CS 4240: Compilers

Lecture 12: Register Allocation

Guest Lecturer: Chris Porter
(cporter35@gatech.edu)

February 20, 2019

Happy International Pipe Day

https://www.youtube.com/watch?v=80oLTiVW_lc

Register allocation

- Assume a 3-address code intermediate representation with an unbounded number of virtual/symbolic registers
- For each virtual register **r**, want to choose a physical machine register to store **r**'s values
- Intermediate step: determine the *live ranges* of each virtual register **r**

Register Allocation: Motivation

Option #1:

Keep in Memory

```
load r1, 4(sp)
load r2, 8(sp)
add r3,r1,r2
store r3, 12(sp)
```

Option #2:

Keep in Registers

```
add r3,r1,r2
```

- Advantages to Option 2:
  - Uses fewer instructions: most instrs are reg ⇔ reg
  - Each instruction is cheaper: accessing memory is expensive

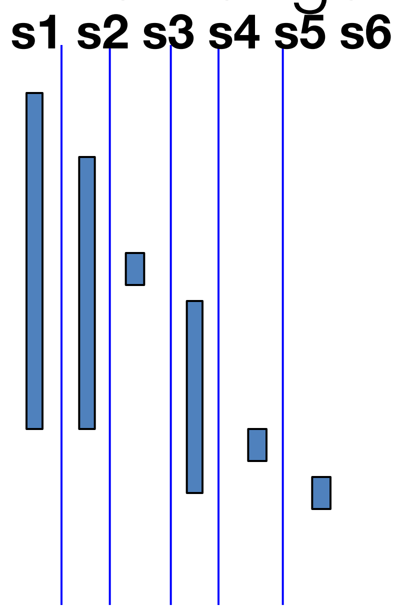
Register Allocation

- Determine which of the values (variables, temporaries, large constants) should be in registers at each program point
  - Processors have very few registers, but they can be accessed very quickly
- Goal: **minimize** the traffic between the CPU registers and the memory hierarchy
  - In practice, often the optimization with the **greatest** impact on performance

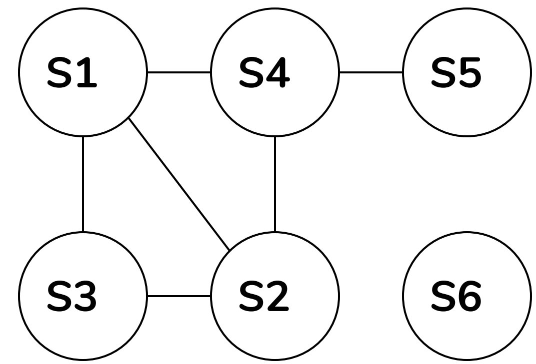
A Simple Example

```
s1 ← 2
s2 ←  4
s3 ← s1 + s2
s4 ← s3 + 1
s5 ← s1 * s2
s6 ← s4 * 2
```

Live Range



Interference



- **Live**: variable will be used before it is overwritten
- **Dead**: variable will be overwritten before it is used

A Simple Example (contd)

Intermediate Code w/ Symbolic Registers

```
s1 ← 2
s2 ← 4
s3 ← s1 + s2
s4 ← s3 + 1
s5 ← s1 * s2
s6 ← s4 * 2
```

Machine Code w/ Physical Registers

```
r1 ← 2
r2 ← 4
r3 ← r1 + r2
r3 ← r3 + 1
r1 ← r1 * r2
r2 ← r3 * 2
```

Global Register Allocation

The big picture

Optimal global allocation is NP-Complete, under almost any assumptions.

At each point in the code

1. Determine which values will reside in registers
2. Select a register for each such value

The goal is an allocation that "minimizes" running time

Most modern, global allocators use a graph-coloring paradigm

- Build a "conflict graph" or "interference graph"
- Find a k-coloring for the graph, or change the code to a nearby problem that it can k-color

Building the Interference Graph

What is an "interference"? (or conflict)

- Two values interfere if there exists an operation where both are simultaneously live
- If x and y interfere, they cannot occupy the same register

To compute interferences, we must know where values are "live"

The interference graph, $G_I = (N_I, E_I)$

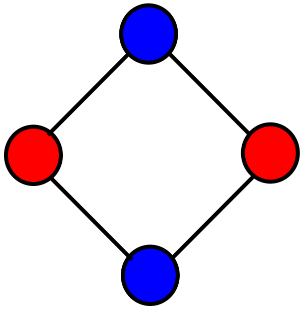
- Nodes in G_I represent values, or live ranges
- Edges in G_I represent individual interferences
  - For $x, y \in N_I$, $\langle x,y \rangle \in E_I$ iff x and y interfere
- A k-coloring of G_I can be mapped into an allocation to k registers

Graph Coloring

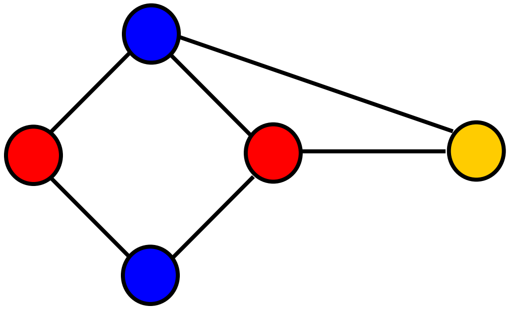
The problem

A graph G is said to be k-colorable iff the nodes can be labeled with integers 1… k so that no edge in G connects two nodes with the same label

Examples



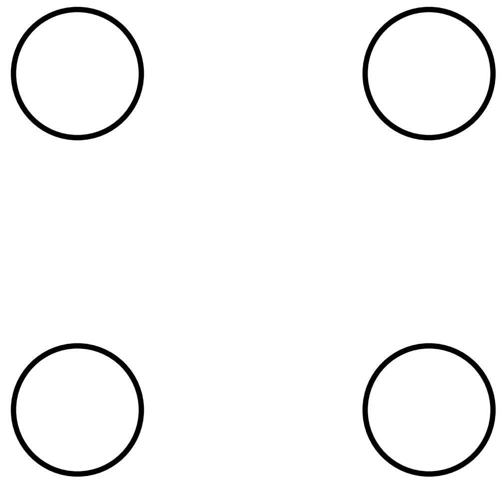
2-colorable



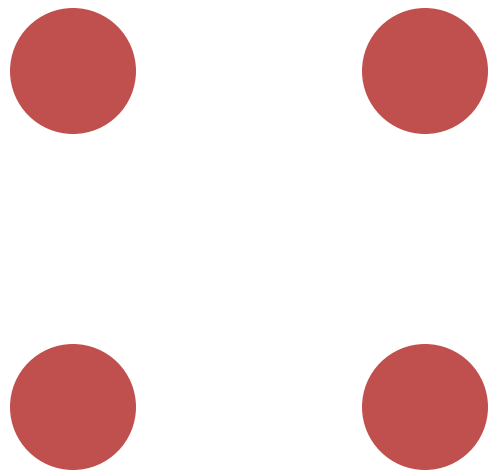
3-colorable

Each color can be mapped to a distinct physical register

Graph Coloring Example #1

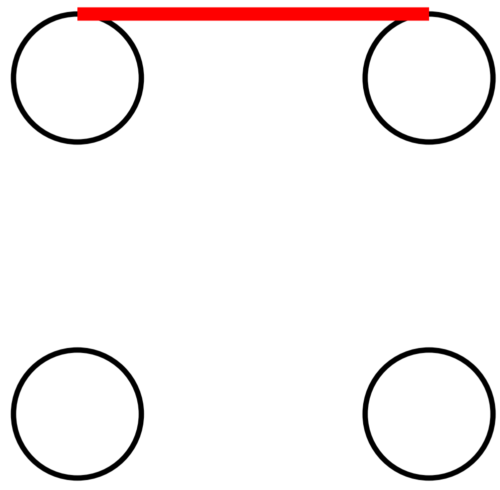


Graph Coloring Example #1

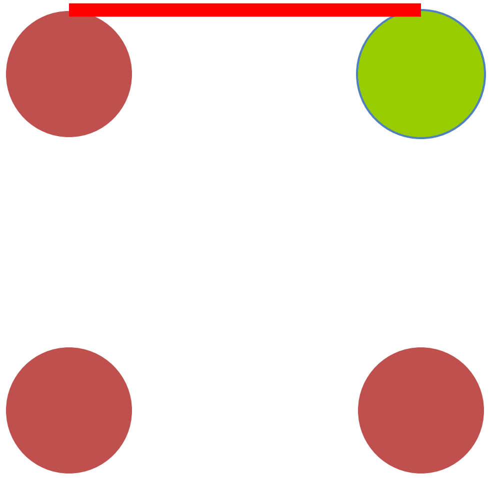


Colors: **1**

Graph Coloring Example #2

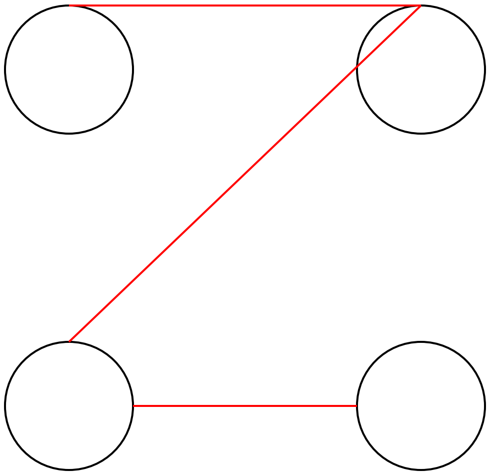


Graph Coloring Example #2

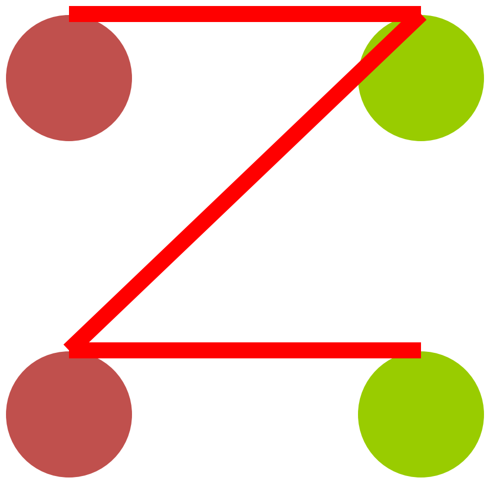


Colors: **2**

Graph Coloring Example #3

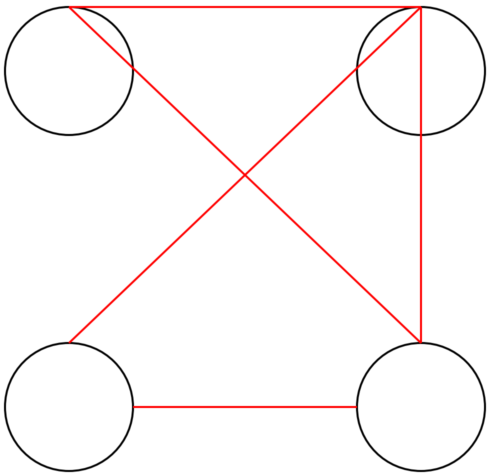


Graph Coloring Example #3

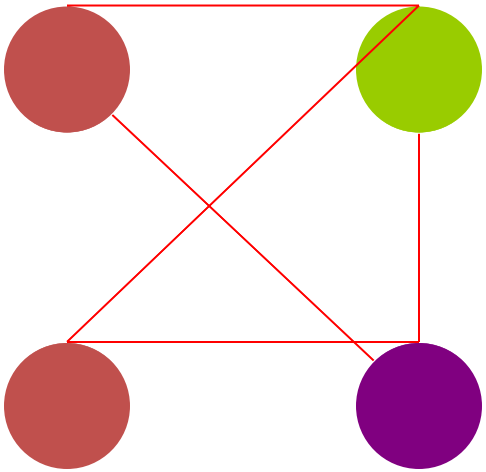


Colors: **still 2**

Graph Coloring Example #4



Graph Coloring Example #4



Colors: **3**

Computing LIVE Sets (live ranges)

A value v is live at program point p if ∃ a path from p to some use of v along which v is not re-defined

Data-flow problems are expressed as simultaneous equations

$$\mathrm{LIVEOUT}(b) = \bigcup_{s \in succ(b)} \mathrm{LIVEIN}(s)$$

$$\mathrm{LIVEIN}(b) = \mathrm{UEVAR}(b) \cup (\mathrm{LIVEOUT}(b) - \mathrm{VARKILL}(b))$$

where

UEVAR(b) is the set of names used in block b before being defined in b (Upwards Exposed Variables)

VARKILL(b) is the set of variables assigned in b

Solve the equations using a fixed-point iterative scheme

Computing LIVE Sets

The compiler can solve these equations with an iterative algorithm

```
WorkList ← { all blocks }
while ( WorkList ≠ Ø)
   remove a block b from WorkList
   Compute LIVEOUT(b)
   Compute LIVEIN(b)
   if LIVEIN(b) changed
      then add pred (b) to WorkList
```

The Worklist Iterative Algorithm

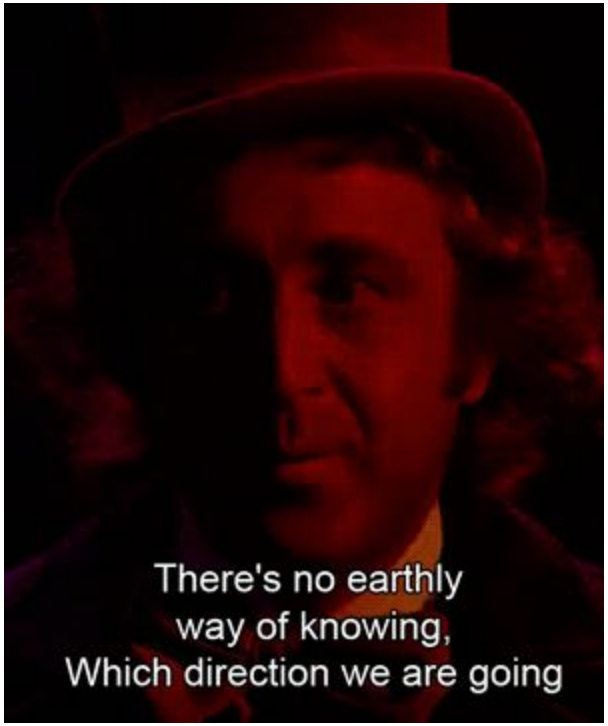
The world's quickest introduction to data-flow analysis !

Computing LIVE Sets

The compiler can solve these equations with an iterative algorithm

```
WorkList ← { all blocks }
while ( WorkList ≠ Ø)
   remove a block b from WorkList
   Compute LIVEOUT(b)
   Compute LIVEIN(b)
   if LIVEIN(b) changed
      then add pred (b) to WorkList
```

The Worklist Iterative Algorithm



The world's quickest introduction to data-flow analysis !

Computing LIVE Sets

The compiler can solve these equations with an iterative algorithm

```
WorkList ← { all blocks }
while ( WorkList ≠ Ø)
   remove a block b from WorkList
   Compute LIVEOUT(b)
   Compute LIVEIN(b)
   if LIVEIN(b) changed
      then add pred (b) to WorkList
```

The Worklist Iterative Algorithm

Why does this work?

- LIVEOUT, LIVEIN $\subseteq 2^{Names}$
- UEVAR, VARKILL are constants for b
- Equations are monotone
- Finite # of additions to sets

⇒ will reach a fixed point !

Speed of convergence depends on the order in which blocks are "removed" & their sets recomputed

The world's quickest introduction to data-flow

Observation on Coloring for Register Allocation

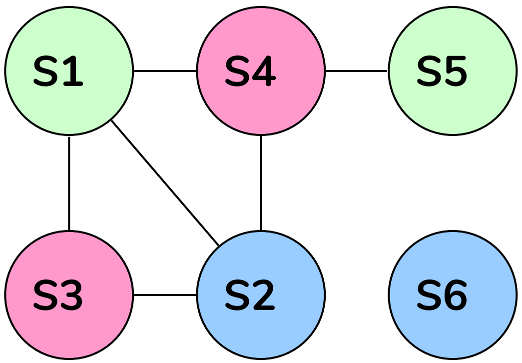
- Suppose you have k registers—look for a k coloring
- Any vertex n that has fewer than k neighbors in the interference graph ($n^\circ < k$) can always be colored!
  - Pick any color not used by its neighbors — there must be one
- Ideas behind a classical algorithm due to Chaitin:
  - Pick any vertex n such that $n^\circ < k$ and put it on the stack
  - Remove that vertex and all edges incident from the interference graph
    - → This may make additional nodes have fewer than k neighbors
  - At the end, if some vertex n still has k or more neighbors, then spill the live range associated with n
  - Otherwise successively pop vertices off the stack and color them in the lowest color not used by some neighbor

Allocation via Graph Coloring

Intermediate Code

```
s1 ← 2
s2 ← 4
s3 ← s1 + s2
s4 ← s3 + 1
s5 ← s1 * s2
s6 ← s4 * 2
```

Graph Coloring



r1 ← green
r2 ← blue
r3 ← pink

Machine Code

```
r1 ← 2
r2 ← 4
r3 ← r1 + r2
r3 ← r3 + 1
r1 ← r1 * r2
r2 ← r3 * 2
```

Register Allocation

1. Determine live ranges for each symbolic register
2. Determine overlapping ranges (**interference**)
3. Compute the benefit of keeping each love range in a register (**spill cost**)
4. Try to assign each live range to a machine register (**allocation**).
   If needed, **spill** or **split** live range
5. Generate code, including spills

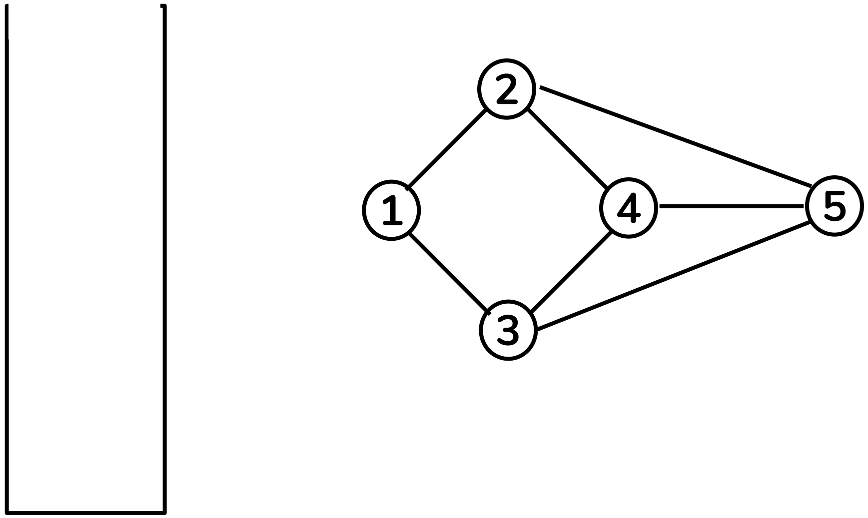
Chaitin's Algorithm

1. While ∃ vertices with < k neighbors in G_I
   - Pick any vertex n such that n°< k and put it on the stack
   - Remove that vertex and all edges incident to it from G_I
     1. This will lower the degree of n's neighbors
2. If G_I is non-empty (all vertices have k or more neighbors) then:
   - Pick a vertex n (using some heuristic) and spill the live range associated with n
   1. Remove vertex n from G_I, along with all edges incident to it and put it on the stack
   - If this causes some vertex in G_I to have fewer than k neighbors, then go to step 1; otherwise, repeat step 2

- Successively pop vertices off the stack and color them in the lowest color not used by some neighbor

Chaitin's Algorithm in Practice

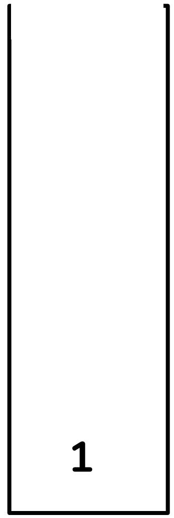
3 Registers



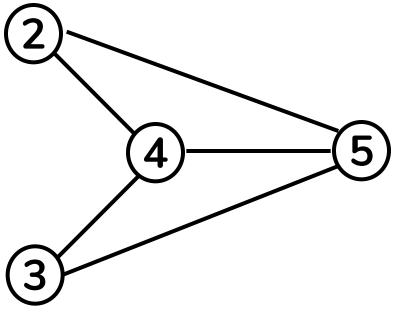
Stack

Chaitin's Algorithm in Practice

3 Registers

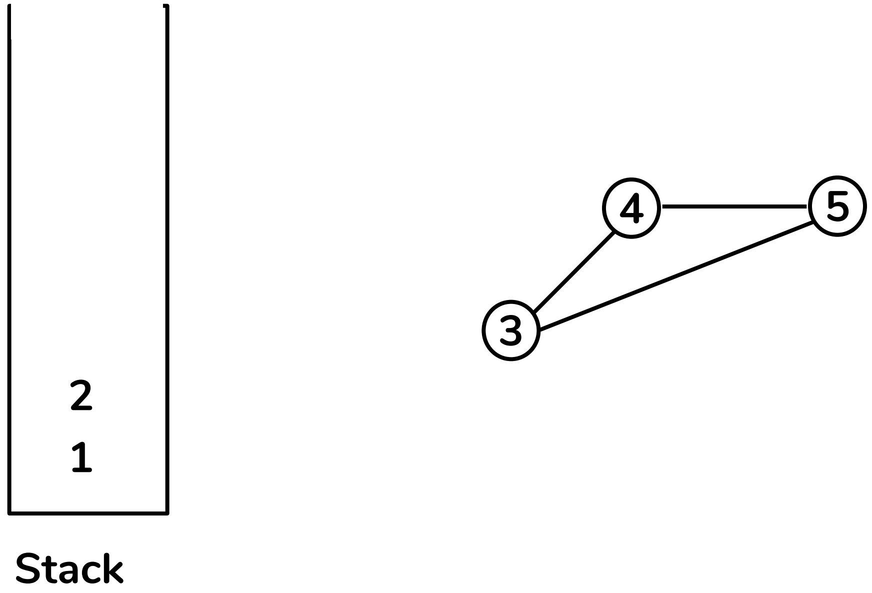


Stack



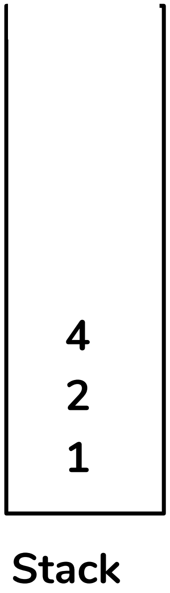
Chaitin's Algorithm in Practice

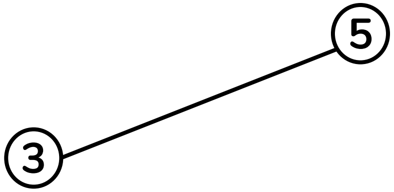
3 Registers



Chaitin's Algorithm in Practice

3 Registers





Chaitin's Algorithm in Practice

3 Registers

5
3
4
2
1

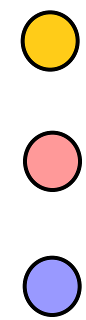
Stack

Colors:

1:

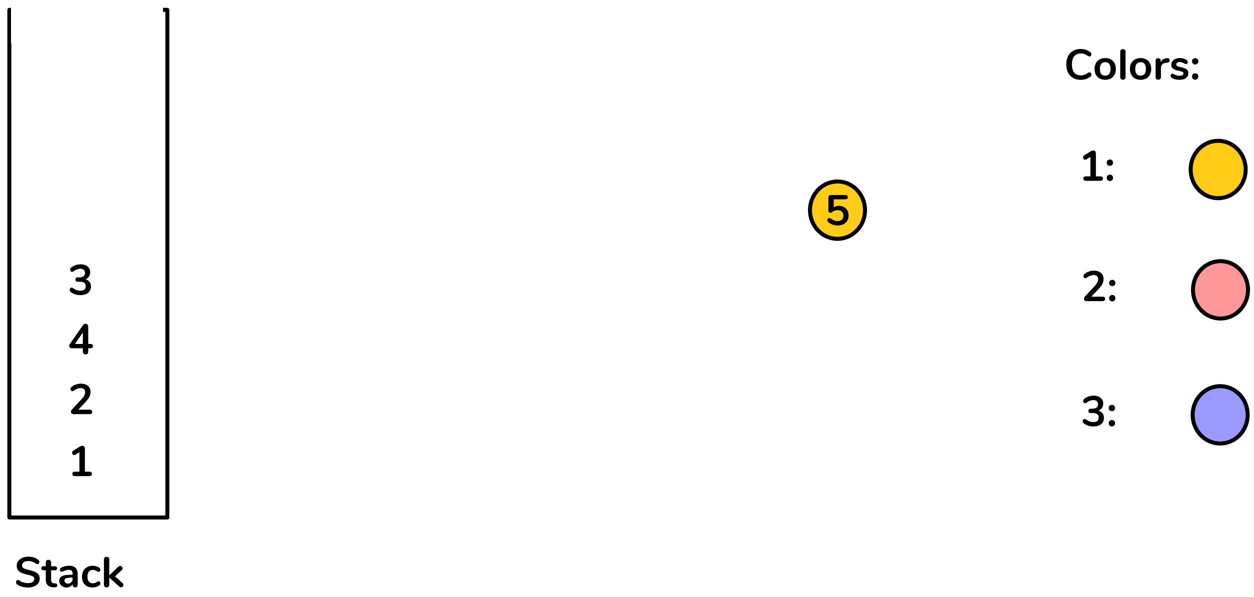
2:

3:



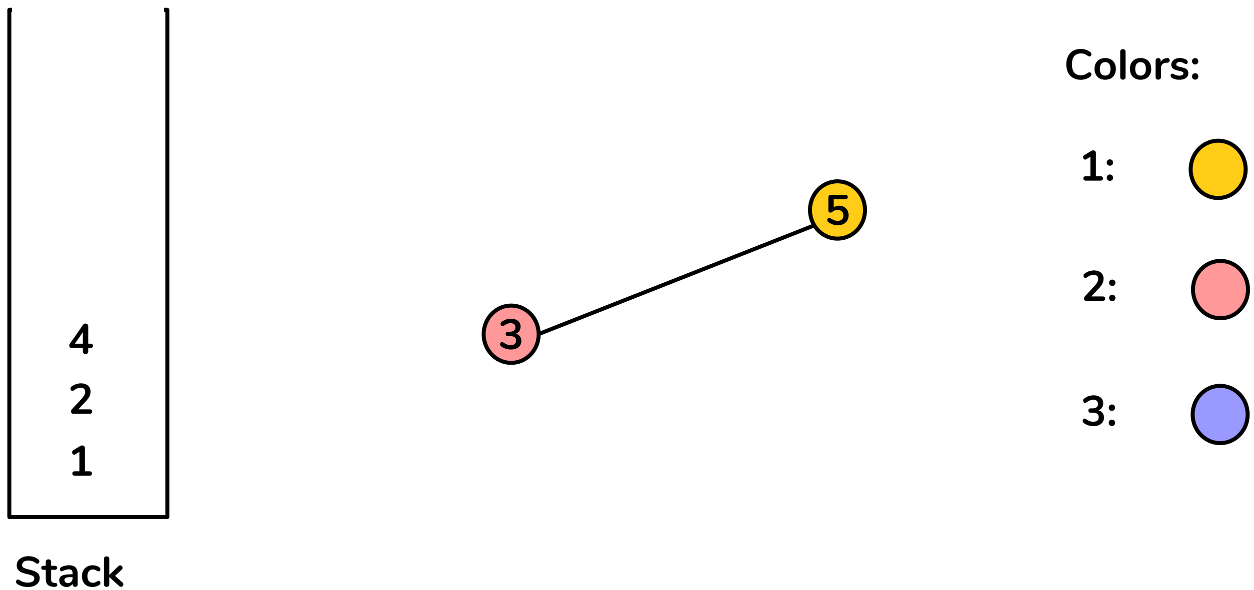
Chaitin's Algorithm in Practice

3 Registers



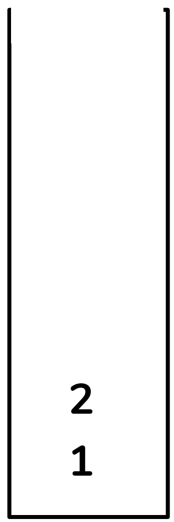
Chaitin's Algorithm in Practice

3 Registers

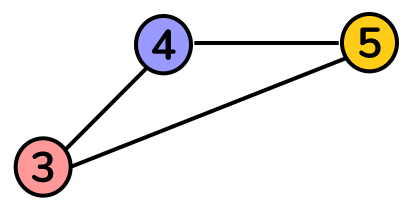


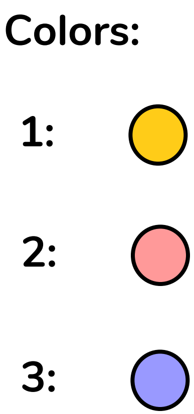
Chaitin's Algorithm in Practice

3 Registers



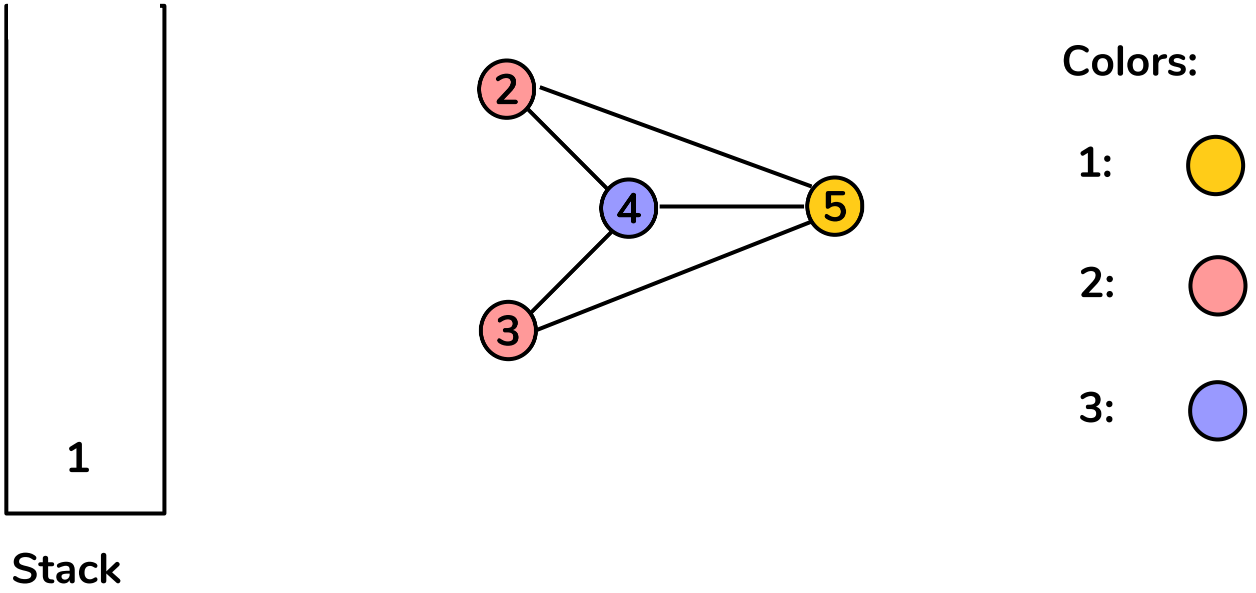
Stack





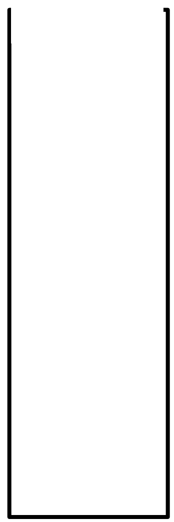
Chaitin's Algorithm in Practice

3 Registers

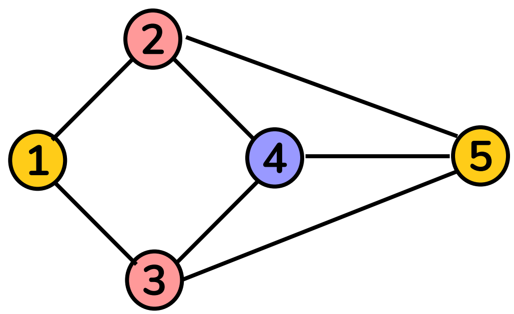


Chaitin's Algorithm in Practice

3 Registers



Stack



Colors:

1:

2:

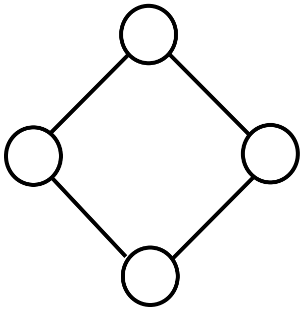
3:

Improvement in Coloring Scheme

Optimistic Coloring (Briggs, Cooper, Kennedy, and Torczon)

- If Chaitin's algorithm reaches a state where every node has k or more neighbors, it chooses a node to spill.
- Briggs said, take that same node and push it on the stack
  - When you pop it off, a color might be available for it!

2 Registers:



Chaitin's algorithm immediately spills one of these nodes

  - For example, a node n might have k+2 neighbors, but those neighbors might only use 3 (<k) colors
    - → Degree is a <u>loose upper bound</u> on colorability

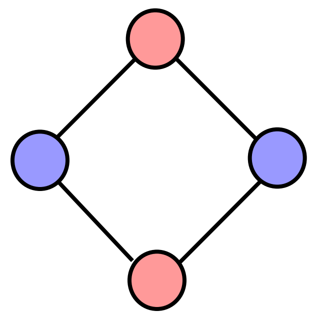
Improvement in Coloring Scheme

Optimistic Coloring (Briggs, Cooper, Kennedy, and Torczon)

- If Chaitin's algorithm reaches a state where every node has k or more neighbors, it chooses a node to spill.
- Briggs said, take that same node and push it on the stack
  - When you pop it off, a color might be available for it!

2 Registers:

2-colorable



Briggs algorithm finds an available color

  - For example, a node n might have k+2 neighbors, but those neighbors might only use just one color (or any number < k)
    → Degree is a <u>loose upper bound</u> on colorability

Chaitin-Briggs Algorithm

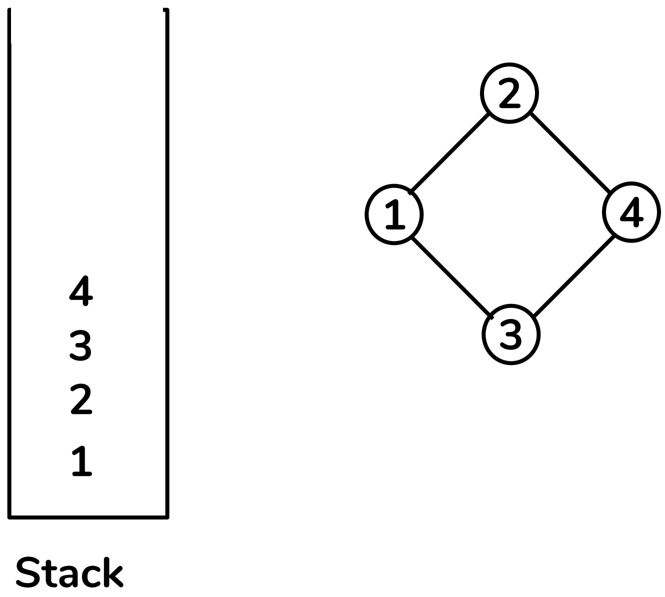
1. While ∃ vertices with < k neighbors in G_I
   > Pick any vertex n such that n°< k and put it on the stack
   > Remove that vertex and all edges incident to it from G_I
      1. This may create vertices with fewer than k neighbors

2. If G_I is non-empty (all vertices have k or more neighbors) then:
   > Pick a vertex n (using some heuristic condition), push n on the stack and remove n from G_I , along with all edges incident to it
   > If this causes some vertex in G_I to have fewer than k neighbors, then go to step 1; otherwise, repeat step 2

> Successively pop vertices off the stack and color them in the lowest color not used by some neighbor
   > If some vertex cannot be colored, then pick an uncolored vertex to spill, spill it, and restart at step 1

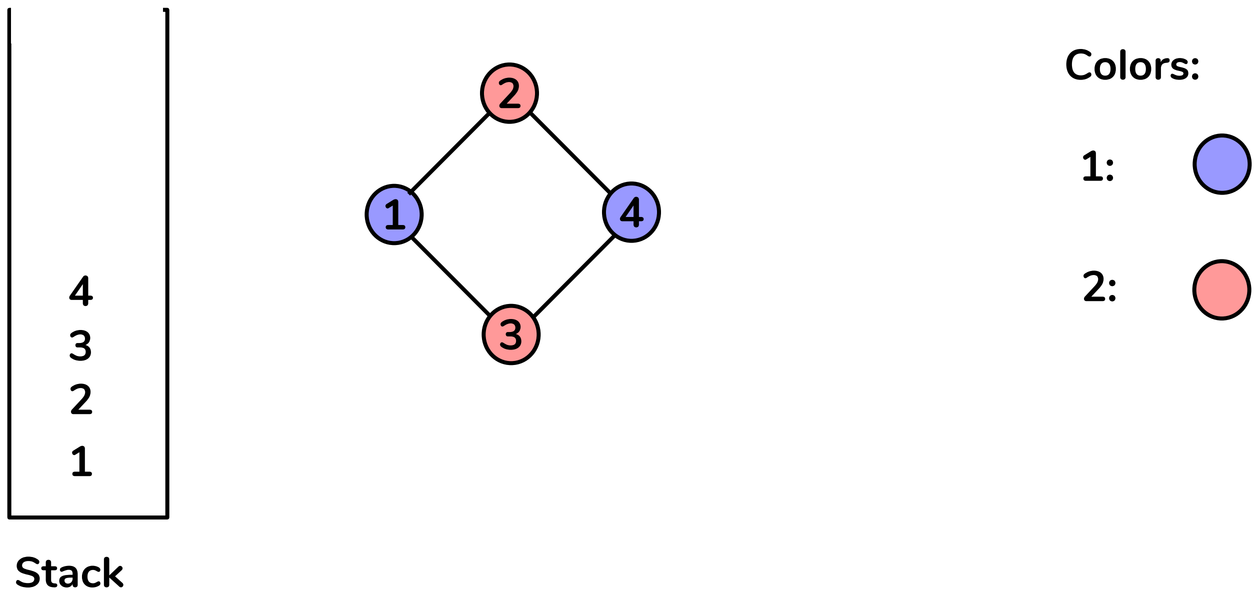
Chaitin-Briggs in Practice

2 Registers



Chaitin-Briggs in Practice

2 Registers



Worksheet 12

» Color the interference graph below with the minimum number of colors. Indicate if this coloring can be obtained using the Chaitin or Chaitin-Briggs algorithms studied in class.