More Code Generation and Optimization

Pat Morin

COMP 3002

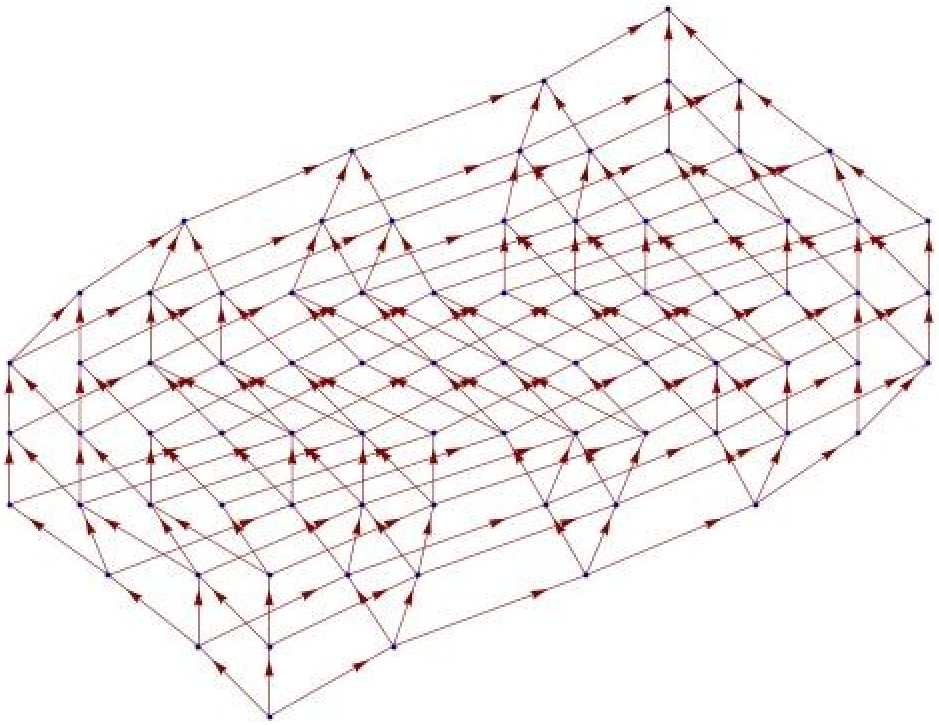
Carleton University
Canada's Capital University

Outline

- DAG representation of basic blocks
- Peephole optimization
- Register allocation by graph coloring

Carleton UNIVERSITY
Canada's Capital University

Basic Blocks as DAGs



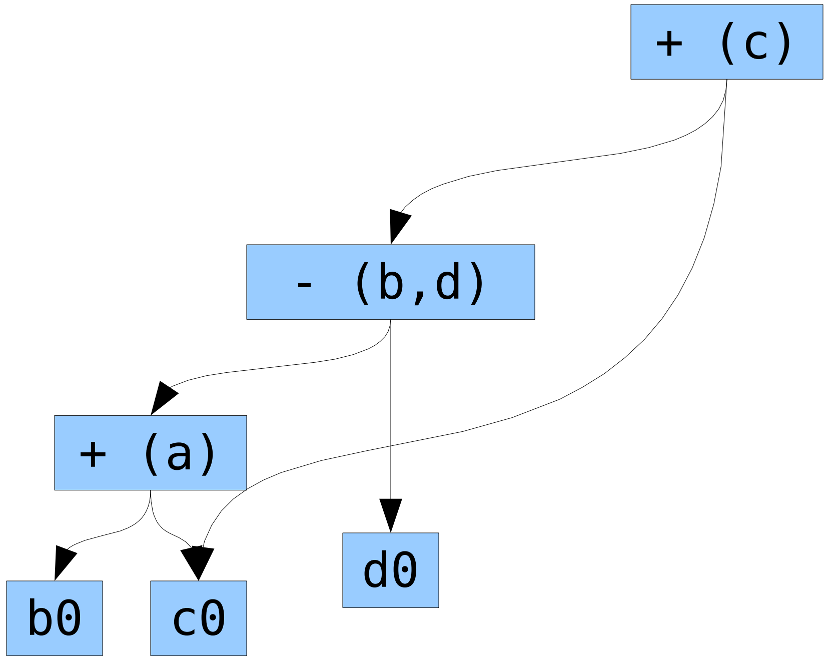
Basic Blocks as DAGs

- Like expressions, we can represent basic blocks as DAGs
  - Create a node for the initial value of each variable that appears in the block
  - For each statement *s*, create a node *N* whose children are the nodes defining the operands of s
  - Each node *N* is labelled by the operation used
  - Each node *N* has a list of variables that it defines
  - A node N is marked as an output node if its value is used outside the basic block

Canada's Capital University

Basic Block Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```



Basic Block Example

- Draw the DAG for this basic block

```
a := b + c
b := b - d
c := c + d
e := b + c
```

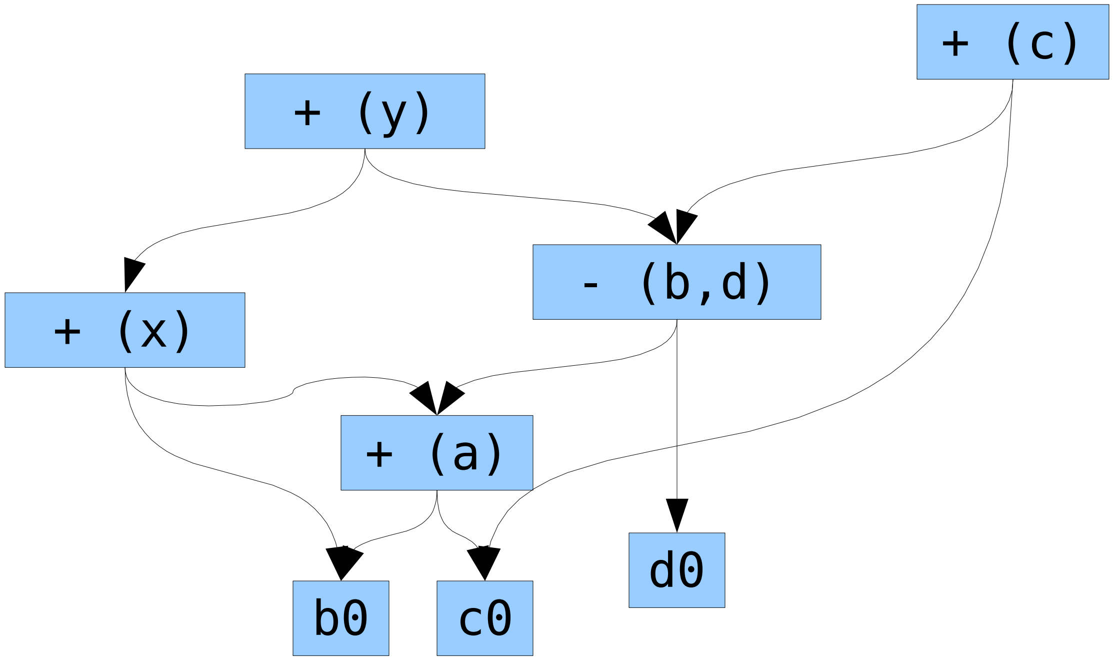
Dead Code Elimination

- *Dead code* is code that computes a value that is never used
- Inspecting the flow graph we can tell if the value of a variable computed within a basic block is used outside that block
  - These are called *live variables* or *live on exit*
- Algorithm
  - While some node *N* has no ancestor and no live variable attached
    - Delete *N* and its outgoing edges

Carleton UNIVERSITY
Canada's Capital University

Dead Code Elimination - Example

- Remove dead code assuming c is the only live variable



Using Algebraic Identities

- The DAG representation can allow *constant folding*
  - When a subexpression involves 2 or more constants its value can sometimes be computed at compile time
- Commutativity: x * y = y * x
  - When creating a node for x * y we should check if there already exists a node named x * y or a node named y * x
- Associativity: (a + b) + c = a + (b + c)

Carleton UNIVERSITY
Canada's Capital University

Associativity

- An example of using associativity

```
Source Code:
a = b + c;
e + c + d + b;
```

```
3-AI (slow):
a := b + c
t := c + d
e := t + b
```

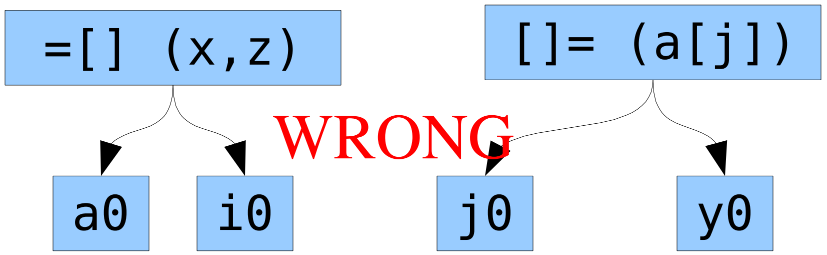
```
3-AI (fast):
a := b + c
e := a + b
```

Carleton UNIVERSITY
Canada's Capital University

Array References

- Array references are a different kind of operator
- Assigning to an array location can change any element of the array

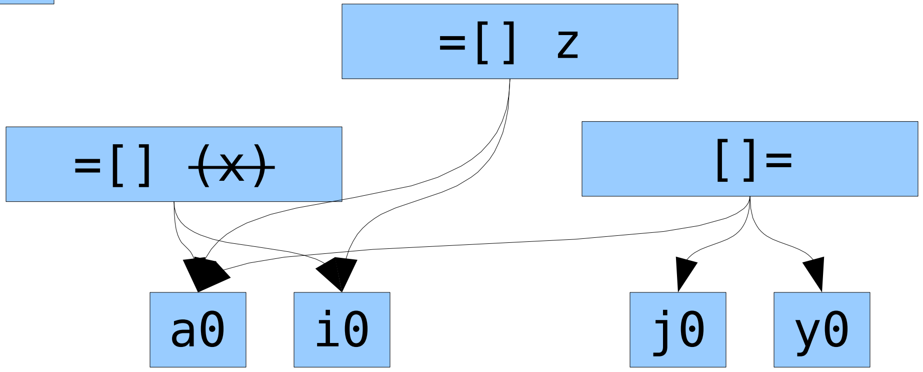
```
x = a[i]
a[j] = y
z = a[i]
```



Arrays References

- The right way
  - x = a[i] creates a node with two children (*a* and *i*) that defines *x*
  - a[j] = y creates a node with 3 children
    - It has the side-effect of killing any node whose value depends on *a*

```
x = a[i]
a[j] = y
z = a[i]
```



Pointer References

- Pointer references are similar to array references
  - assigning to a pointer kills all nodes
  - changing any variable kills all pointer references

```
x = *p
t = a + b
z = *p      ; maybe p points to t
*q = y
x = a + b   ; maybe q points to a or b
```

Carleton UNIVERSITY
Canada's Capital University

Reassembling Basic Blocks

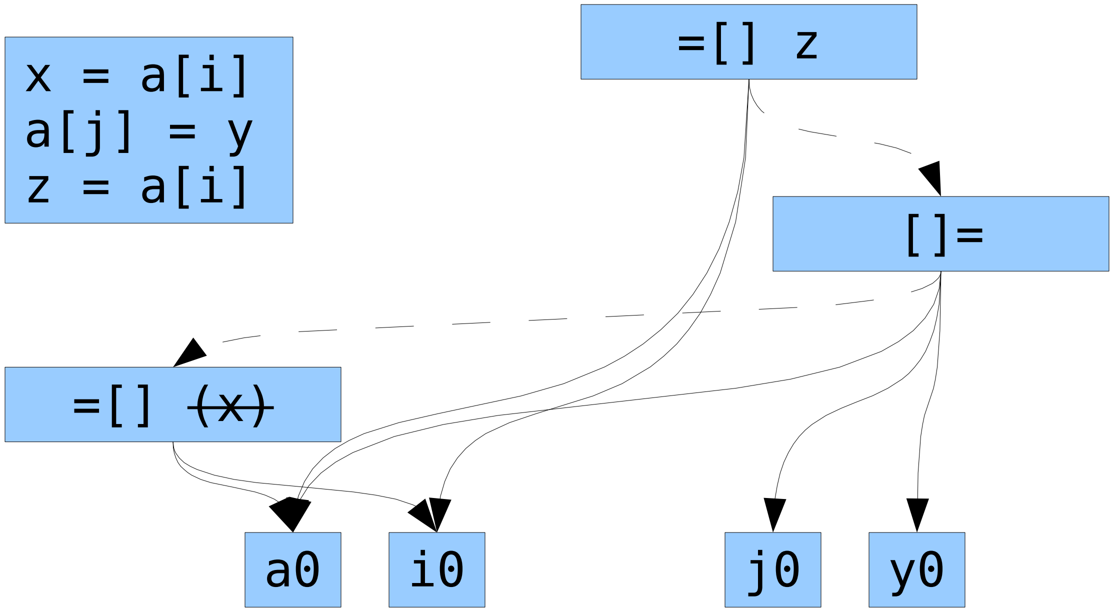
- After converting a basic block into a DAG and applying optimizations we must reassemble the basic block
- Rules:
  1) The order of instructions must obey the order of the DAG
  2) Assigning to an array must follow any previous assignments to the same array
  3) Evaluations of an array must follow any previous assignments to the same array
  4) Any use of a variable must follow any previous indirect (pointer) assignments
  5) Any indirect assignment must follow all previous evaluations of variables

Carleton UNIVERSITY
Canada's Capital University

Rearrangement Order Example

- Extra (dashed) edges implement rules 2-5

```
x = a[i]
a[j] = y
z = a[i]
```



Basic Blocks as DAGs - Examples

- For each of the following lines of C
  - x = a + b * c;
  - x = a / (b+c) - d * (e+f);
  - x = a[i] + 1;
  - a[i] = b[c[i]];
  - *p++ = *q++;
- Write a basic block of 3-address instructions
- Convert the basic block into a DAG
- Convert the DAG back into a basic block

Carleton UNIVERSITY
Canada's Capital University

Peephole Optimization

Carleton UNIVERSITY
Canada's Capital University

Peephole Optimization

- Peephole optimization is done by looking at a small number of lines of code at a time
  - We are looking at a small piece of code (the peephole)
- Some rules are applied to simplify or speed up various patterns
- Can be applied to intermediate code or machine code

Carleton UNIVERSITY
Canada's Capital University

Redundant Loads and Stores

- An adjacent pair of load and store operations (to the same variable) is unnecessary and can be eliminated

```
mov x, R0
mov R0, x
```

```
fload 4
fstore 4
```

```
fload 4
pop
```

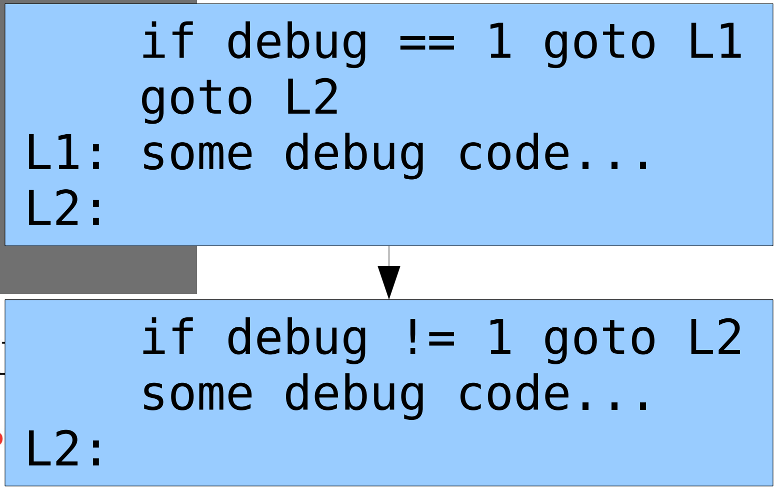
Canada's Capital University

Algebraic Simplification and Reduction in Strength

- Peephole optimization may recognize algebraic simplification and reduction in strength
  - x = x + 1 => inc x [machine instruction]
  - x = x + 0 => unnecessary
  - x = x * 1 => unnecessary
  - x = 2 * x => x = x + x
- To do this in the JVM we need a 3 line peephole

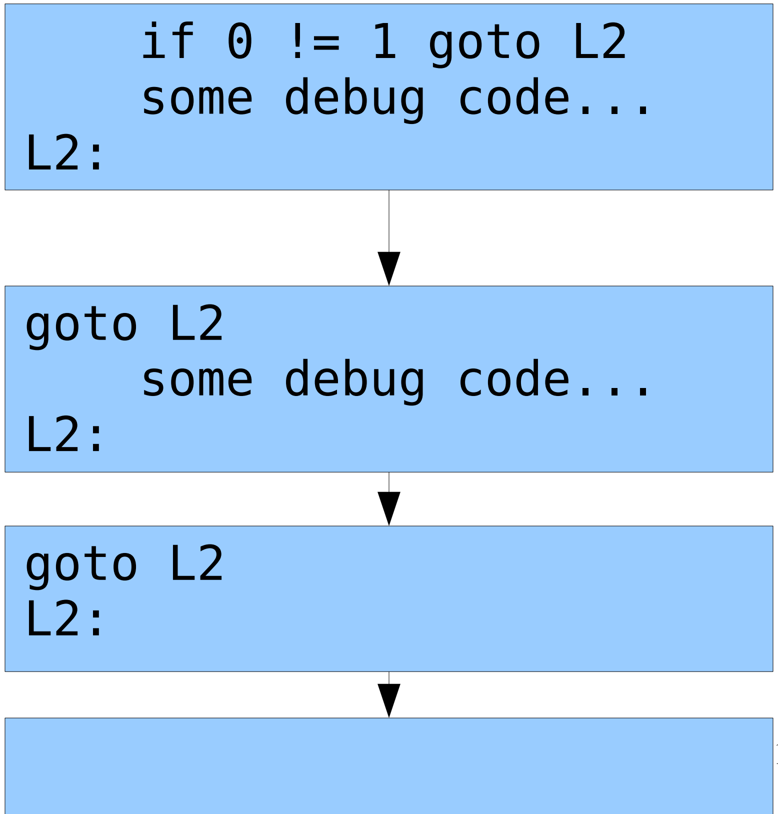
Unreachable Code

- Examining the code surrounding jump statements can often help



```
    if debug == 1 goto L1
    goto L2
L1: some debug code...
L2:
```

```
    if debug != 1 goto L2
    some debug code...
L2:
```



```
    if 0 != 1 goto L2
    some debug code...
L2:
```

```
goto L2
    some debug code...
L2:
```

```
goto L2
L2:
```

Flow of Control Optimization

```
    goto L1
    ...
L1: goto L2
```

```
    goto L2
    ...
L1: goto L2
```

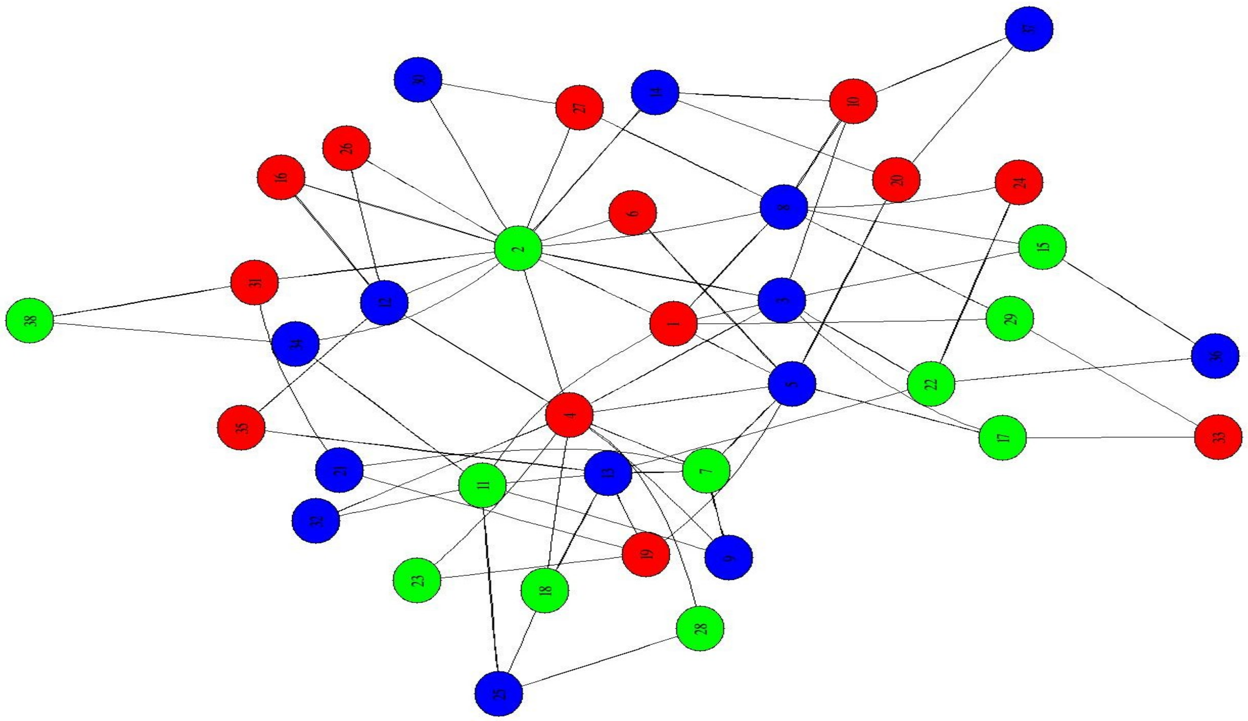
```
    if a < b goto L1
    ...
L1: goto L2
```

```
    if a < b goto L2
    ...
L1: goto L2
```

```
    goto L1
    ...
L1: if a < b goto L2
L3:
```

```
    if a < b goto L2
    goto L3
    ...
L3:
[saves a goto when a < b]
```

Register Allocation by Graph Coloring



Register Allocation By Graph Coloring

- We have seen how to manage registers within a basic block
  - lazy load and store algorithm
  - spill (or flush) a register back to memory only when necessary
- We have seen how to assign *register variables* across blocks (savings calculations)
- Both these methods are heuristics

Canada's Capital University

Register Allocation by Graph Coloring

- Step 1: Generate code as if we have infinitely many registers R1,R2,R3,….
  - R1, R2, R3,... are just like variable names
- Step 2: Create the register interference graph
  - Nodes are R1, R2, R3, ….
  - Two nodes are adjacent if they are active at the same time
- Step 3: Properly color the register interference graph with the "colors" 1,2,3,...,k where *k* is as small as possible

Canada's Capital University

Register Allocation by Graph Coloring

- A virtual register *Ri* that is colored with color c is assigned to the real register *R*(*c* mod *r*) where r is the number of real registers
- If k <= r then there are no conflicts and every variable gets its own register
- Otherwise, conflicts results in extra loads/stores

Carleton UNIVERSITY
Canada's Capital University

The Register Interference Graph

- For each register Ri, determine which lines of which basic blocks the register Ri is active in
- Ri conflicts with Rj if they are active at the same time

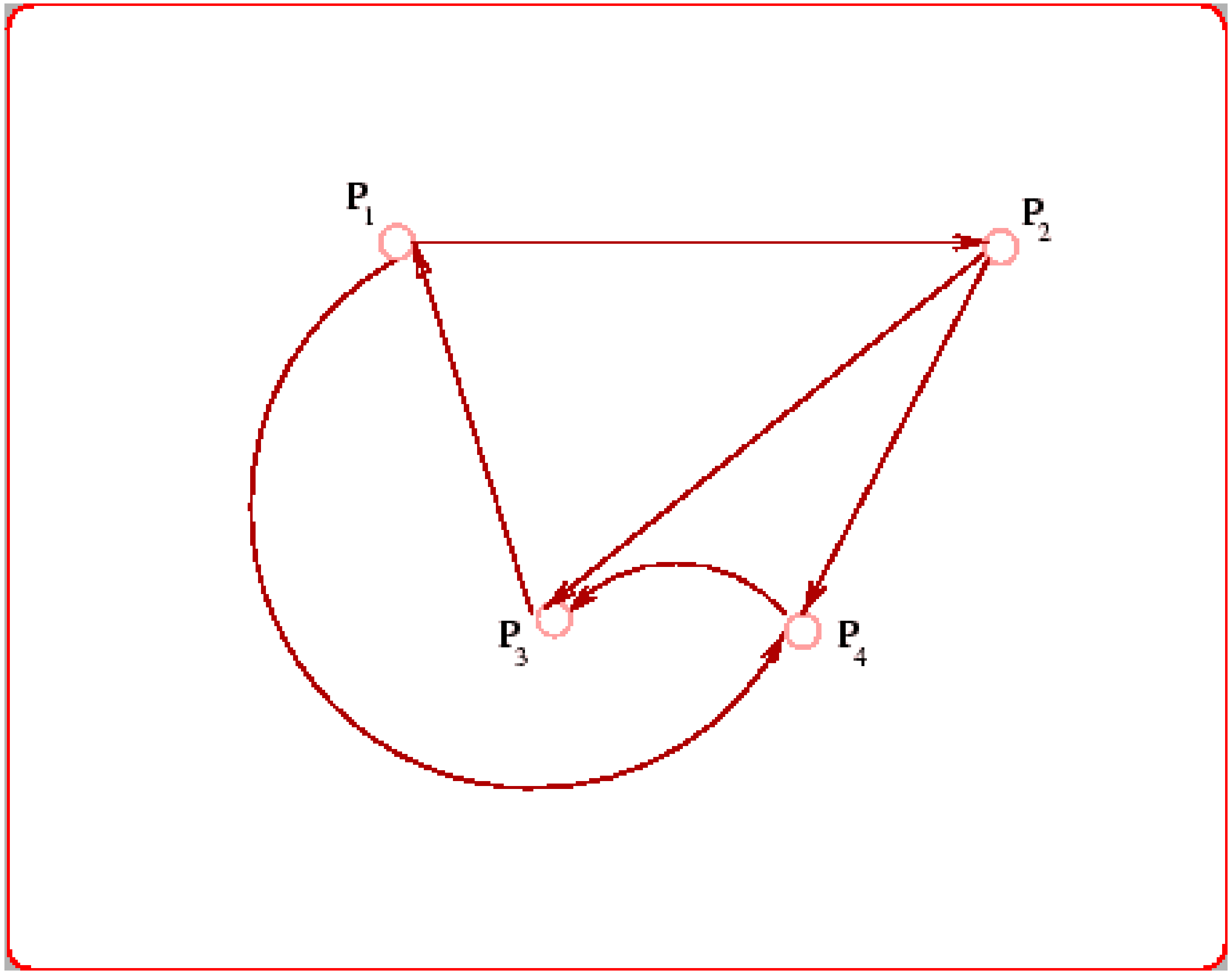
Carleton UNIVERSITY
Canada's Capital University

Coloring a Graph

- Coloring a graph with the fewest colors is NP-hard
- The following heuristic is used to color a graph $G=(V,E)$ with the colors 1,2,3,...
  - Find the vertex v with smallest degree (say d)
  - Recursively color the graph $G \setminus \{v\}$
  - Color v with smallest color not used by one of G's neighbours
- (Note: v will receive color at most d+1)

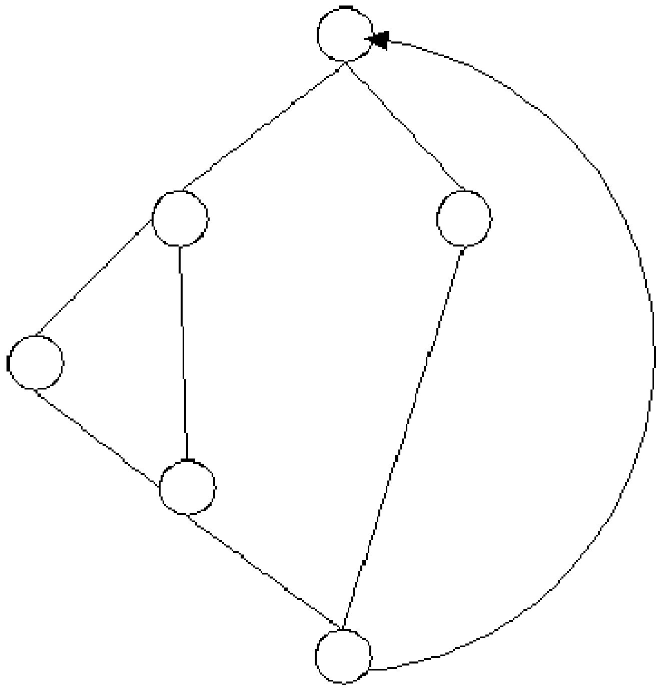
Carleton UNIVERSITY
Canada's Capital University

Graph Color Example

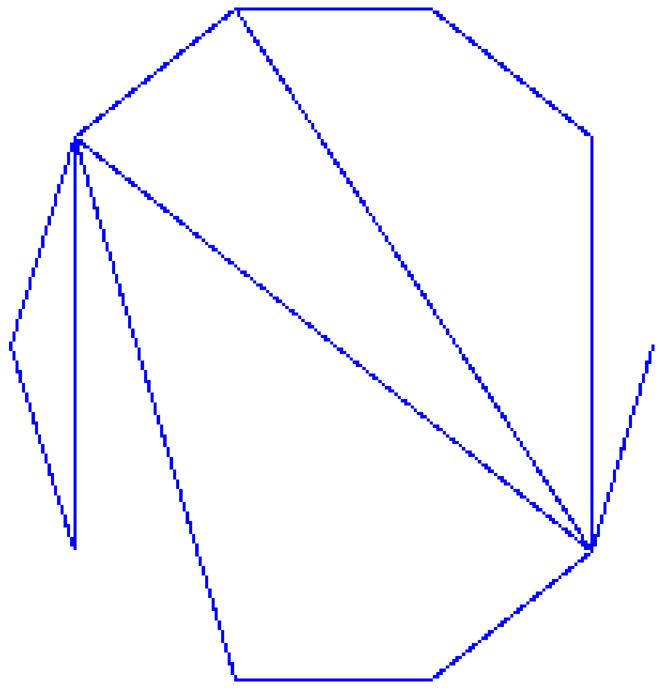


Canada's Capital University

Graph Coloring Example



Graph Coloring Example



Notes on Graph Coloring

- What we are actually trying to find is a *balanced ordering* on the vertices of G
  - If at most d neighbours of v appear before v in the ordering then v will be colored with color at most d+1
- Thorup (1995) has shown that structured programming languages generate register interference graphs with small treewidth
  - Optimally coloring a graph with tree width t can be done in $O(f(t)\ n)$ where $f(t)$ is a really fast growing function of t.
  - Optimal graph coloring can be done in linear time!

Carleton UNIVERSITY

Canada's Capital University

Summary

- We've seen some more code generation and optimization techniques
  - Representing basic blocks as DAGs allows:
    - Common subexpression elimination
    - Dead code elimination
    - Use of algebraic identities
    - Reordering of instructions
  - Peephole optimization allows:
    - Useless code elimination
    - Use of algebraic identities and machine idioms
  - Register allocation by graph color
    - Can do "optimal" register allocation

Carleton UNIVERSITY

Canada's Capital University