Code Generation

Register Allocation and Assignment

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 Allocation and Assignment

One approach:

- Assign specific values to certain registers, e.g., base addresses to one group of registers, arithmetic computation to another etc. Reducing load/store operations

Advantage:

- Simplifies the design of a compiler

Disadvantage:

- It uses registers inefficiently
- Certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated

Tradition:

- Reserve few registers for base registers, stack pointers etc and allow the remaining registers to be used by the compiler

Global Register Allocation

Look at the entire flow graph

- Look at variable lifetimes
- Identify “Live ranges”
- Map each Live range into a register

- Assign some fixed number of registers to hold the most active values
- Selected values may be different
- Fixed number of registers is not always the right number to make available for global register allocation

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

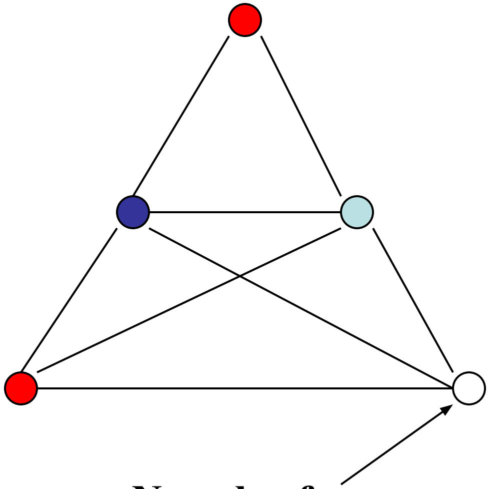
```
              live =  {b,c}
a := b + c
              live =  {a}
t1 := a * a
              live =  {a,t1}
b := t1 + a
              live = { b,t1}
c := t1 * b
              live =  {b,c}
t2 := c + b
              live = {b,c,t2}
a := t2 + t2
              live = {a,b,c}
```

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).

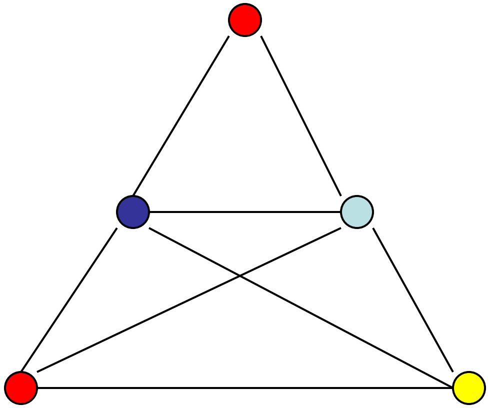
Coloring a graph with K colors

K = 3



No color for
this node

K = 4



Register Allocation and Graph K-Coloring

K = number of available registers

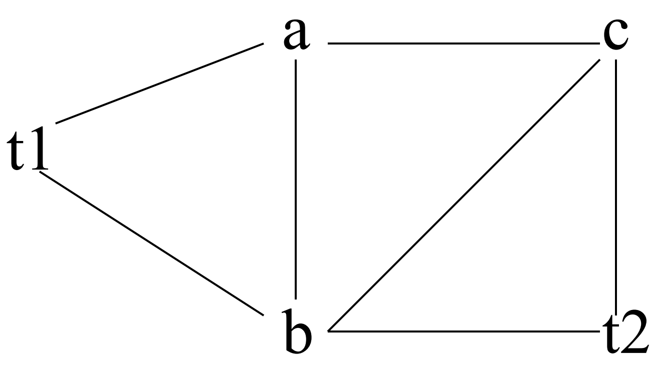
G = (V,E) where

- 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.

4. **Spill** - If we get to the point where we are left with only nodes with degree >= K, mark some node for potential spilling. Remove and push onto stack. Back to step 3.

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.

Choosing a Spill Node

Potential criteria:

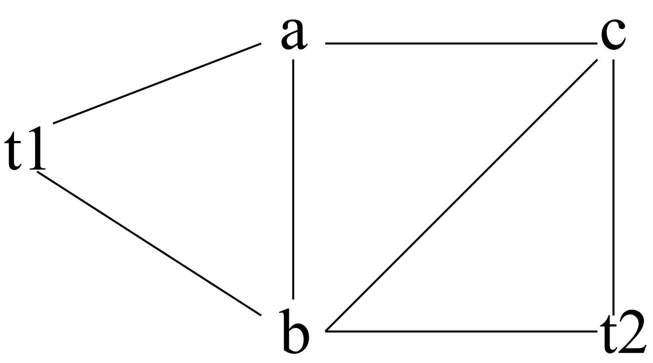
- Random
- Most neighbors
- Longest live range (in code)
  - with or without taking the access pattern into consideration

Rewriting the code

- Process may require iterations and rewriting of some of the code to create more temporaries.
- Want to be able to remove some edges in the interference graph
  - write variable to memory earlier
  - compute/read in variable later

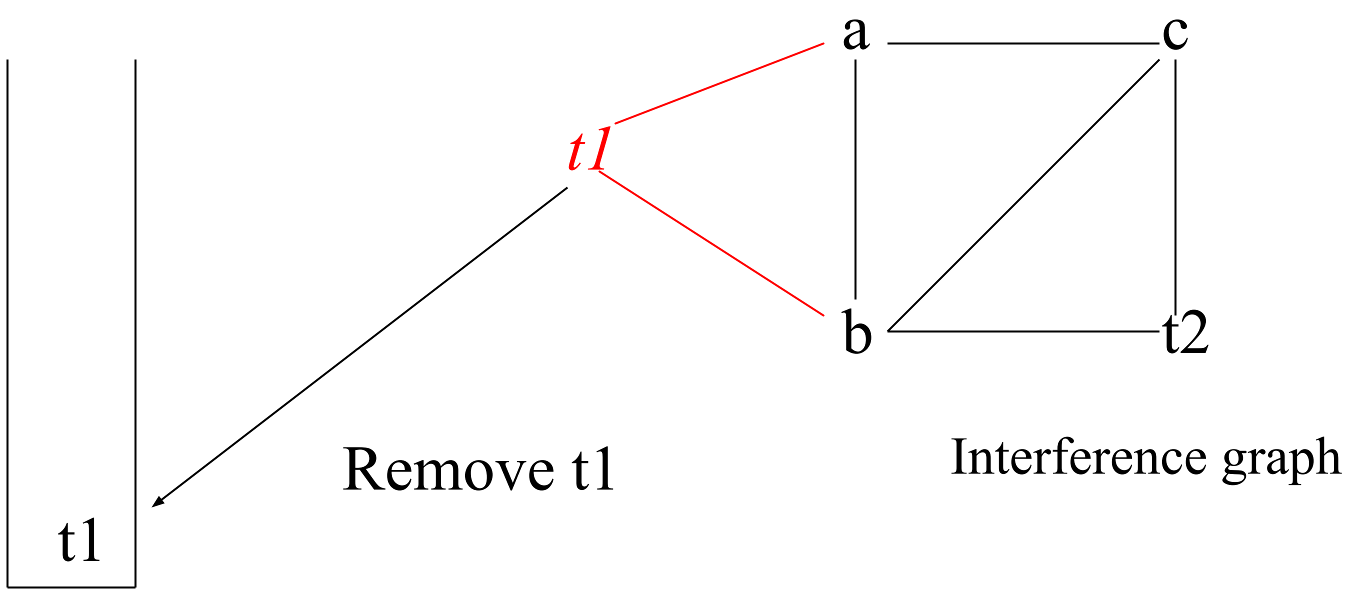
Back to example

```
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

Assume k = 3

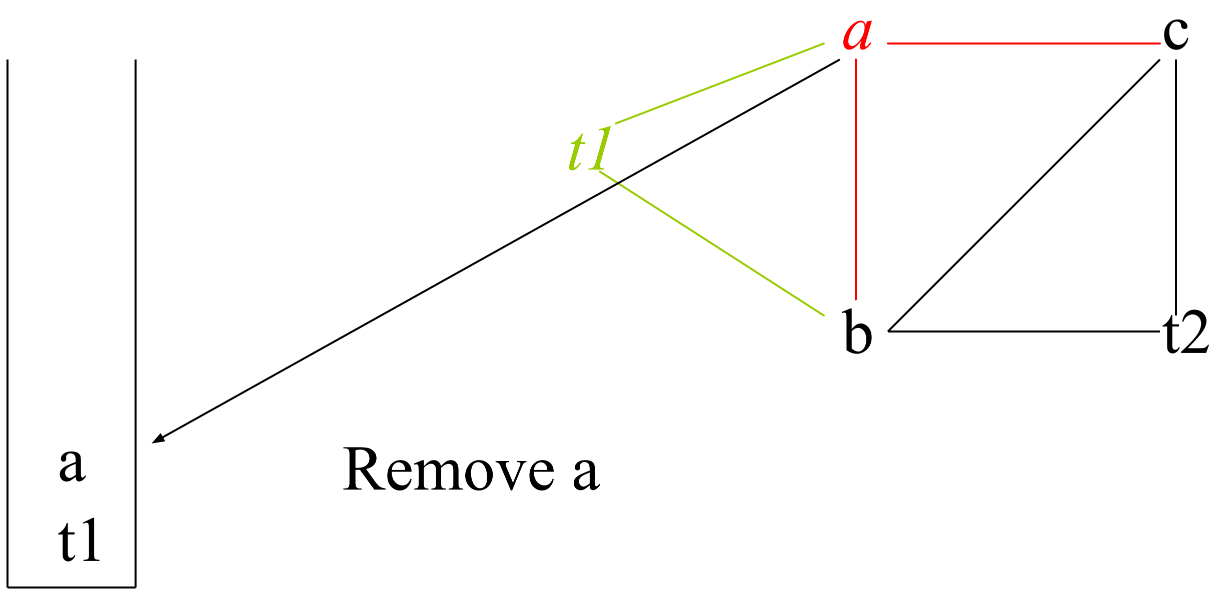


Remove t1

Interference graph

Example

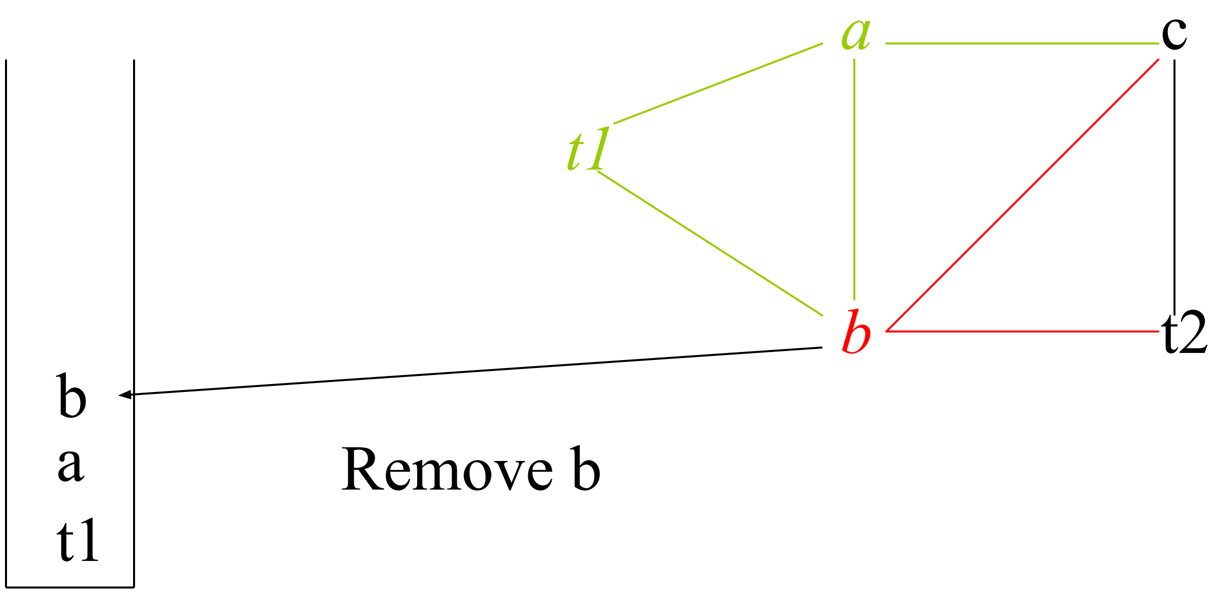
Assume k = 3



Remove a

Example

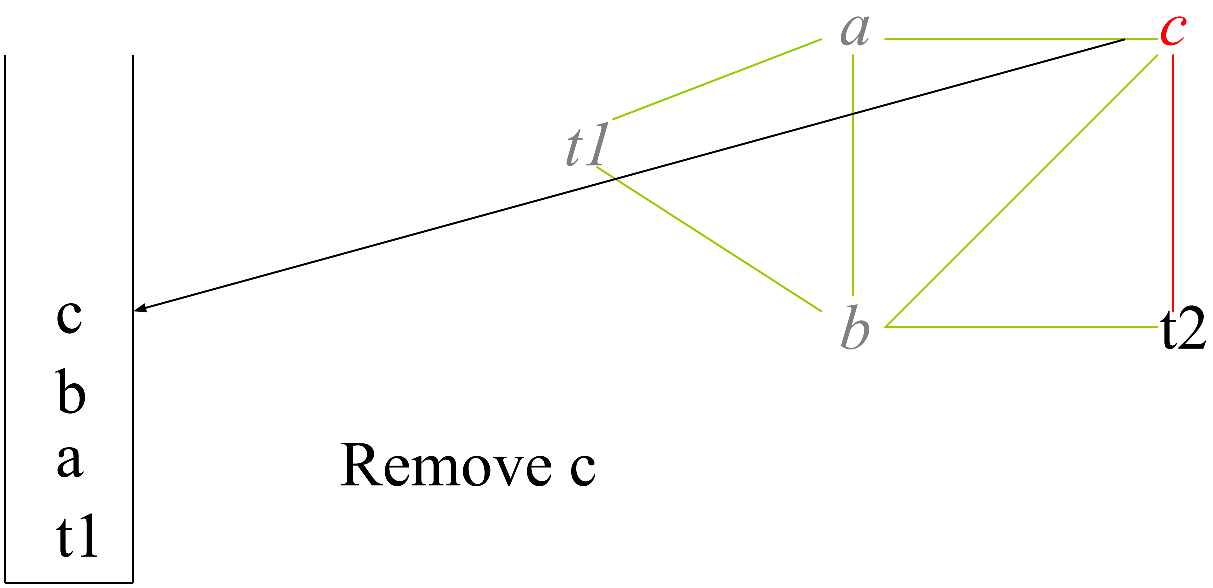
Assume k = 3



Remove b

Example

Assume k = 3



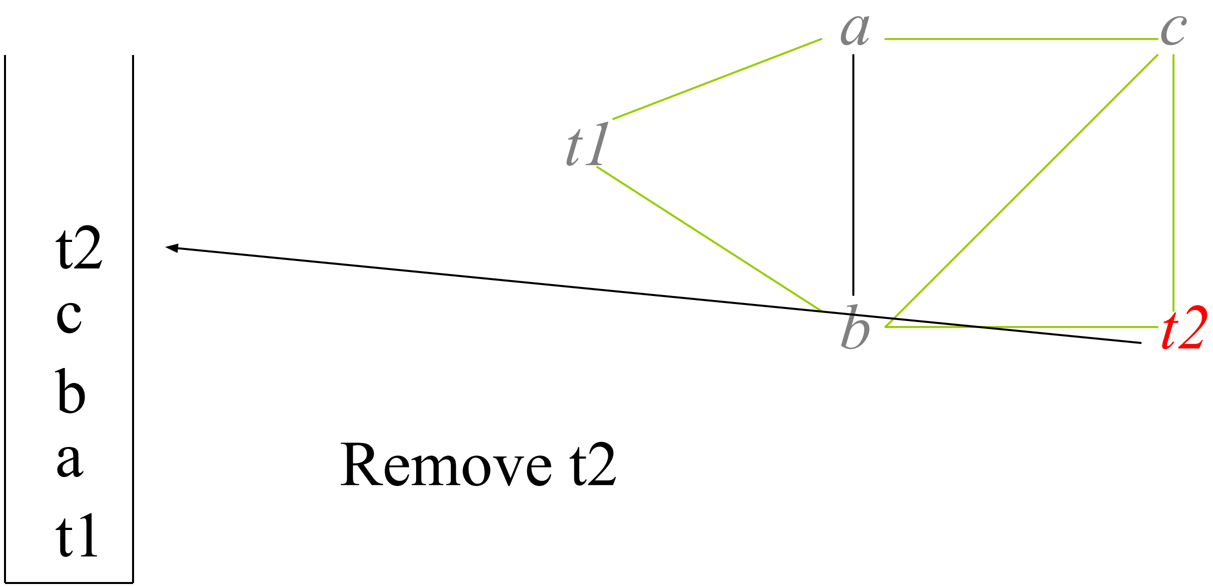
c
b
a
t1

a
c
t1
b
t2

Remove c

Example

Assume k = 3

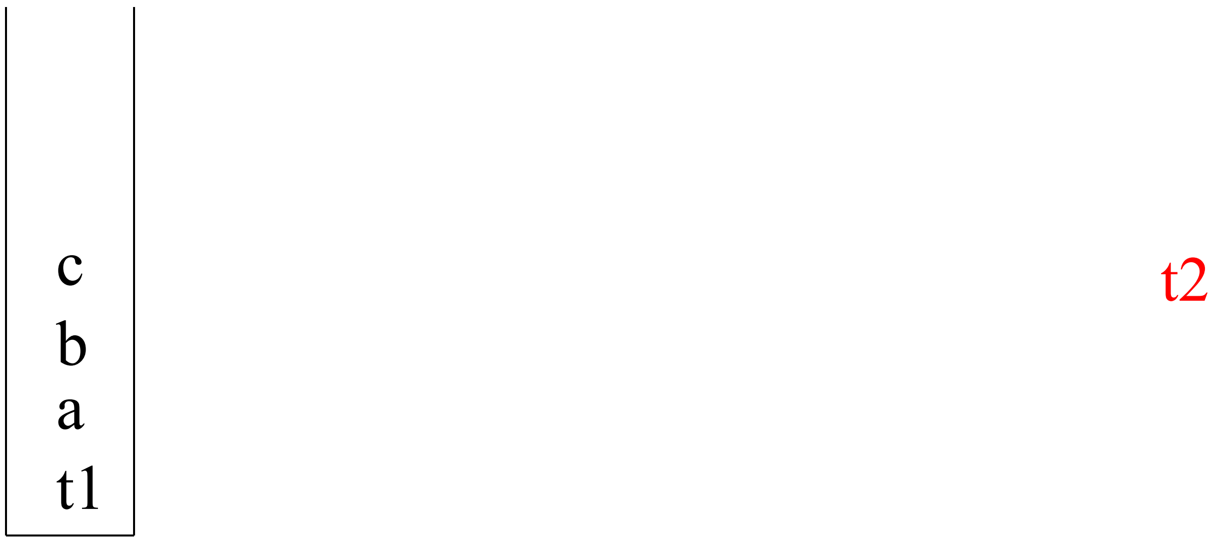


t2
c
b
a
t1

Remove t2

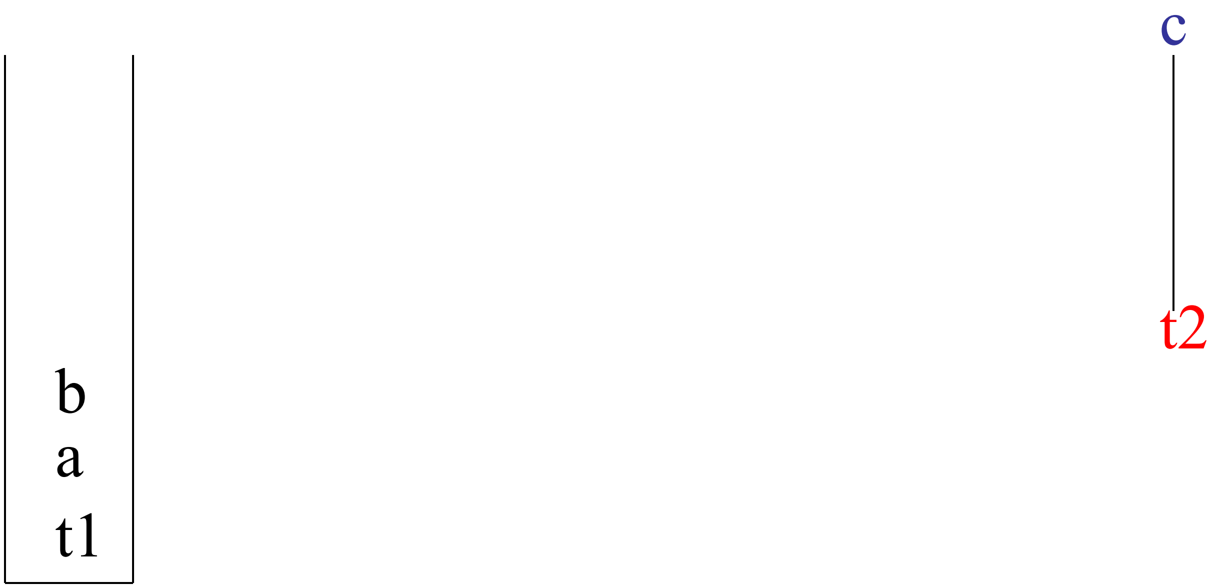
Rebuild the graph

Assume k = 3



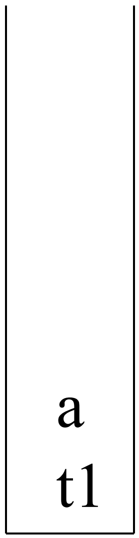
Example

Assume k = 3



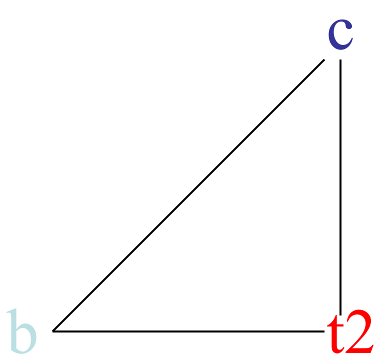
Example

Assume k = 3



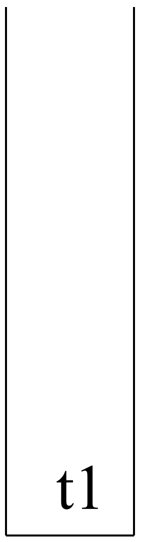
a

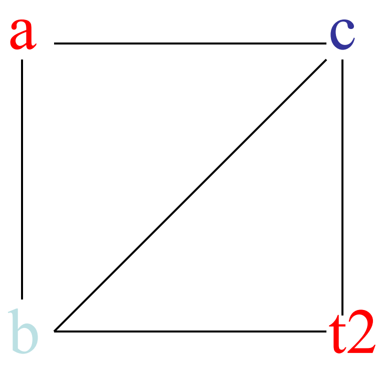
t1



Example

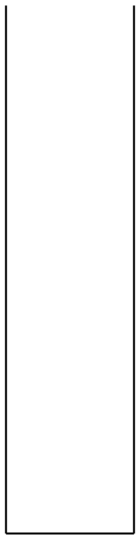
Assume k = 3

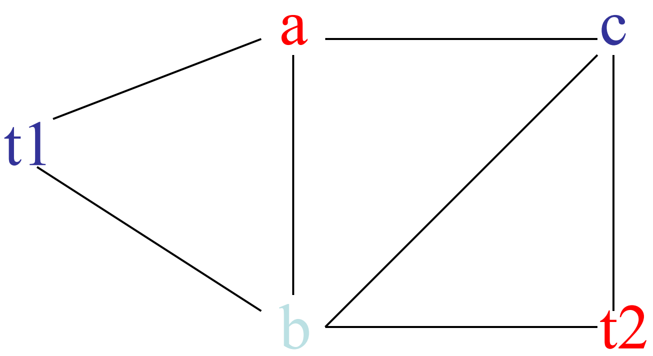




Example

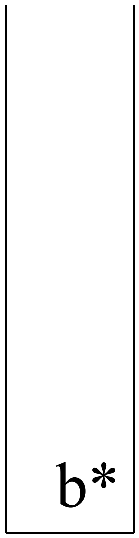
Assume k = 3

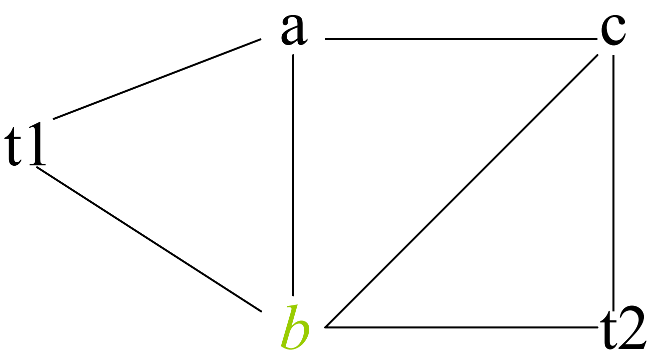




Example, k = 2

Assume k = 2

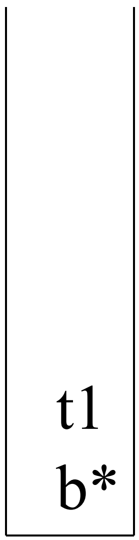




Remove b as spill

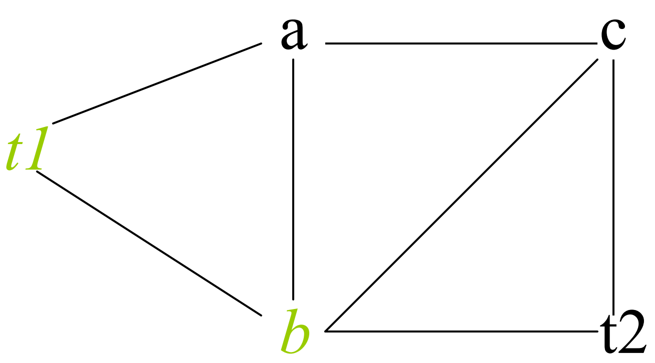
Example

Assume k = 2



t1

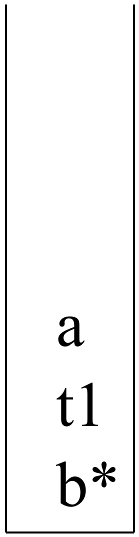
b*



Remove t1

Example

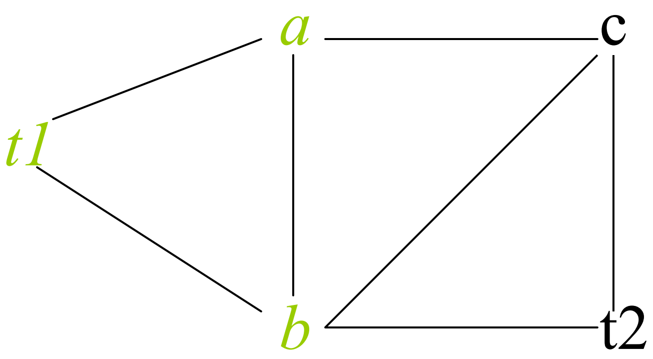
Assume k = 2



a

t1

b*



Remove a

Example

Assume k = 2

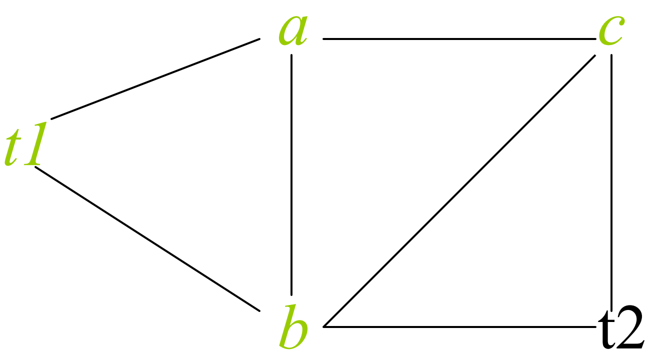
c

a

t1

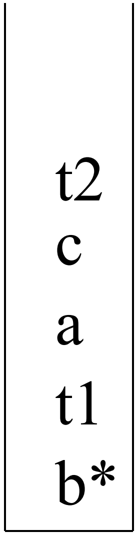
b*

Remove c

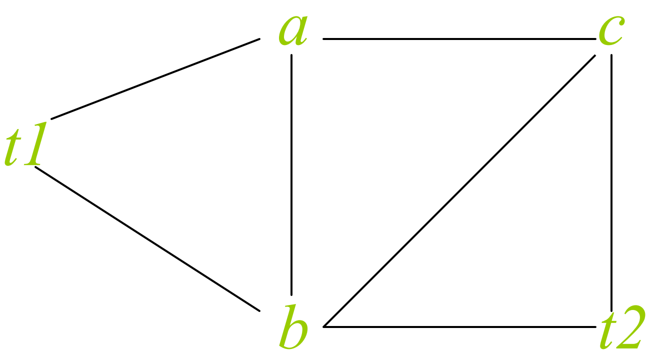


Example

Assume k = 2



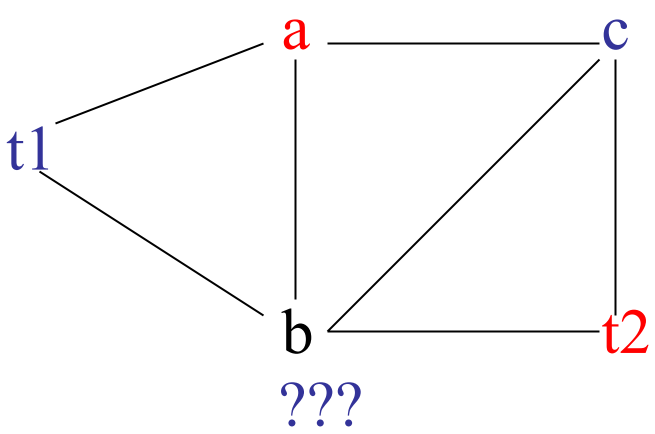
t2
c
a
t1
b*



Remove t2

Example

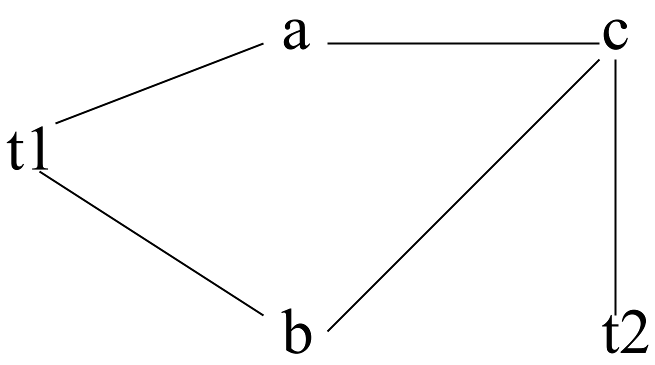
Assume k = 2



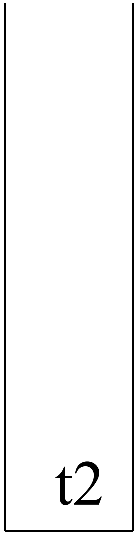
Can flush b out to memory, creating a smaller window

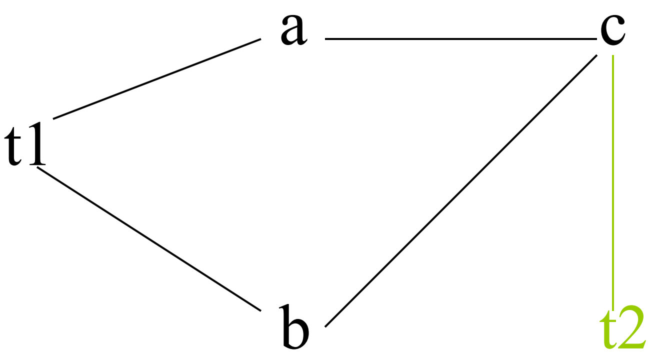
After spilling b:

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

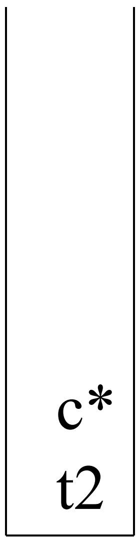


After spilling b:



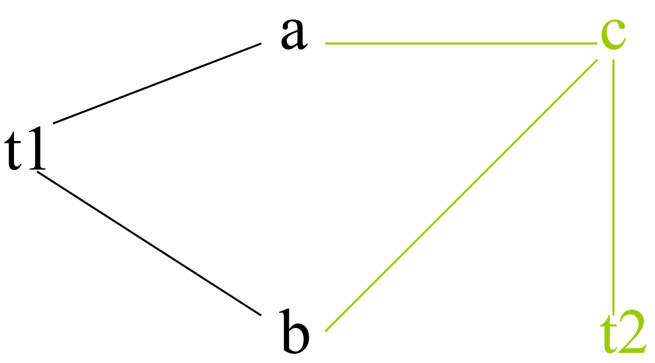


After spilling b:



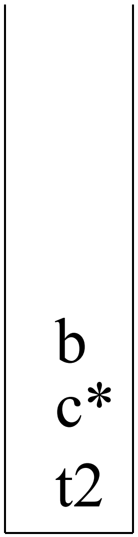
c*

t2

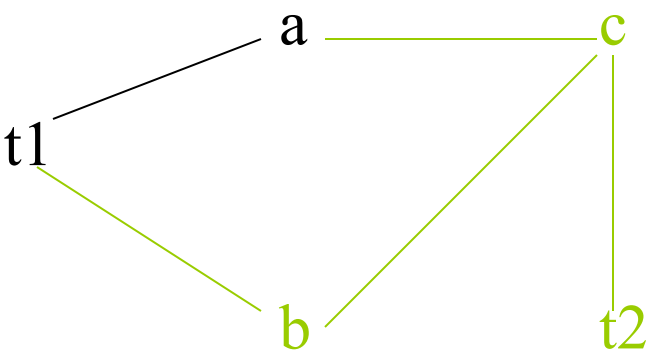


Have to choose c as a potential spill node.

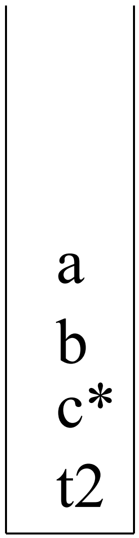
After spilling b:



b
c*
t2



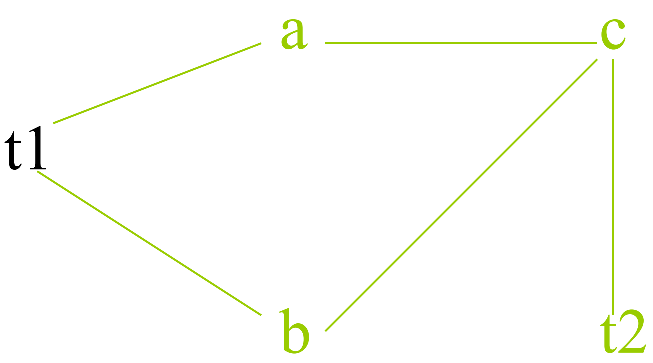
After spilling b:


a

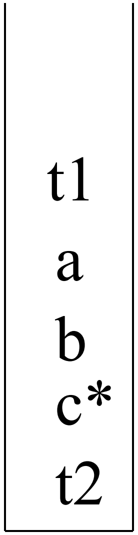
b

c*

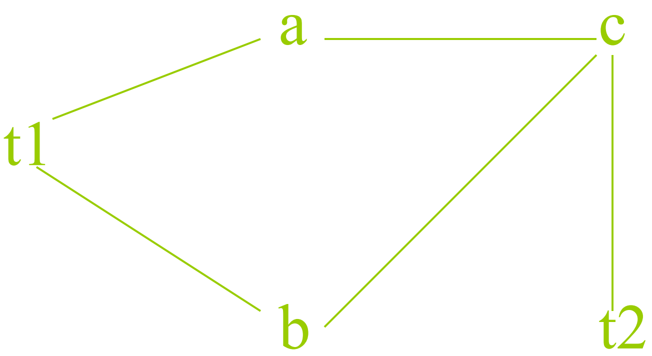
t2



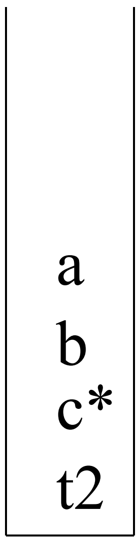
After spilling b:



t1
a
b
c*
t2



Now rebuild:

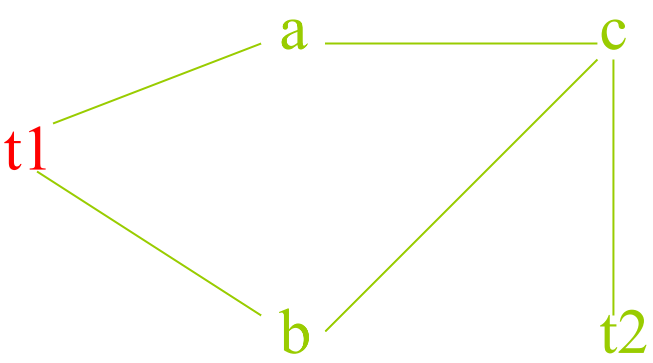


a

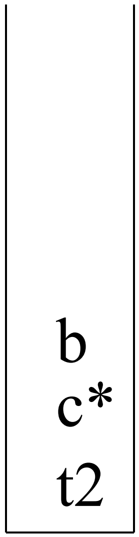
b

c*

t2



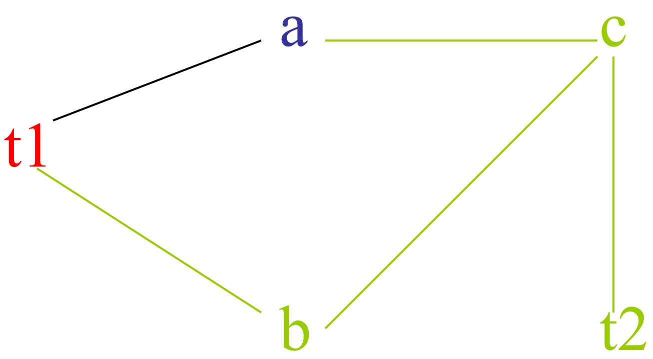
Now rebuild:



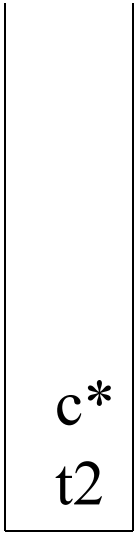
b

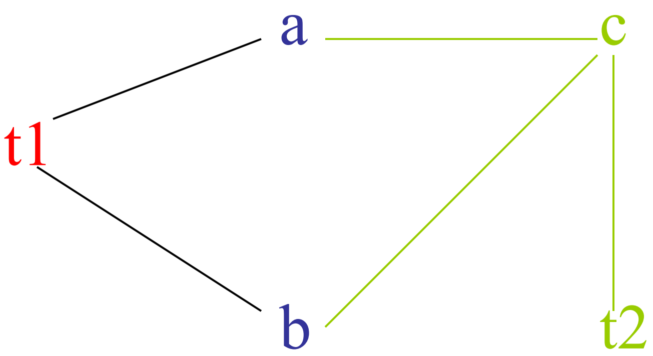
c*

t2

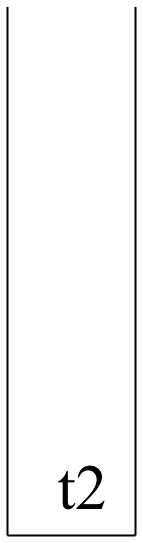


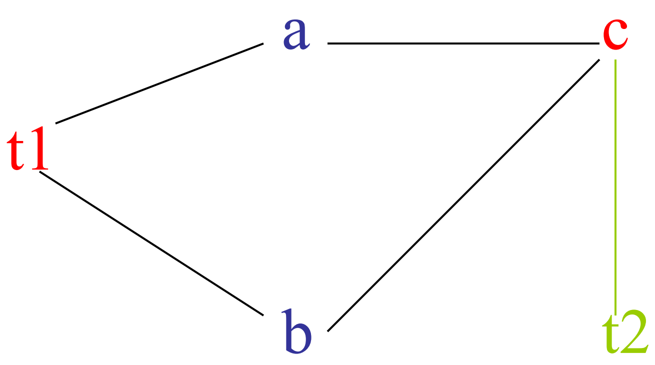
Now rebuild:





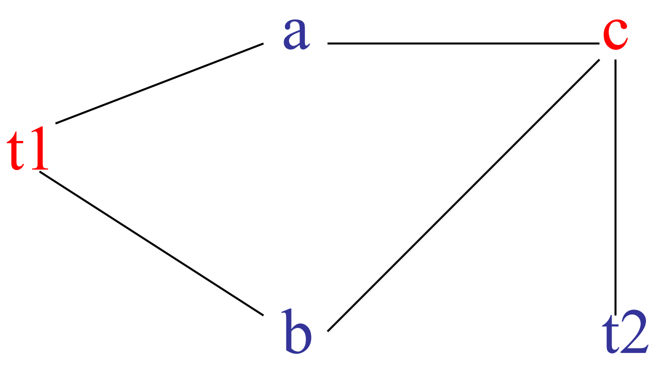
Now rebuild:





Fortunately, there is a color for c

Now rebuild:



The graph is 2-colorable now