```
PriorityRegAlloc(proc, regCount) {
  ig ← buildInterferenceGraph(proc)
  unconstrained ←
     { n ∈ nodes(ig) | neighborCount(n) < regCount }
  constrained ←
     { n ∈ nodes(ig) | neighborCount(n) ≥ regCount }

  while( constrained ≠ ϕ )  {
     for ( c ∈ constrained such that not colorable(c)
             and canSplit(c) ) {
        c1, c2 ← split(c)
        constrained ← constrained - {c}
        if ( neighborCount(c1) < regCount )
             unconstrained ← unconstrained U { c1}
        else  constrained ← constrained U {c1}
        if ( neighborCount(c2) < regCount )
             unconstrained ← unconstrained U { c2}
        else  constrained ← constrained U {c2}
        for ( d ∈ neighbors(c) such that
               d ∈ unconstrained and
                 neighborCount(d) ≥ regCount ){
                  unconstrained ← unconstrained - {d}
                 constrained ← constrained U {d}
     }     } // End of both for loops
```

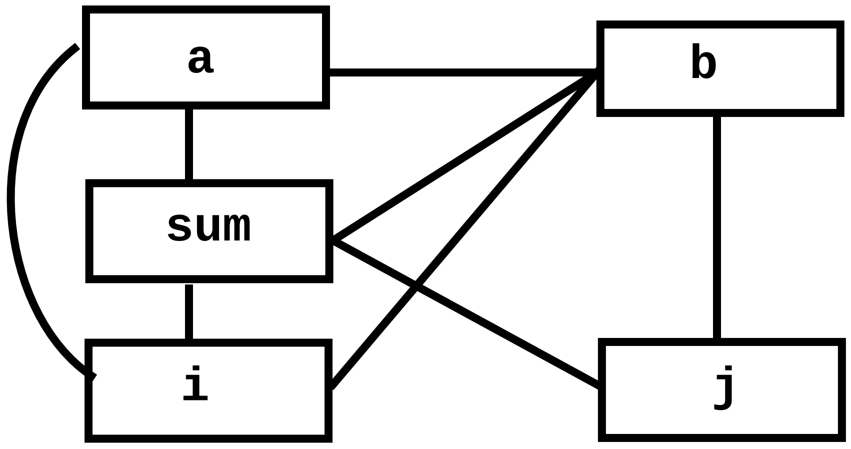
```
      /* At this point all nodes in constrained are
           colorable or can't be split */

      Select p ∈ constrained such that
                  priority(p) is maximized
       if ( colorable(p) )
              color(p)
       else  spill(p)
   } // End of While
  color all nodes ∈ unconstrained
}
```

How to Split a Constrained Node

- There are many possible partitions of a live range; too many to fully explore.
- Heuristics are used instead. One simple heuristic is:
    1. Remove the first basic block (or instruction) of the live range. Put it into a new live range, NR.
    2. Move successor blocks (or instructions) from the original live range into NR, as long as NR remains colorable.
    3. Single Basic Blocks (or instructions) that can't be colored are spilled.

Example

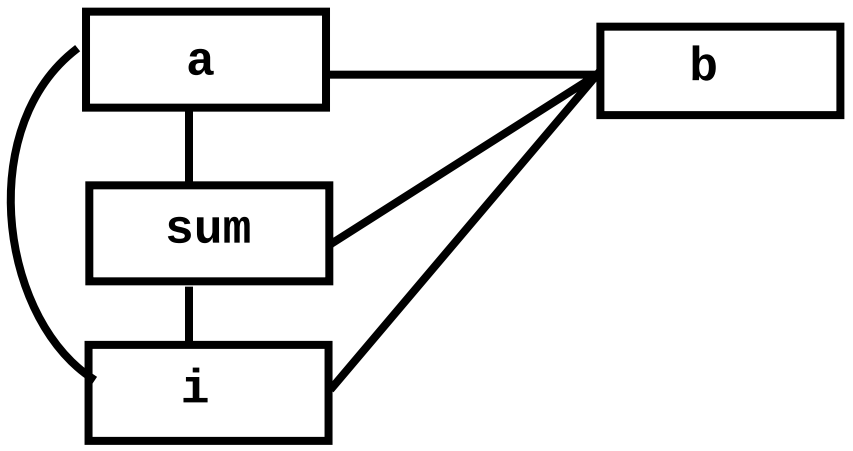
```
int sum(int a[], int b[]) {
  int sum = 0;
  for (int i=0; i<1000; i++)
     sum += a[i];
  for (int j=0; j<1000; j++)
     sum += b[j];
  return sum;
}
```



Assume we want a 3-coloring.

We first simplify the graph by removing unconstrained nodes (those with < 3 neighbors).

Node **`j`** is removed. We now have:

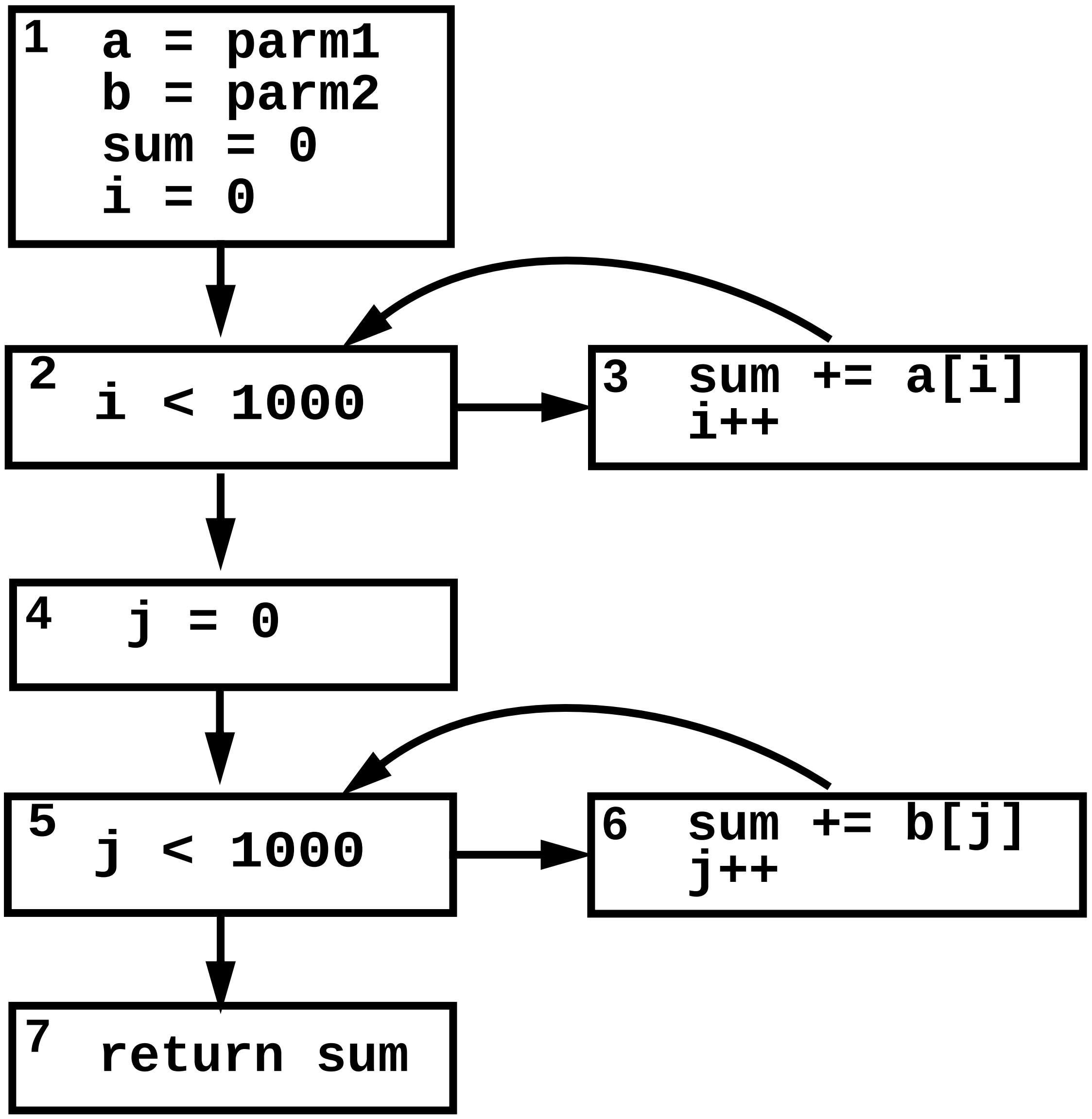


At this point, each node has 3 neighbors, so either spilling or splitting is necessary.

A spill really isn’t attractive as each of the 4 register candidates is used within a loop, magnifying the costs of accessing memory.

Coloring by Priorities

We’ll color constrained nodes by priority values, with preference given to large priority values.



	a	b	sum	i
Cost	11	11	42	41
Cost/Size	11/3	11/6	42/7	41/3

Variables **`i`**, **`sum`** and **`a`** are assigned colors **`R1`**, **`R2`** and **`R3`**.

Variable **`b`** can’t be colored, so we will try to split it. **`b`**’s live range is blocks 1 to 6, with 1 as **`b`**’s entry point.

Blocks 1 to 3 can’t be colored, so **`b`** is spilled in block 1. However, blocks 4 to 6 form a split live range that can be colored (using **`R3`**).

We will reload **`b`** into **`R3`** in block 4, and it will be register-allocated throughout the second loop. The added cost due to the split is minor— a store in block 1 and a reload in block 4.

Choice of Spill Heuristics

We have seen a number of heuristics used to choose the live ranges to be spilled (or colored).

These heuristics are typically chosen using one's intuition of what register candidates are most (or least) important. Then a heuristic is tested and "fine tuned" using a variety of test programs.

Recently, researchers have suggested using machine learning techniques to automatically determine effective heuristics.

In "Meta Optimization: Improving Compiler Heuristics with Machine Learning," Stephenson, Amarasinghe, et al, suggest using *genetic programming* techniques in which priority functions (like choice of spill candidates) are mutated and allowed to “evolve.”

Although the approach seems rather random and unfocused, it can be effective. Priority functions *better than* those used in real compilers have been reported, with research still ongoing.

Interprocedural Register Allocation

The goal of register allocation is to keep frequently used values in registers.

Ideally, we’d like to go to memory only to access values that may be aliased or pointed to.

For example, array elements and heap objects are routinely loaded from and stored to memory each time they are accessed.

With alias analysis, optimizations like Scalarization are possible.

```
for (i=0; i<1000; i++)
     for (j=0; j<1000; j++)
        a[i] += i * b[j];
```

is optimized to

```
for (i=0; i<1000; i++){
     int Ai = a[i];
     for (j=0; j<1000; j++)
        Ai += i * b[j];
     a[i] = Ai;
}
```

Attacking Call Overhead

- Even with good global register allocation calls are still a problem.
- In general, the caller and callee may use the same registers, requiring saves and restores across calls.
- Register windows help, but they are inflexible, forcing all subprograms to use the same number of registers.
- We'd prefer a register allocator that is sensitive to the calling structure of a program.

Reading Assignment

- Read “Minimum Cost Interprocedural Register Allocation,” by S. Kurlander et al. (linked from class Web page).

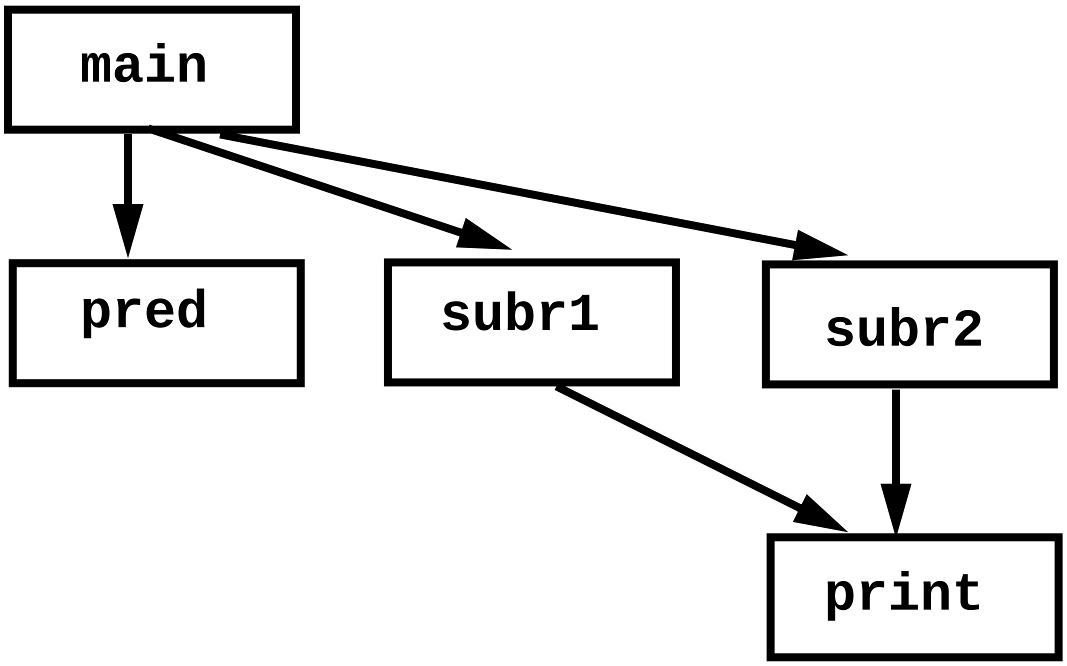
Call Graphs

A *Call Graph* represents the calling structure of a program.

- Nodes are subprograms (procedures and functions).
- Arcs represent calls between subprograms. An arc from A to B denotes that a call to B appears within A.
- For an indirect call (a function parameter or a function pointer) an arc is added to all potential callees.

Example

```
main() {
  if (pred(a,b))
       subr1(a)
  else subr2(b);}

bool pred(int a, int b){
   return a==b; }

subr1(int a){ print(a);}

subr2(int x){ print(2*x);}
```



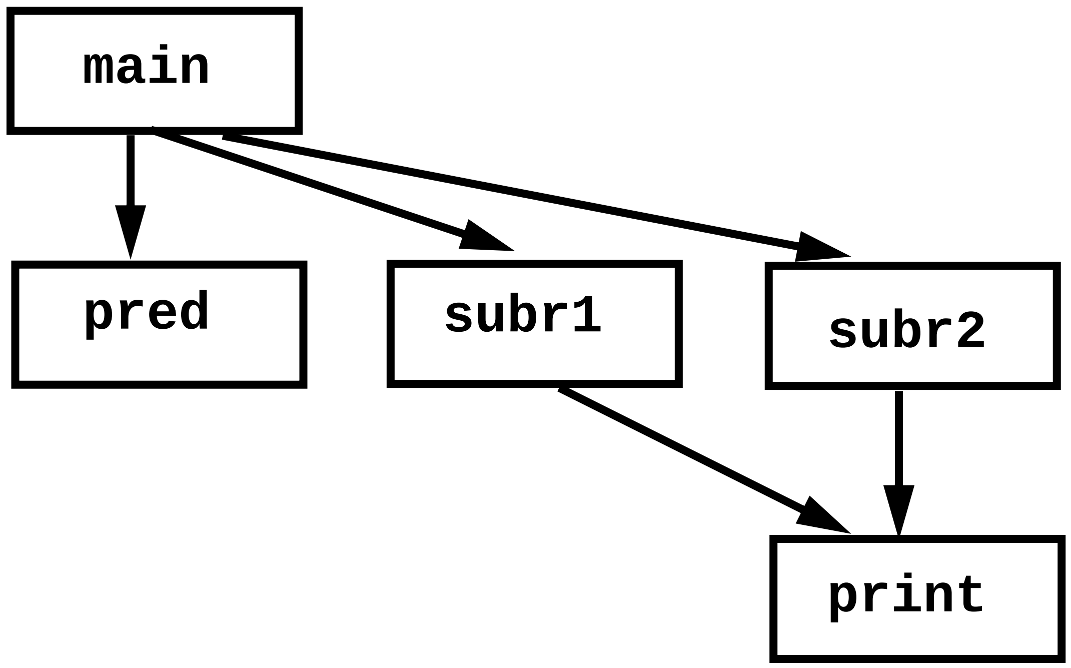
Wall’s Interprocedural Register Allocator

Operates in two phases:

1. Register candidates are identified at the subprogram level.
   Each candidate (a single variable *or* a set of non-interfering live ranges) is compiled as if it *won’t* get a register. At link-time unnecessary loads and stores are edited away *if* the candidate *is* allocated a register.
2. At link-time, when all subprograms are known and available, register candidates are allocated registers.

Identifying Interprocedural Register Sharing

If two subprograms are not connected in the call graph, a register candidate in each can share the same register without any saving or restoring across calls.



A register candidate from **`pred`**, **`subr1`** and **`subr2`** can all share one register.

At the interprocedural level we must answer 2 questions:

1. A local candidate of one subprogram can share a register with candidates of what other subprograms?
2. Which local register candidates will yield the greatest benefit if given a register?

Wall designed his allocator for a machine with 52 registers. This is enough to divide all the registers among the subprograms without any saves or restores at call sites.

With fewer registers, spills, saves and restores will often be needed (if registers are used aggressively within a subprogram).

Restrictions on the Call Graph

Wall limited calls graphs to DAGs since cycles in a call graph imply recursion, which will force saves and restores (why?)

Cost Computations

Each register candidate is given a per-call cost, based on the number of saves and restores that can be removed, scaled by $10^{\text{loop\_depth}}$.

This benefit is then multiplied by the *expected* number of calls, obtained by summing the total number of call sites, scaled by loop nesting depth.

Grouping Register Candidates

We now have an estimate of the benefit of allocating a register to a candidate. Call this estimate

cost(candidate)

We estimate potential interprocedural sharing of register candidates by assigning each candidate to a *Group*.

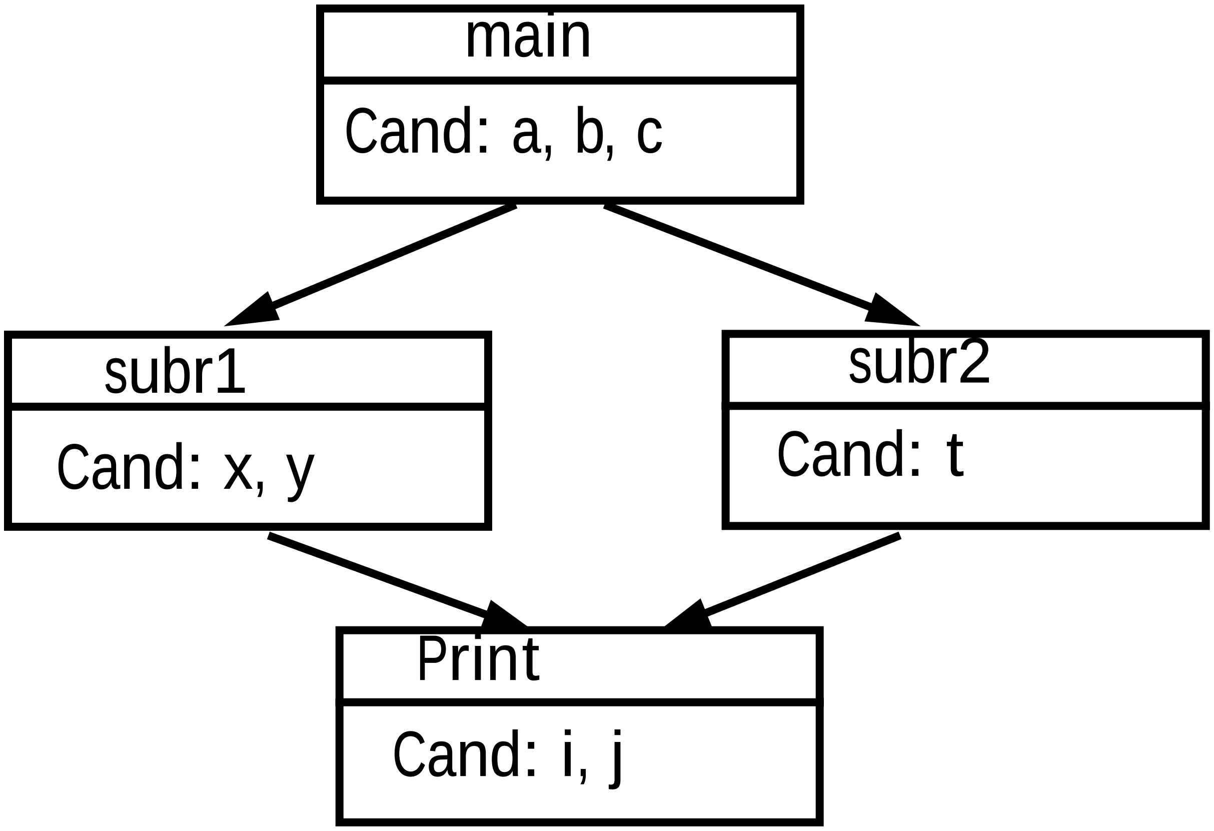
All candidates within a group can share a register. No two candidates in any subprogram are in the same group.

Groups are assigned during a reverse depth-first traversal of the call graph.

```
 AsgGroup(node n) {
    if (n is a leaf node)
        grp = 0
    else { for (each c ∈ children(n)) {
                AsgGroup(c) }
        grp =
            1+ Max (Max group used in c)
            c ∈ children(n)
    }
    for (each r ∈ registerCandidates(n)){
          assign r to grp
          grp++  }
 }
```

Global variables are assigned to a singleton group.

Example



At Print: grp(i)=0, grp(j)=1

At subr1: Max grp used in print is 1
grp(x)=2, grp(y)=3

At subr2: Max grp used in print is 1
grp(t)=2

At main: Max grp used in children is 3
grp(a)=4, grp(b)=5, grp(c)=6

If A calls B (directly or indirectly), then none of A’s register candidates are in the same group as any of B’s register candidates.

This *guarantees* that A and B will use different registers.

Thus no saves or restores are needed across a call from A to B.