Code Generation

Part III

Peephole Optimization

- Section: 8.7 of Dragon Book
  - Page 549~552
  - Read Yourself
  - Important

Register Allocation

How to best use the bounded number of registers.

- Reducing load/store operations
- What are best values to keep in registers?
- When can we ‘free’ registers?

Complications:

- special purpose registers
- operators requiring multiple registers.

Register Strategies

Option 1

Keep every variable in memory at all times.

Use 1 or 2 "work" registers during code generation.

Code Generator #1:

Every statement in isolation.

Variables in memory between each IR instruction.

Option 2

Keep all variables in memory between basic blocks.

Generate code for each basic block in isolation.

Within the basic block, use registers to hold values.

At the end of the basic block,

store all (LIVE) variables back to memory.

Register Strategies

Option 3

Divide registers into groups
- Work registers (e.g., R0-R4)
- Variable storage (e.g., R5-R23)
- Other (e.g., %fp, not available for variable storage)

Look at the entire flow graph.
Select some variables that are used "a lot."
Put them in registers for the entire routine.
Beginning of routine: generate a "LOAD"
End of routine: generate a "STORE"
Remaining variables: Keep in memory
Generate LOADs and STOREs when they are accessed.

Problems:
- Non-local access?
  Other routines expect variable to be in memory.
  Identify variables that are used only locally.
  Generate STOREs before each CALL
- How to select variables?
  A variable used only 2 times... but within a loop!

Register Class / Pragmas

Terminology

"Local Register Allocation"

Register allocation is done for the entire basic block.

... But each basic block is done independently.

"Global Register Allocation"

Register allocation is done for the entire flow graph / routine.

... But each routine is done independently.

Option 2 + 3

Set aside some registers for

"Local Allocation" -- within the basic block

"Global Allocation" -- for the entire routine

Global Register Allocation

Option 4

"Global Register Allocation"

Look at the entire flow graph.

- Look at variable lifetimes
- Identify "Live Ranges"
- Map each Live Range into a register

Graph-Coloring Algorithm

The Register-To-Register Model

Assume an infinite number of registers

"Virtual Registers"

During code generation... an inexhaustable supply.

Example: a := (x + y) * z;

```
LOAD    [%fp+offset_x],R_x
LOAD    [%fp+offset_y],R_y
ADD     R_x,R_y,R_tmp1
LOAD    [%fp+offset_z],R_z
MULT    R_tmp1,R_z,R_tmp2
STORE   R_tmp2,[%fp+offset_a]
```

The Register Allocation Phase

After target code generation

A limited supply of *"Physical Registers"*

Must select which Virtual Regs will go into Physical Regs.

When there are not enough Physical Regs...

Generate **STORE**s and **LOAD**s

"Spill" instructions

The Memory-To-Memory Model

Don't worry about registers.
Keep all variables in memory.
During code generation... an inexhaustable supply of *temporary variables*.

Example: **a := (x + y) * z;**

```
ADD     x,y,tmp1
MULT    tmp1,z,tmp2
STORE   tmp2,a
```

(This is the approach taken in our compiler.)

The Register Allocation Phase

After target code generation
A limited supply of *"Physical Registers"*
Must select which variables will go into Physical Registers.
Some variables are selected for *"promotion"* to registers.
Must generate **LOAD**s and **STORE**s.

The approaches are similar.

Register Allocation Algorithm

- Assume the code uses "Virtual Registers"
- Look at each basic block in isolation (a "local" approach)
- Identify all the virtual registers used in the basic block.
- Assign a "priority" to each virtual register.
  - Run through the instructions.
  - Count the number of times the virtual register is used.
- Assume that we have K physical registers available.
  - Identify the K virtual registers with the highest priority
  - Assign each to one of the physical registers.
- Run through the instructions and replace all uses of virt. registers.
  - If the virtual register has been assigned to a phys register, use that.
  - Otherwise, generate LOADs and STOREs as necessary.
    - Must set aside a couple of "work" registers for this.
      - Move the variable into a work register.
      - Use it.
      - Store it back in memory.

Global Register Allocation

Assign a variable to a register.
Keep it in register at all times, across Basic Block boundaries.

Problem: Non-Local Accesses

Call a subroutine?

It uses registers in its own ways.
[Will save any registers it modifies.]
It expects non-local variables to be
stored in their frames, buried in the stack.

Solution #1:

Save all variables back to memory whenever a call is made.

```
store   r3,[fp-4]
store   r4,[fp-8]
store   r5,[fp-12]
...
call    foo
load    [fp-4],r3
load    [fp-8],r4
load    [fp-12],r5
...
```

Global Register Allocation

Assign a variable to a register.
Keep it in register at all times, across Basic Block boundaries.

Problem: Non-Local Accesses

Call a subroutine?
- It uses registers in its own ways.
- [Will save any registers it modifies.]
- It expects non-local variables to be
  - stored in their frames, buried in the stack.

Solution #2:

Identify which variables are...
- accessed only locally
- accessed non-locally

Keep only "only locally accessed" variables in registers.
Approximation:
- Keep track of compiler-generated temporaries
- Keep only these in registers
  - But the real benefit is to keep heavily used variables in registers!

Global Allocation for Loops

Idea:

Identify Loops

Use Global Register Allocation for the duration of the loop

- Identify the basic blocks in a loop.
- Before going into the loop...
  - Move variables into registers
- Within the loop...
  - Just use the registers.

Issues:

- How many registers for global register allocation?
  - How many registers for "working storage"?
- Which variables to put into registers?
  - Choose "heavily used" variables
- Nested Loops ["inner loop" / "outer loop"]
  - Do the inner loops first.
- How to identify loops?

Register Allocation via Graph Coloring

Goal:

Keep all variables in registers

Keep each variable in a different register

...unless they are never LIVE simultaneously!

Example: "x" and "y"

Both LIVE at some point in the code?

⇒ Must put into different registers

Never LIVE at the same time?

⇒ May keep them in the same register!

Register Allocation with Graph Coloring

Local register allocation - graph coloring problem.

Uses liveness information.

Allocate K registers where each register is associated with one of the K colors.

Computing live status in basic blocks

Input: A basic block.

Output: For each statement, set of live variables

1. Initially all non-temporary variables go into live set (L).
2. for i = *last* statement to *first* statement:

   for statement i: x := y op z

   1. Attach L to statement i.
   2. Remove x from set L.
   3. Add y and z to set L.

Example

```
                        live =  {
a := b + c
                        live =  {
t1 := a * a
                        live =  {
b := t1 + a
                        live = {
c := t1 * b
                        live =  {
t2 := c + b
                        live = {
a := t2 + t2
                        live = {a,b,c}
```

Example Answers

live = {}

a := b + c

live = {}

t1 := a * a

live = {}

b := t1 + a

live = {}

c := t1 * b

live = {}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {}

t1 := a * a

live = {}

b := t1 + a

live = {}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

live = {}

a := b + c

live = {}

t1 := a * a

live = {}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Answers

```
                        live =  {}
a := b + c
                        live =  {}
t1 := a * a
                        live =  {a,t1}
b := t1 + a
                        live = { b,t1}
c := t1 * b
                        live =  {b,c}
t2 := c + b
                        live = {b,c,t2}
a := t2 + t2
                        live = {a,b,c}
```

Example Answers

```
                        live =  {}
a := b + c
                        live =  {a}
t1 := a * a
                        live =  {a,t1}
b := t1 + a
                        live = { b,t1}
c := t1 * b
                        live =  {b,c}
t2 := c + b
                        live = {b,c,t2}
a := t2 + t2
                        live = {a,b,c}
```

Example Answers

live = {b,c} ← what does this mean???

a := b + c

live = {a}

t1 := a * a

live = {a,t1}

b := t1 + a

live = { b,t1}

c := t1 * b

live = {b,c}

t2 := c + b

live = {b,c,t2}

a := t2 + t2

live = {a,b,c}

Example Code

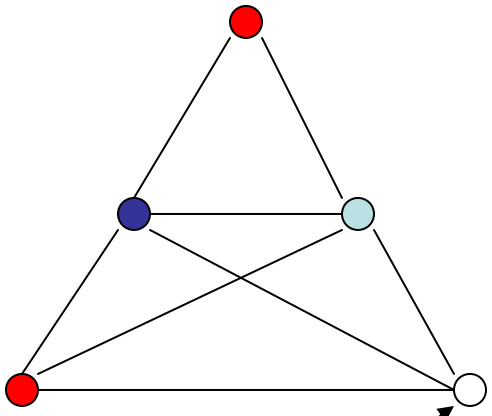
	live = {b,c}
a := b + c	
	live = {a}
t1 := a * a	
	live = {a,t1}
b := t1 + a	
	live = { b,t1}
c := t1 * b	
	live = {b,c}
t2 := c + b	
	live = {b,c,t2}
a := t2 + t2	
	live = {a,b,c}

Graph Coloring

- The coloring of a graph G = (V,E) is a mapping C: V→ S, where S is a finite set of colors, such that if edge vw is in E, C(v) <> C(w).
- Problem is NP (for more than 2 colors) ➔ no polynomial time solution.
- Fortunately there are approximation algorithms.

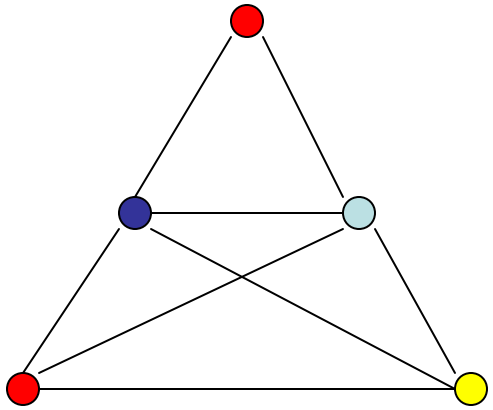
Coloring a graph with K colors

K = 3



No color for this node

K = 4



Register Allocation and Graph K-Coloring

K = number of available registers

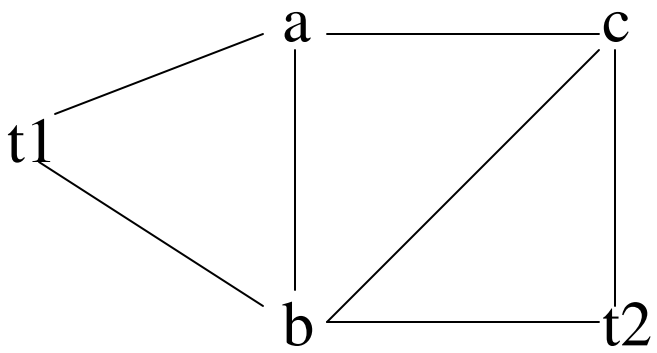
G = (V,E) where

- Vertex set V = {V_s | s is a program variable}
- Edge V_s V_t in E if s and t can be live at the same time

G is an '*interference graph*'

Example Interference Graph

a := b + c	{b,c}
t1 := a * a	{a}
b := t1 + a	{t1,a}
c := t1 * b	{b,t1}
t2 := c + b	{b,c}
a := t2 + t2	{b,c,t2}
	{a,b,c}



Algorithm: K registers

1. Compute liveness information for the basic block.

2. Create interference graph G - one node for each variable, an edge connecting any two variables alive simultaneously.

3. **Simplify** - For any node m with fewer than K neighbors, remove it from the graph and push it onto a stack. If G - m can be colored with K colors, so can G. If we reduce the entire graph, goto step 5.

Choosing a Spill Node

Potential criteria:

- Random
- Most neighbors
- Longest live range (in code)
  - with or without taking the access pattern into consideration

Algorithm: K registers

5. **Assign colors** - Starting with empty graph, rebuild graph by popping elements off the stack, putting them back into the graph and assigning them colors different from neighbors. Potential spill nodes may or may not be colorable.

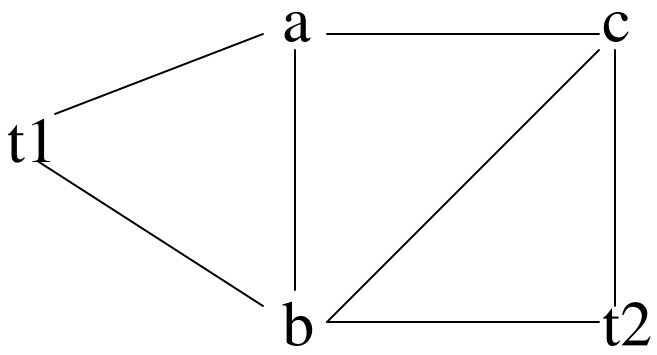
Process may require iterations and rewriting of some of the code to create more temporaries.

Rewriting the code

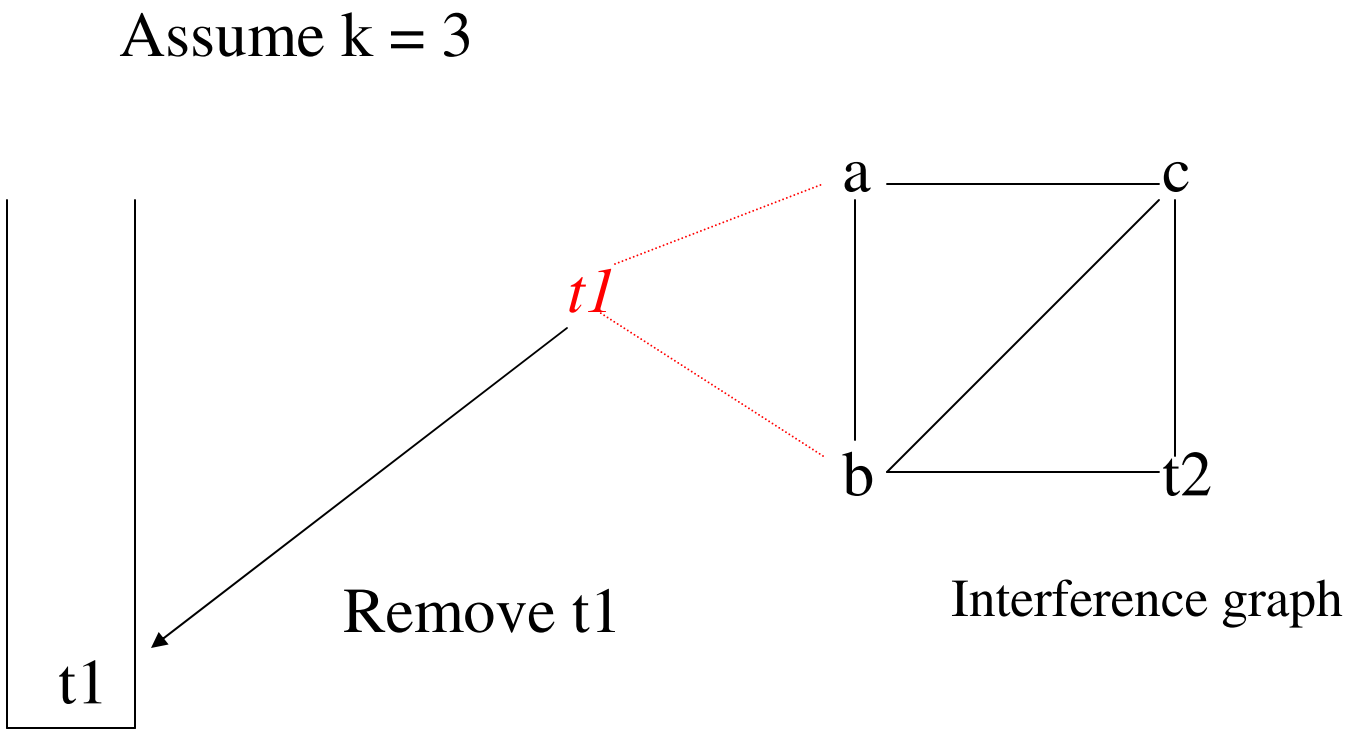
- Want to be able to remove some edges in the interference graph
  - write variable to memory earlier
  - compute/read in variable later

Back to example

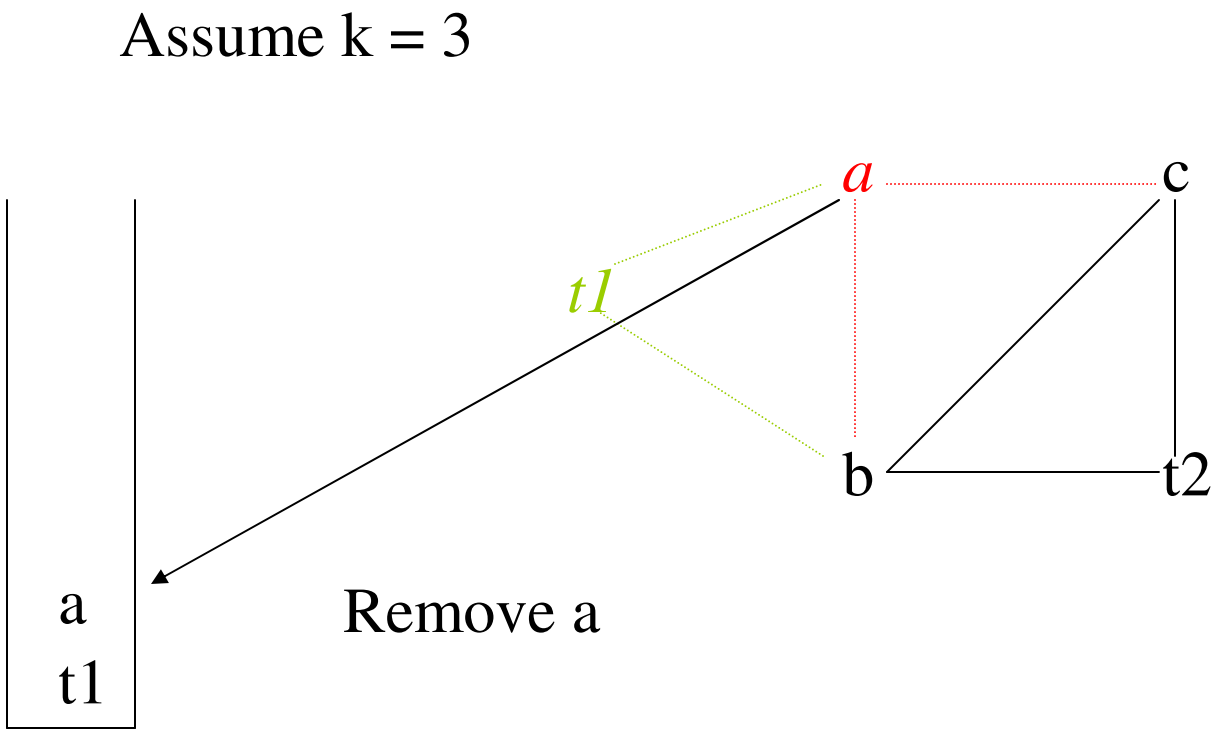
a := b + c	{b,c}
t1 := a * a	{a}
b := t1 + a	{t1,a}
c := t1 * b	{b,t1}
t2 := c + b	{b,c}
a := t2 + t2	{b,c,t2}
	{a,b,c}



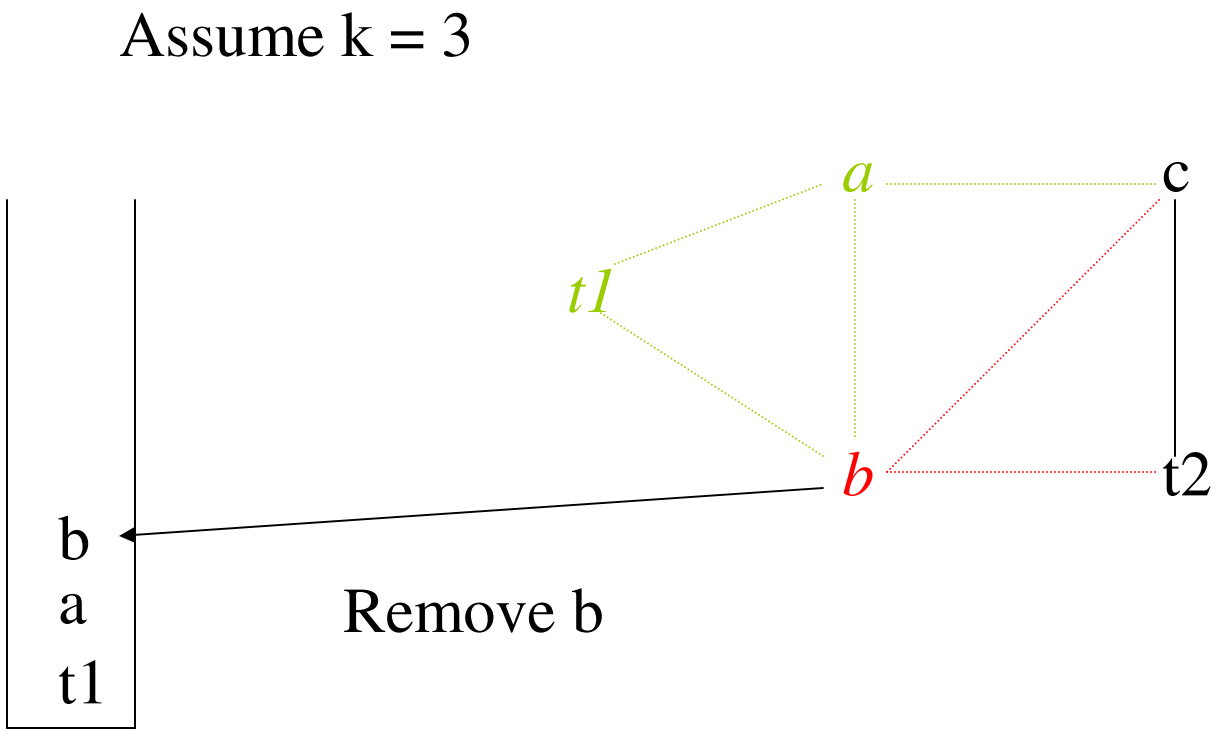
Example, k = 3



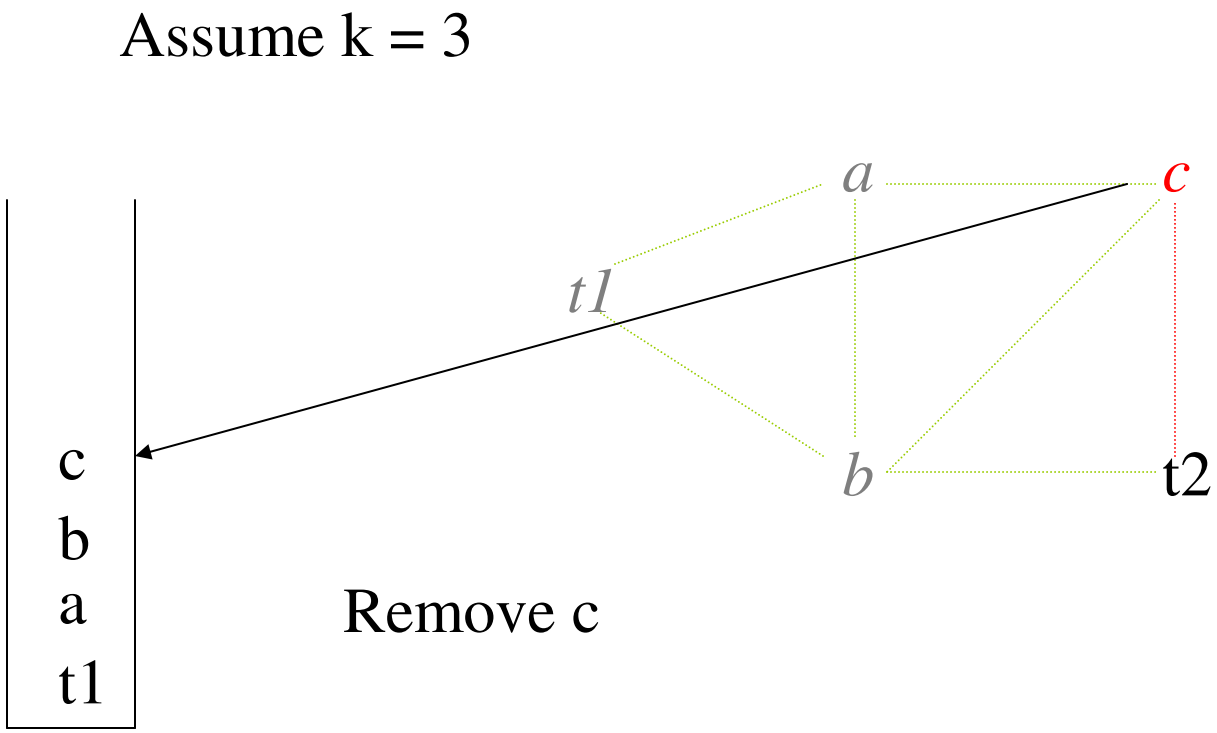
Example



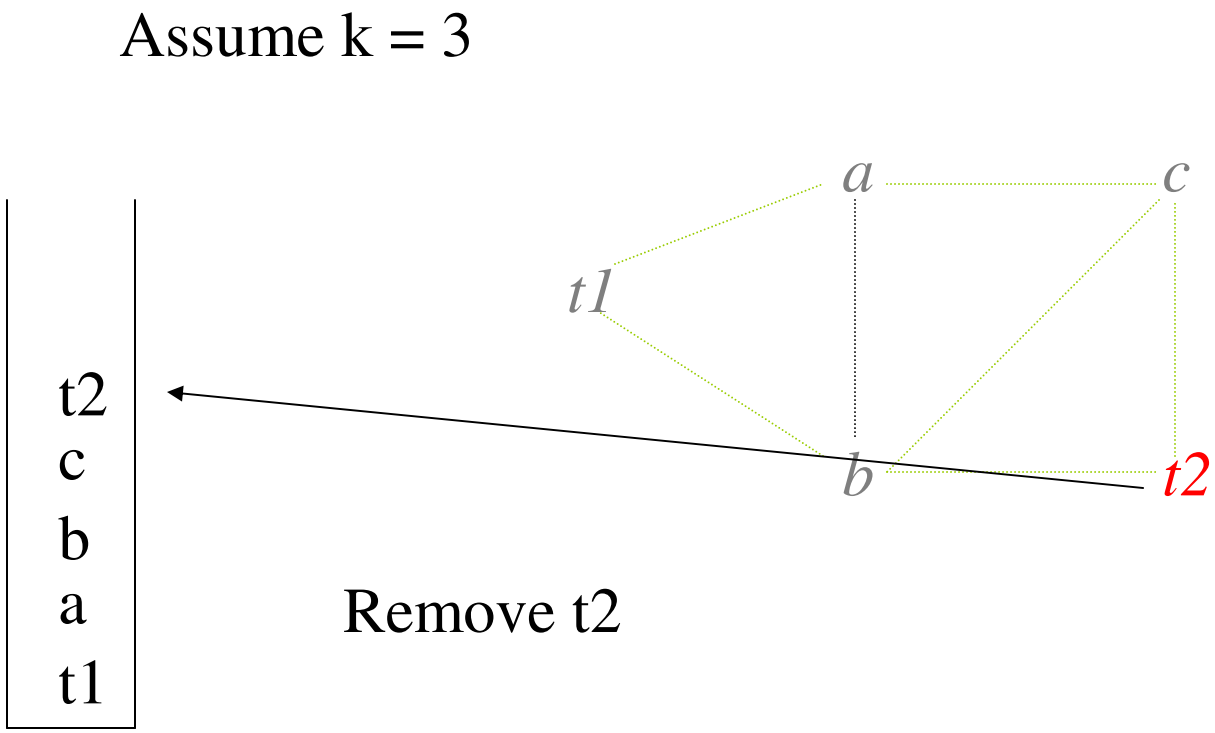
Example



Example

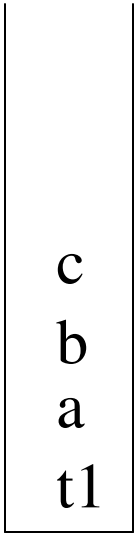


Example



Rebuild the graph

Assume k = 3



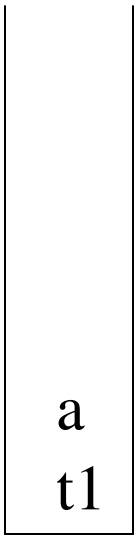
t2

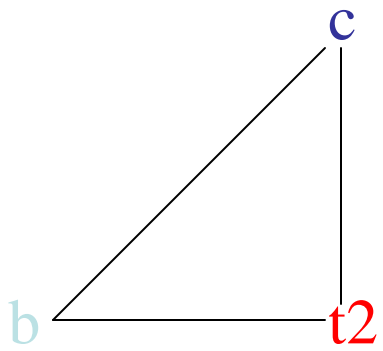
Example

Assume k = 3

Example

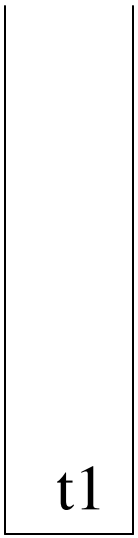
Assume k = 3

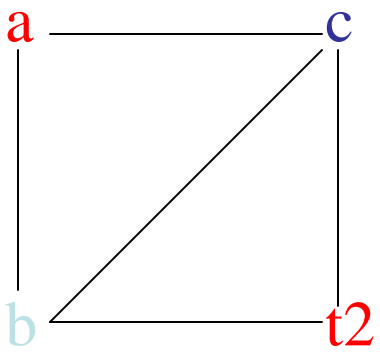




Example

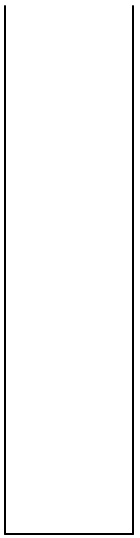
Assume k = 3



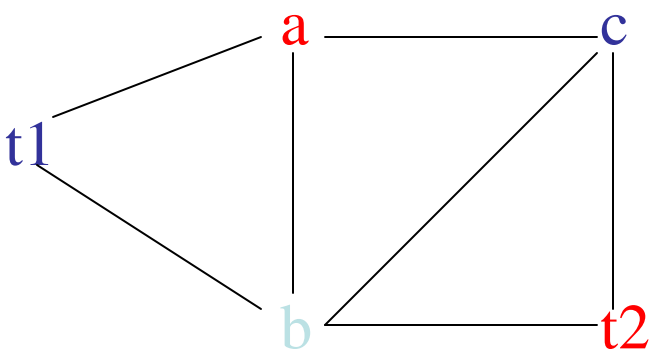


Example

Assume k = 3

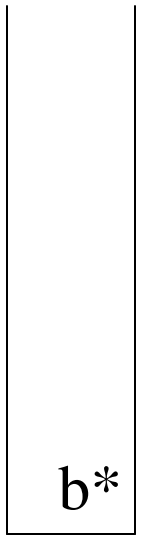


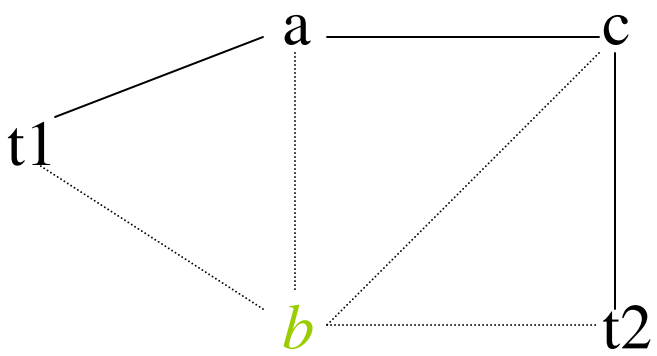
a	t0
b	t1
c	t2
t1	t2
t2	t0



Example, k = 2

Assume k = 2

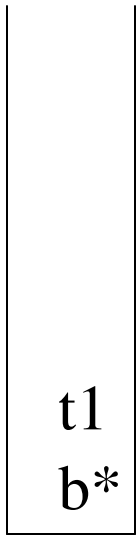




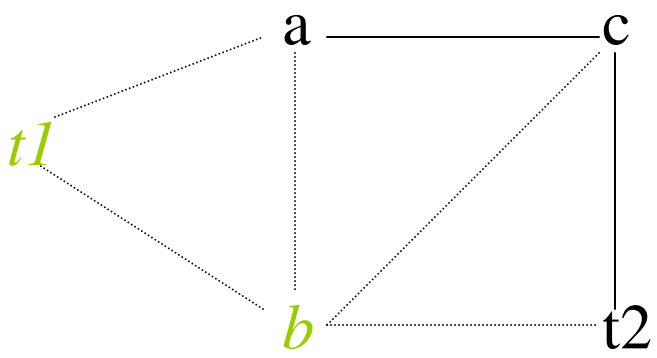
Remove b as spill

Example

Assume k = 2

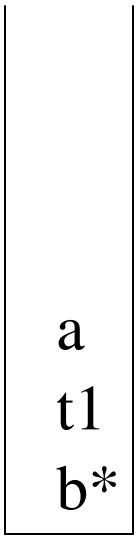


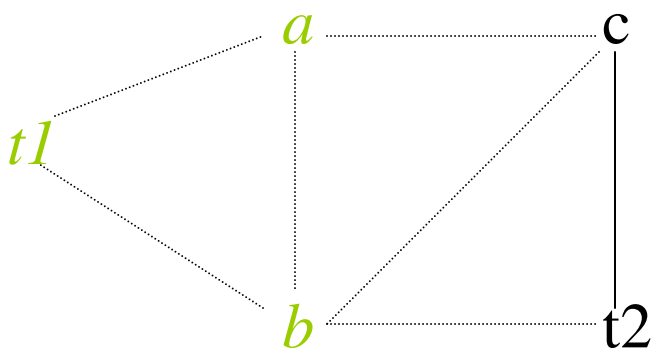
Remove t1



Example

Assume k = 2

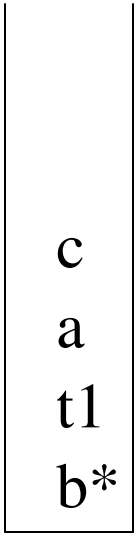


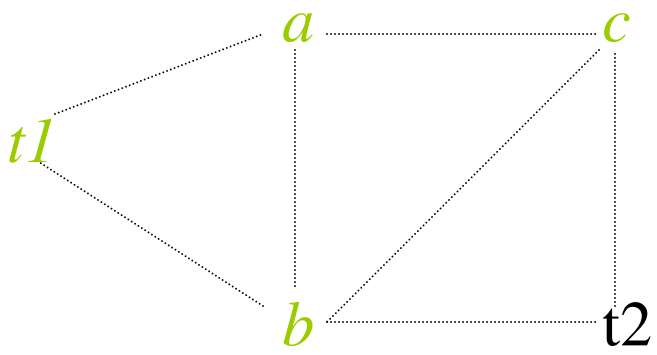


Remove a

Example

Assume k = 2

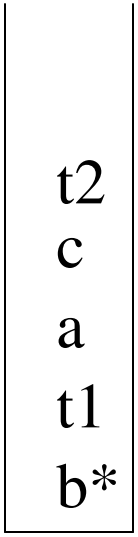


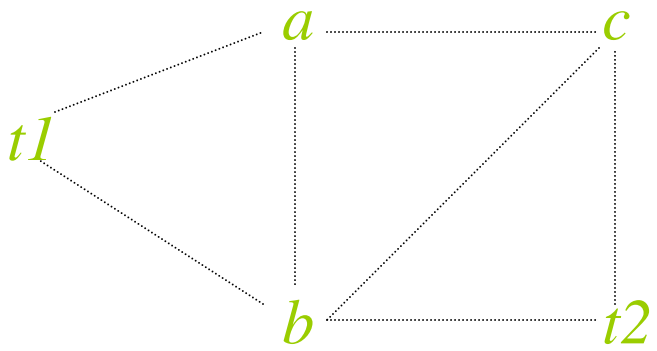


Remove c

Example

Assume k = 2

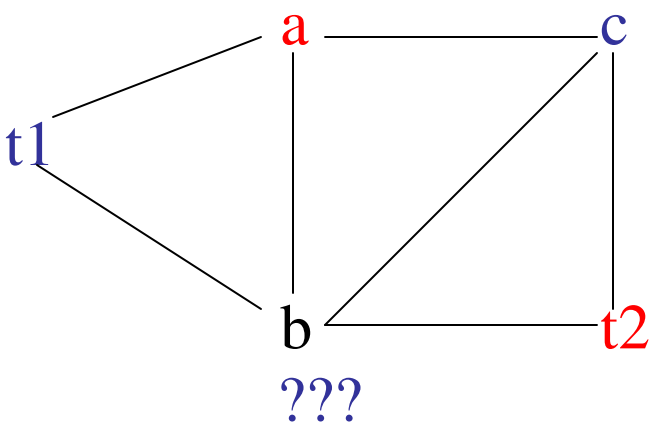




Remove t2

Example

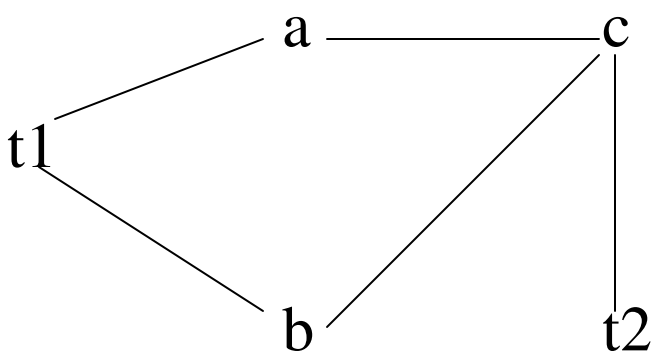
Assume k = 2



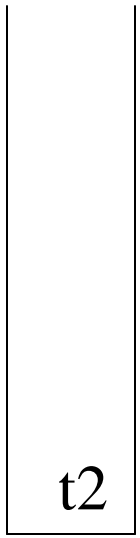
Can flush b out to memory, creating a smaller window

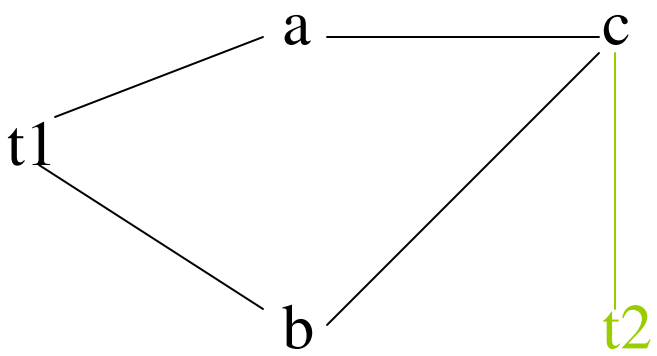
After spilling b:

a := b + c	{b,c}
t1 := a * a	{a}
b := t1 + a	{t1,a}
c := t1 * b	{b,t1}
b to memory	
t2 := c + b	{b,c}
a := t2 + t2	{c,t2}
	{a, c}

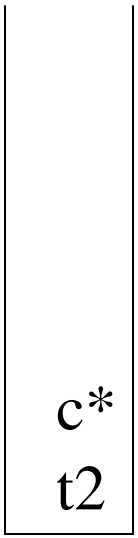


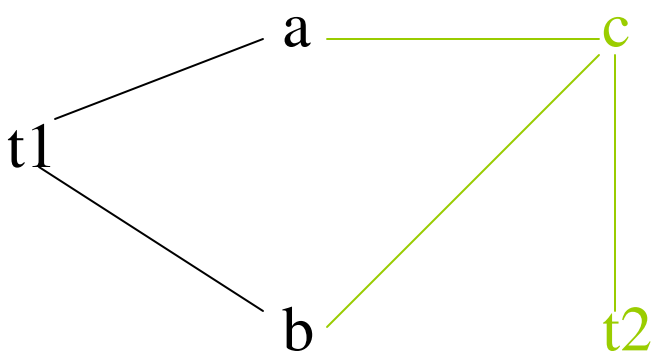
After spilling b:





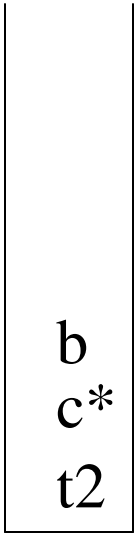
After spilling b:

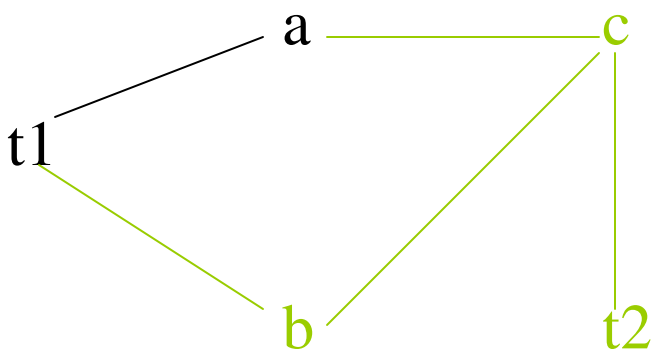




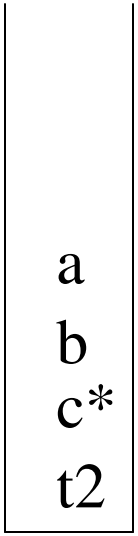
Have to choose c as a potential spill node.

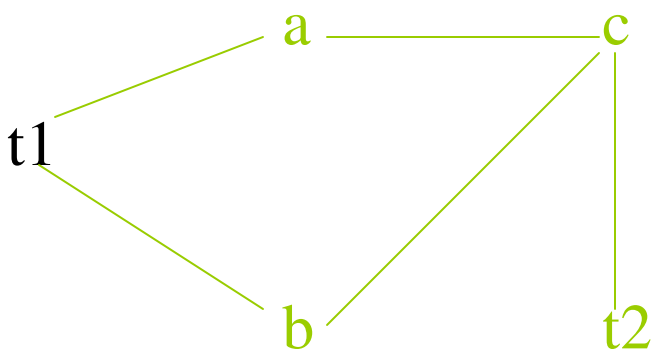
After spilling b:



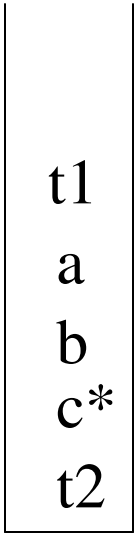


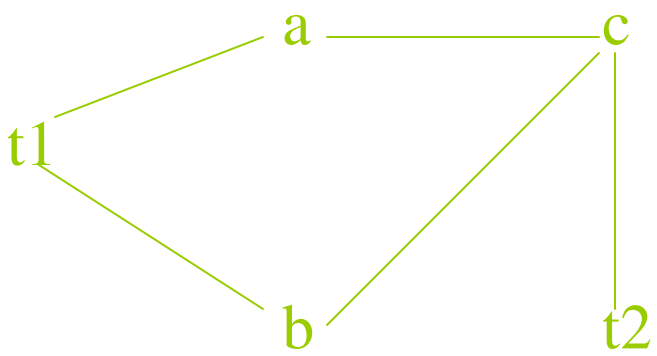
After spilling b:



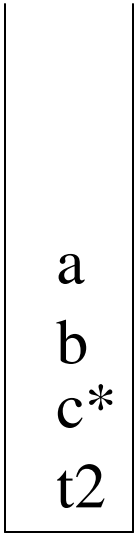


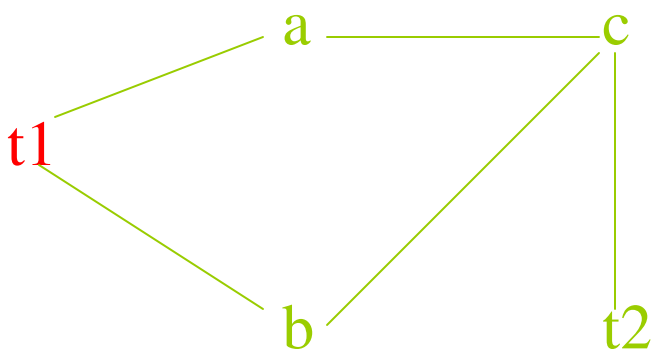
After spilling b:



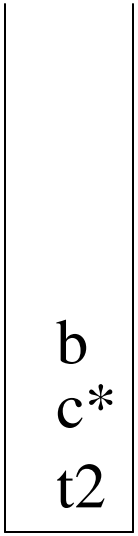


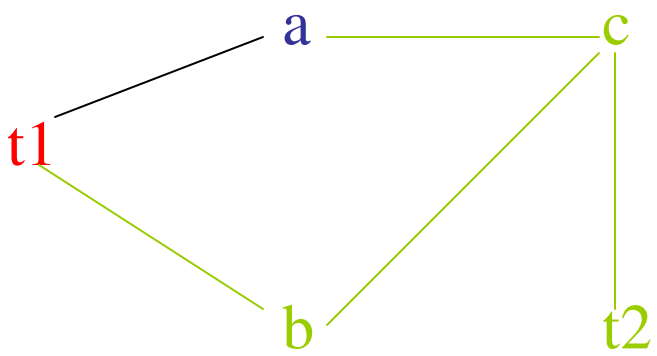
Now rebuild:



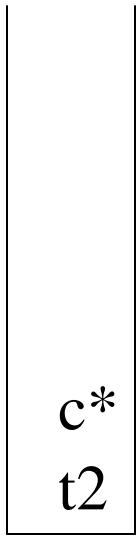


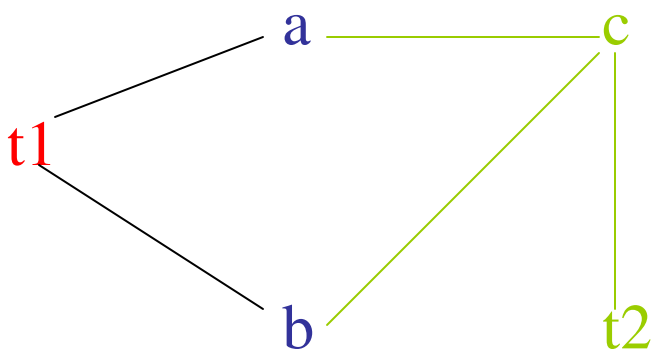
Now rebuild:



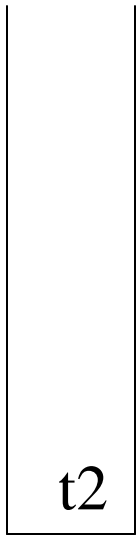


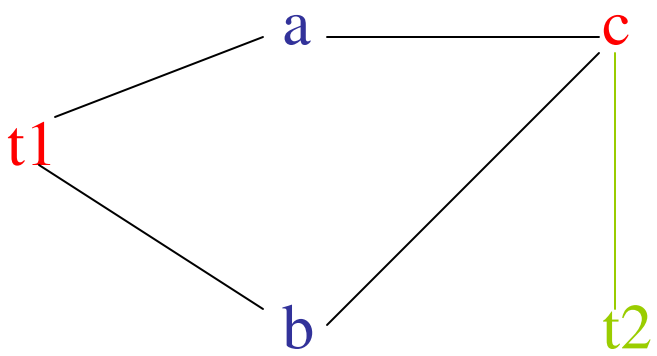
Now rebuild:





Now rebuild:

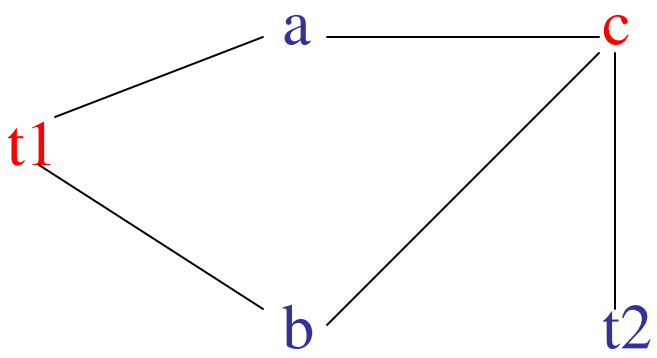




Fortunately, there is a color for c

Now rebuild:

a	t0
b	t0
c	t1
t1	t1
t2	t0



The graph is 2-colorable now