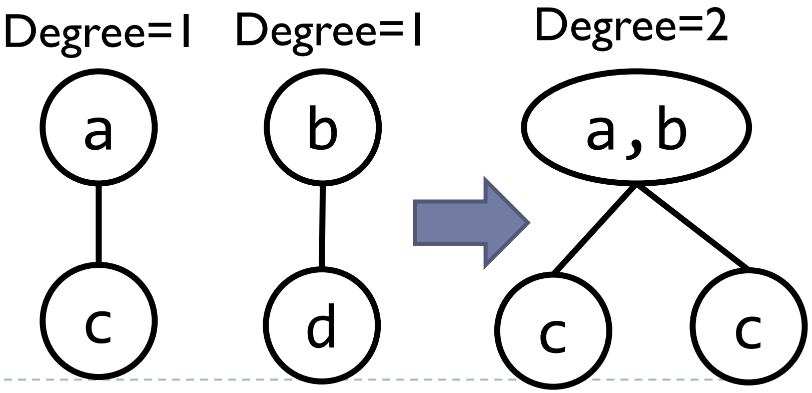


2-colourable

Live Range Coalescing

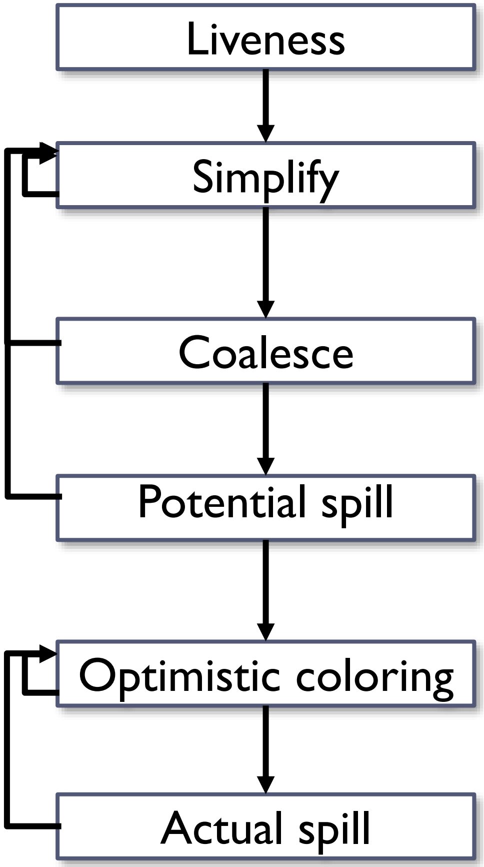
- If two ranges do not interfere, and connected by a copy instruction, we can coalesce the two variables into one.
  - Reduce the degree of nodes that interfered with both
  - Eliminate the copy instruction
- Coalescing can make the graph harder to colour
  - We perform coalescing only when the degree of the coalesced node is still smaller than k.





The Overall Picture of Register Allocation

- Liveness analysis for interference graph
- Simplification
  - Simplify the nodes in the graph
  - Coalesce possible nodes
  - Select potential spills
- Coloring
  - Perform optimistic coloring
  - Insert code to implement the actual spill


Liveness
Simplify
Coalesce
Potential spill
Optimistic coloring
Actual spill