Optimal Basic Block Reordering via Hammock Decomposition

Oleg Medvedev
St. Petersburg State University
Russia

introduction

- a problem of control flow graph basic blocks reordering to eliminate some pipeline flushes and unconditional jumps
- its informal reduction to an NP-hard problem of covering an edge-weighted graph with a set of non-intersecting paths of maximal total weight
- a way to use <u>hammock decomposition</u> of a control flow graph to solve this problem <u>precisely</u> using a simple branch-and-bounds algorithm to solve small subproblems

processor pipeline

time →

	0	1	2	3	4	5
Fetch	add	shl				
Decode		add	shl			
Get args			add	shl		
Execute				add	shl	
Store results					add	shl

- instruction execution is separated to many stages
- different stages for different instructions can be executed in parallel, which gives a performance boost

conditional/indirect jump instruction leads to “pipeline bubbles” = lost cycles

time →

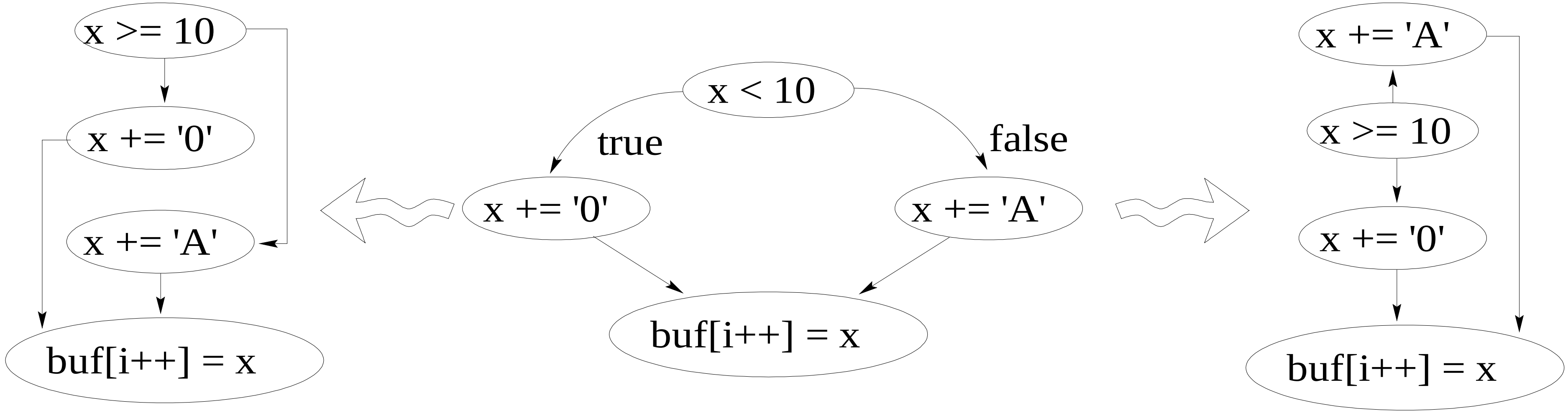
	0	1	2	3	4	5	6
Fetch		je L0	add	adc	inc	mov	
Decode			je L0	add	adc	inc	mov
Get arguments	xor			je L0	add		
Execute		xor			je L0		
Store result			xor			je L0	

```
    xor r0, r0
    je L0
    add r1, r2
    adc r3, r4
    inc r5
L0:
    mov r0, [r3]
```

all processors try to predict jump behaviour to fix this

- simple prediction heuristics, like “every branch is not taken”, or “backward jump is usually taken”
- prediction based on results of previous executions of the jump
  - branch target buffer (BTB)
  - BTB + history patterns
  - perhaps something more sophisticated

but part of this work can be statically done by compiler

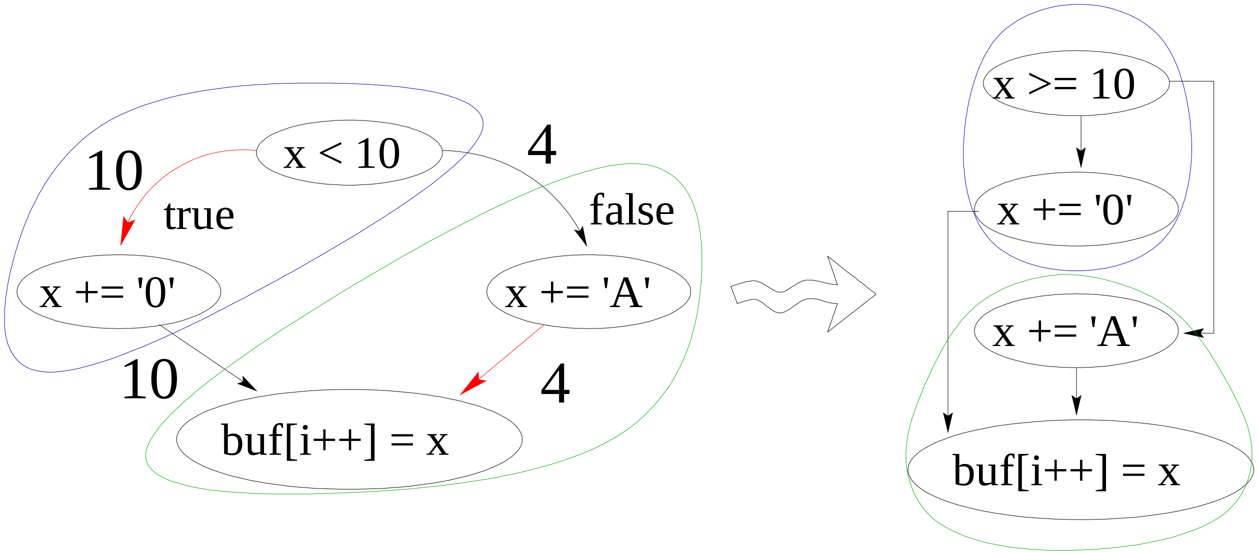


- we may reorder basic blocks of a control flow graph of every function to make the most frequent jumps be “fall-through”

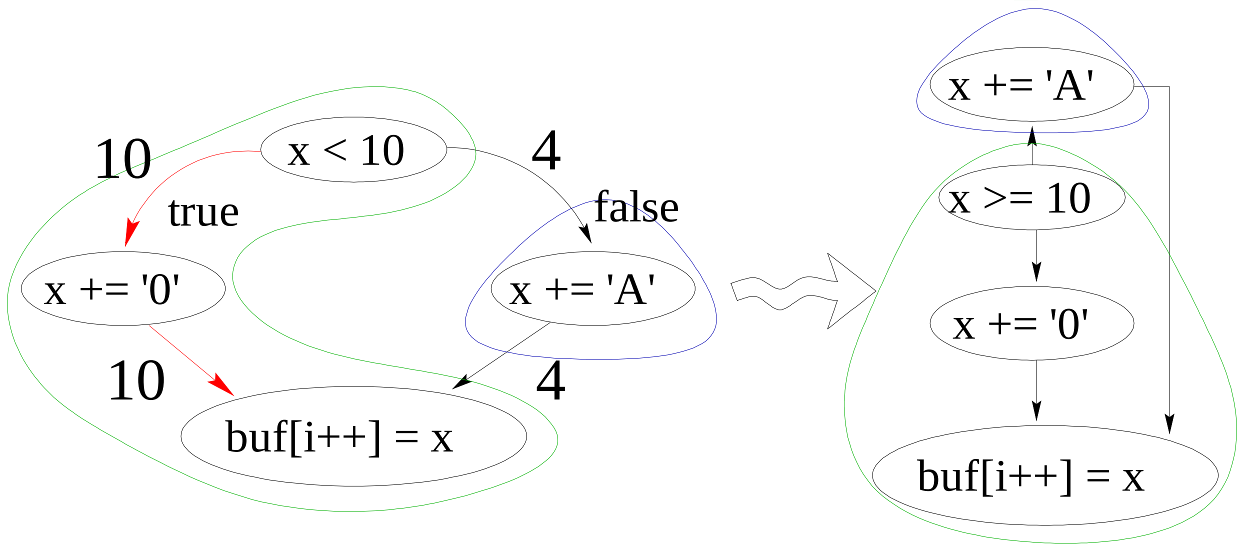
basic block reordering

- consider a CFG with execution counts on edges
- the penalty to be minimized is the total amount of cycles spent on execution of jump instructions on the same tests for which the profiling information was gathered
- to minimize the penalty we find a covering of a CFG by a set of non-intersecting paths with maximal total weight and make all the edges in these paths fall-through

covering and reordering



= penalty of 4 pipeline flushes + 10 unconditional jumps

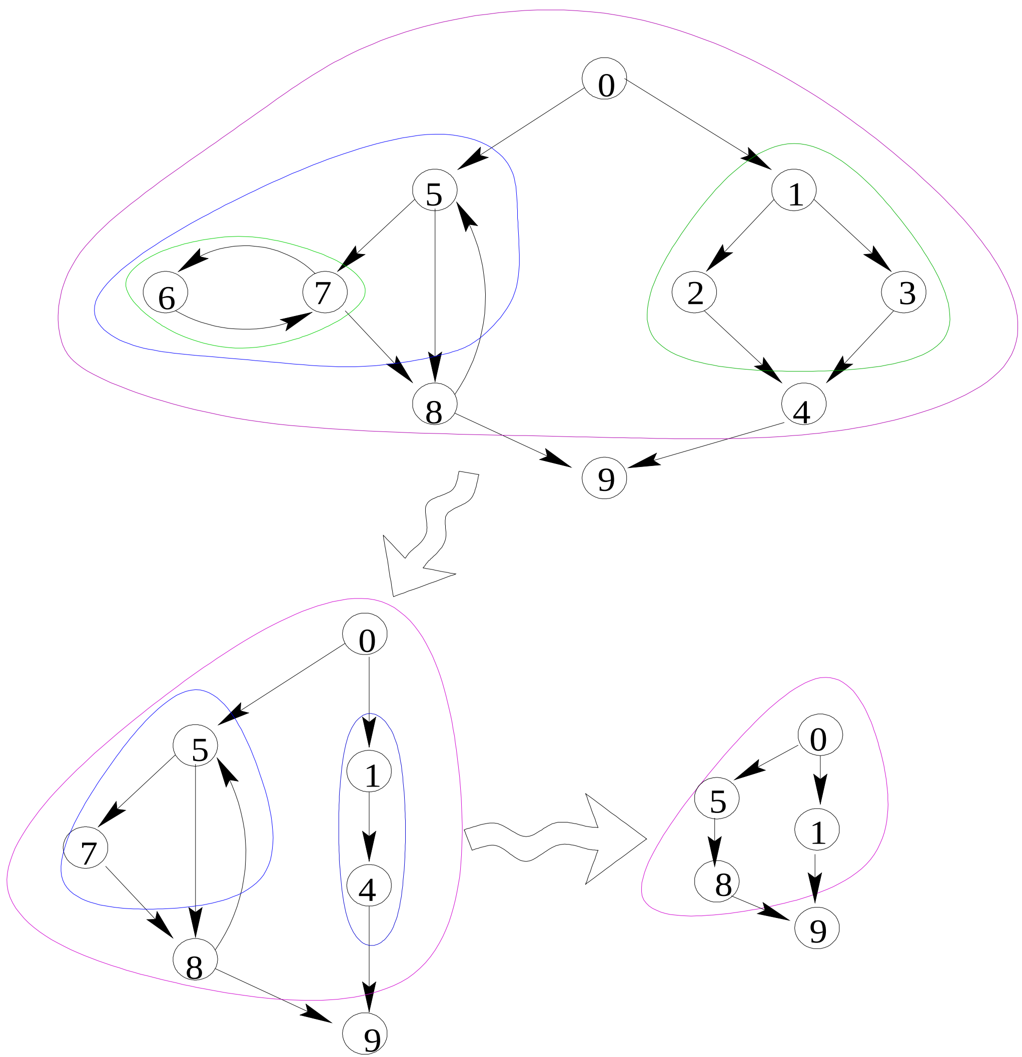


= penalty of 4 pipeline flushes + 4 unconditional jumps

complexity of the problem

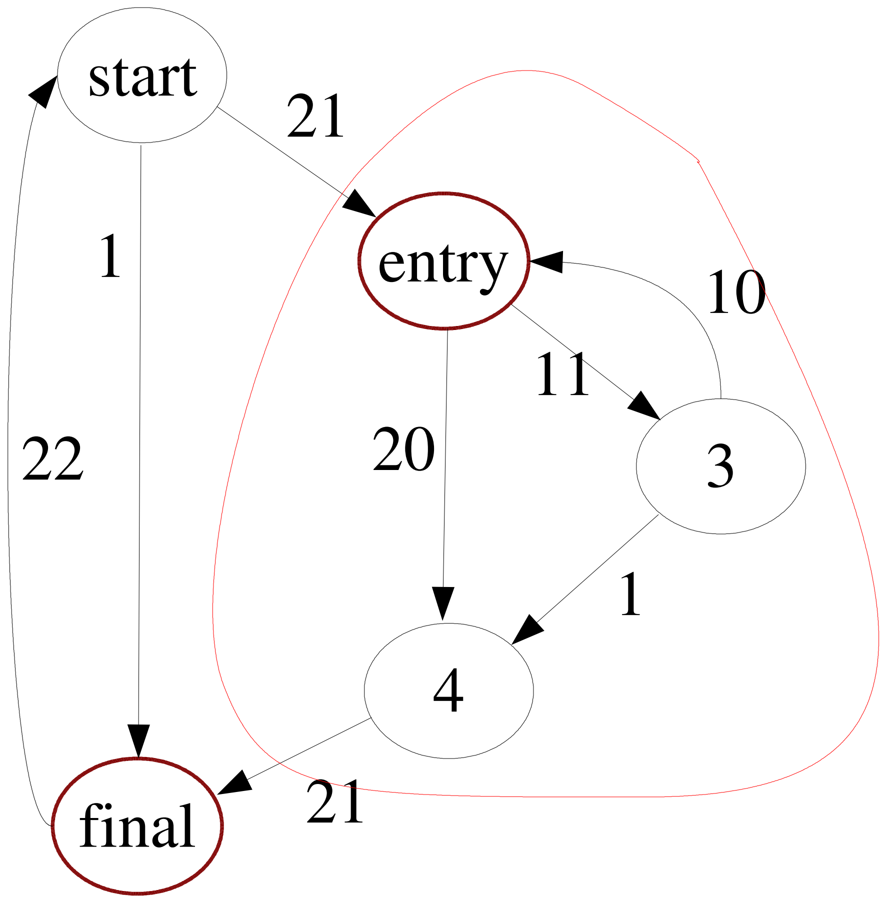
- covering a weighted graph with non-intersecting paths is NP-hard, because we can simply check if the graph has a hamiltonian path by solving this problem
- it is usually solved by a greedy approximation algorithm
- one can implement a simple exponential branch-and-bounds algorithm that finds an optimal answer for tiny programs, but never finishes even for GNU zip (gzip)
- thus, one has to use a kind of decomposition of a graph to find the precise solution fast

what is a hammock

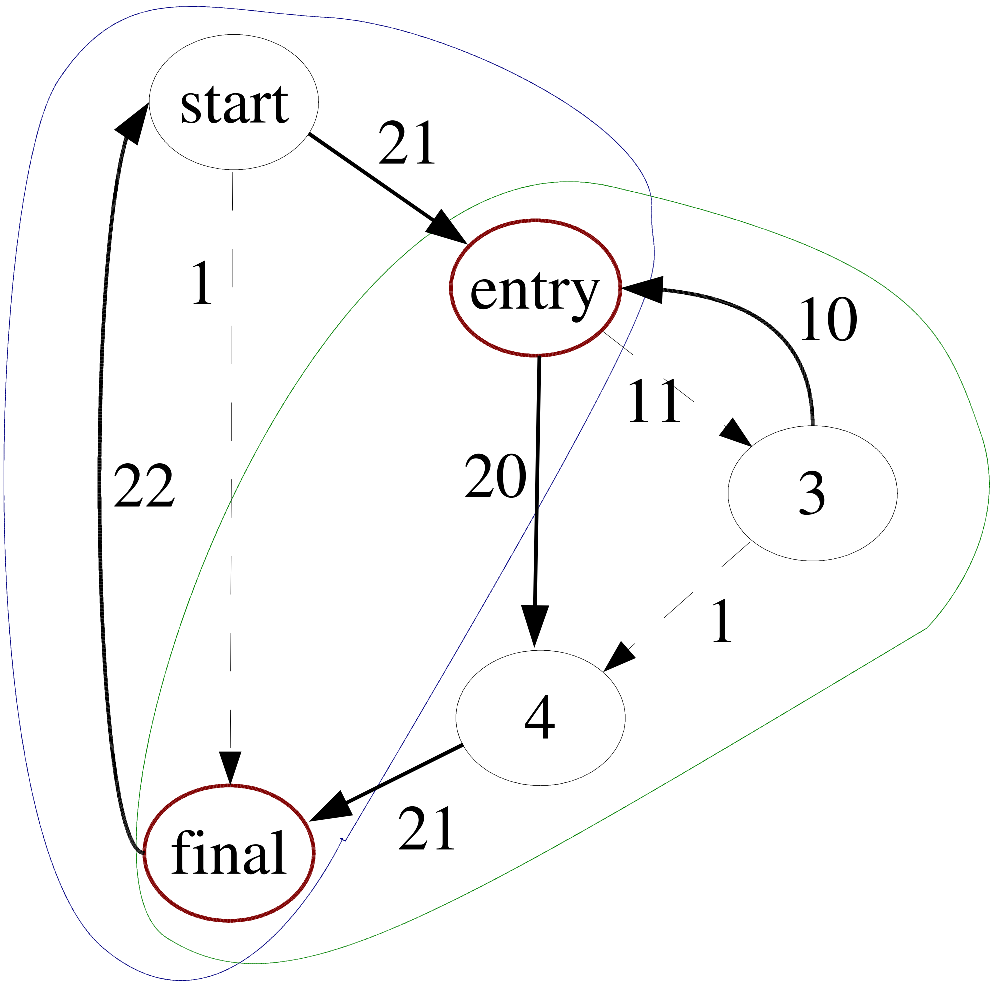


- a CFG subgraph with one entry node and at most one final node
- in other words, a fragment useful for decomposing graph problems
- all hammocks in a graph can be found in $O(E^2)$ time

general decomposition idea: example graph

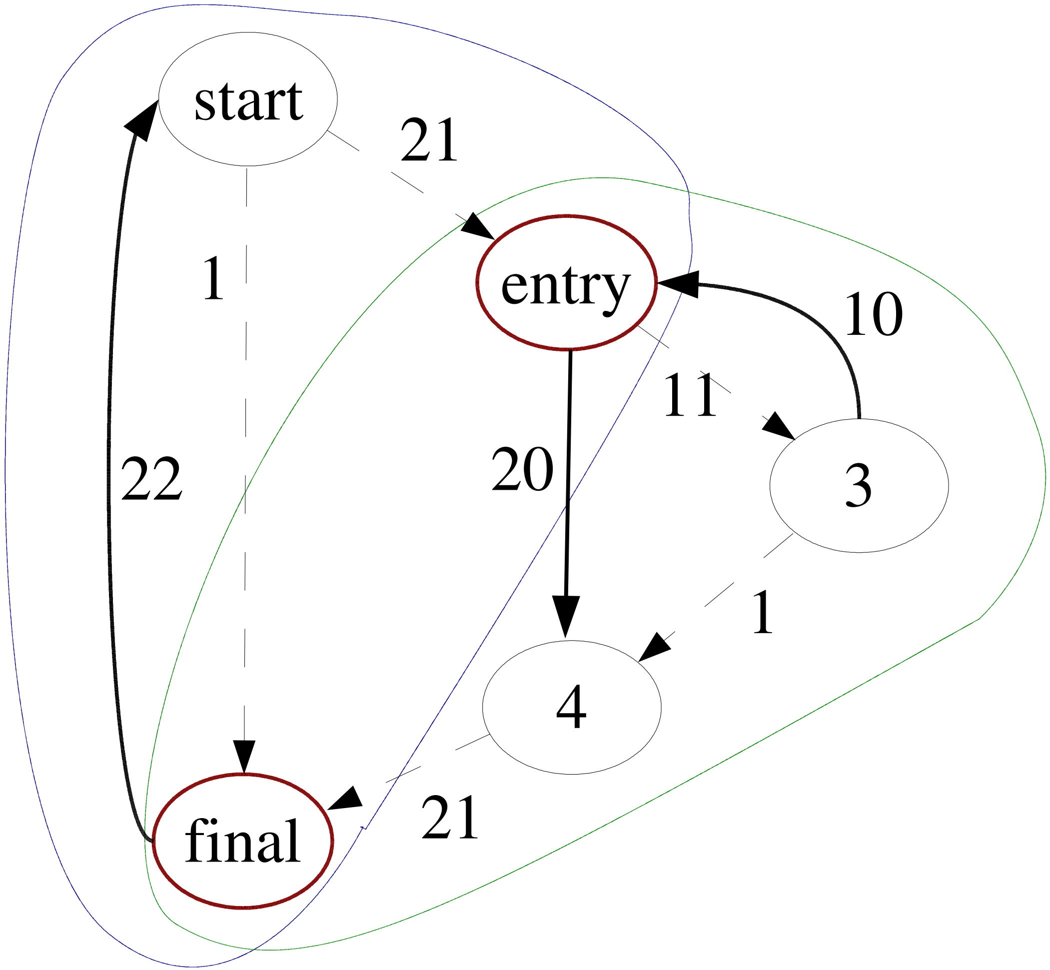


general decomposition idea



- we have a graph with a hammock inside it
- we know how to find an optimal path covering for smaller graphs
- we want to obtain a covering for the whole graph by combining its “hammock” and “non-hammock” optimal coverings
- we cannot just paste the two coverings, but have to consider several ways to cover the entry and final nodes instead

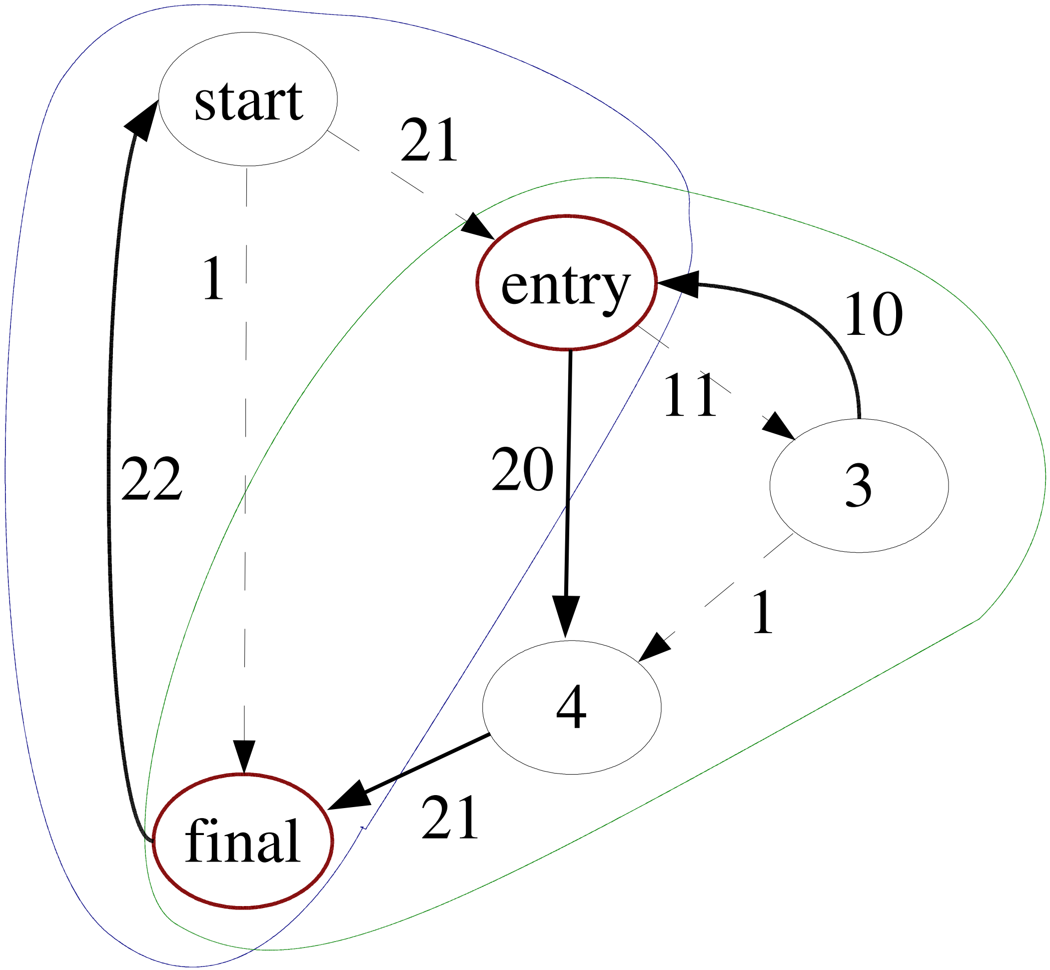
decomposition: case 1



a total weight of the covering is 52 here

- we find an optimal covering of the green graph that is **allowed** to enter the entry node and is **not allowed** to enter the final one
- we find an optimal covering of the blue graph that is **allowed** to enter the final node and is **not allowed** to enter the entry one
- we paste the two coverings and consider the result as a candidate optimal covering for the whole graph

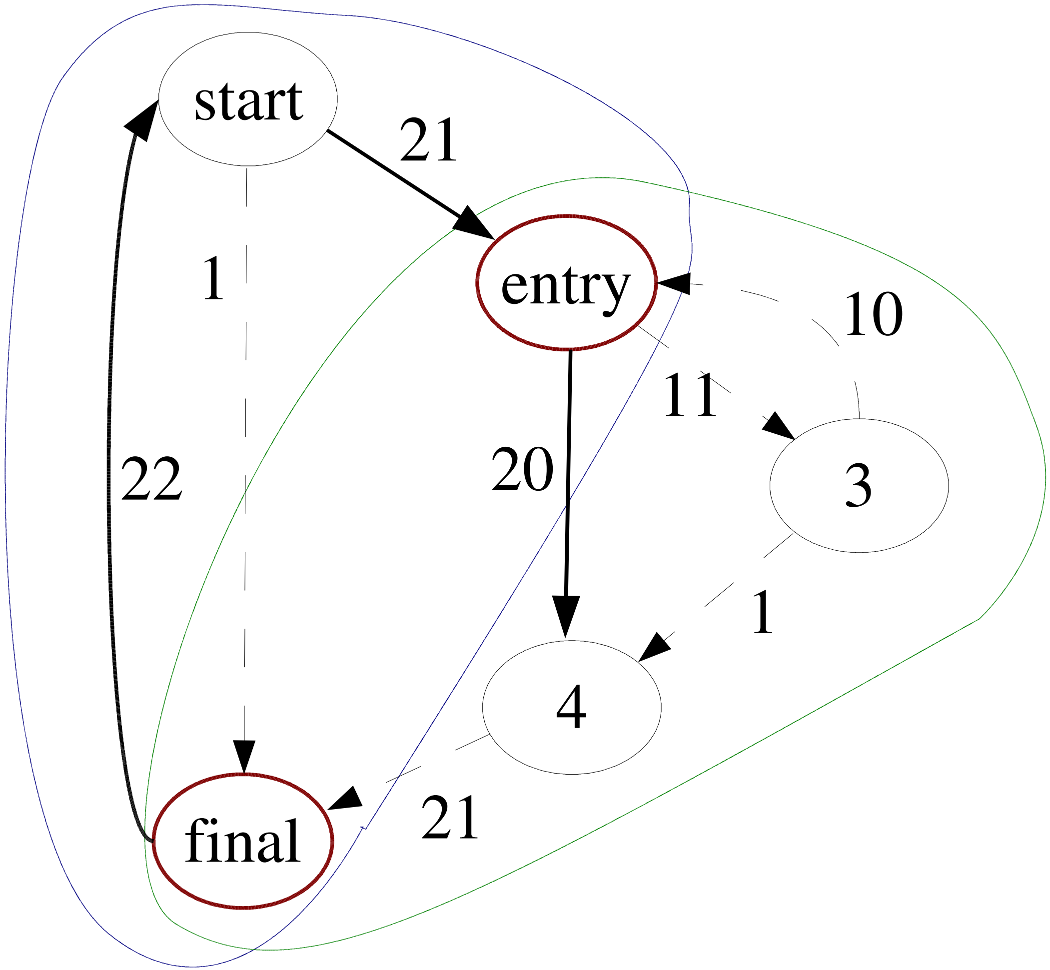
decomposition: case 2



a total weight of the covering is 73 here

- we find an optimal covering of the green graph that is **allowed** to enter the <u>entry</u> node and is **allowed** to enter the <u>final</u> one
- we find an optimal covering of the blue graph that is **not allowed** to enter the <u>final</u> node and is **not allowed** to enter the <u>entry</u> one
- we paste the two coverings and consider the result as a candidate optimal covering for the whole graph

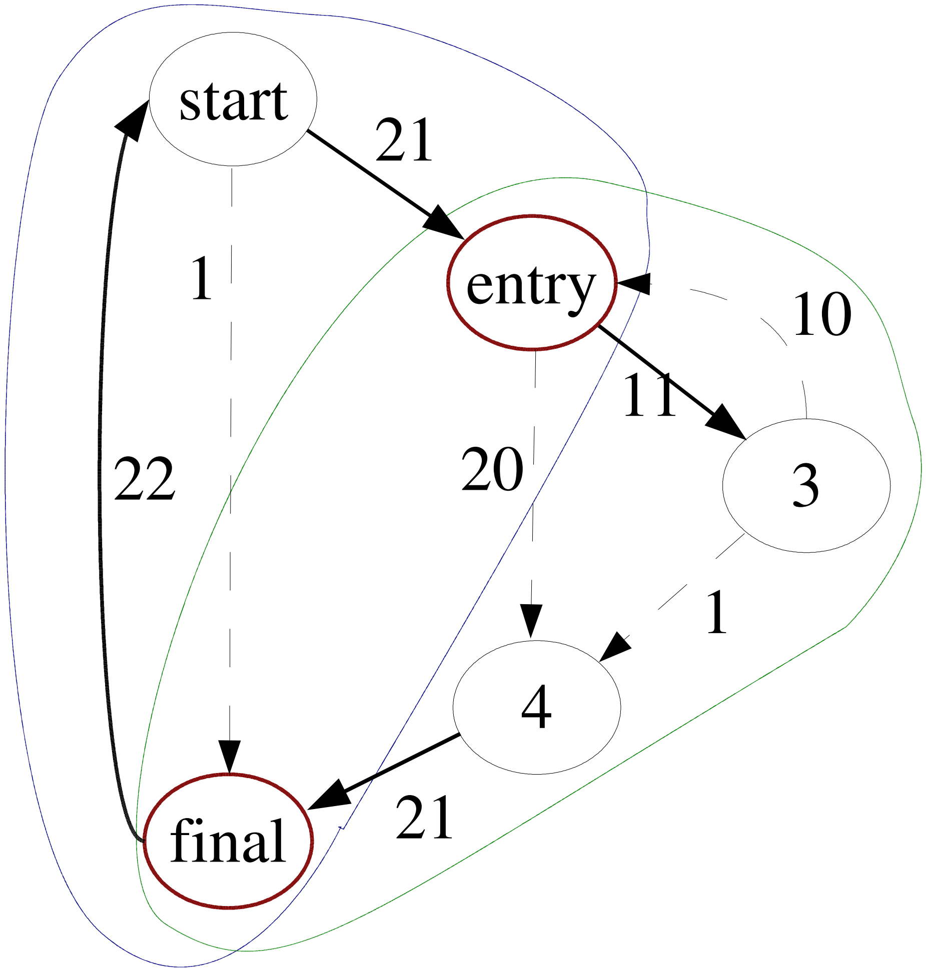
decomposition: case 3



- we try an optimal covering of the green graph that is **not allowed** to enter the <u>entry</u> node and is **not allowed** to enter the <u>final</u> one
- together with an optimal covering of the blue graph that is **allowed** to enter the <u>final</u> node and is **allowed** to enter the <u>entry</u> one

a total weight of the covering is 63 here

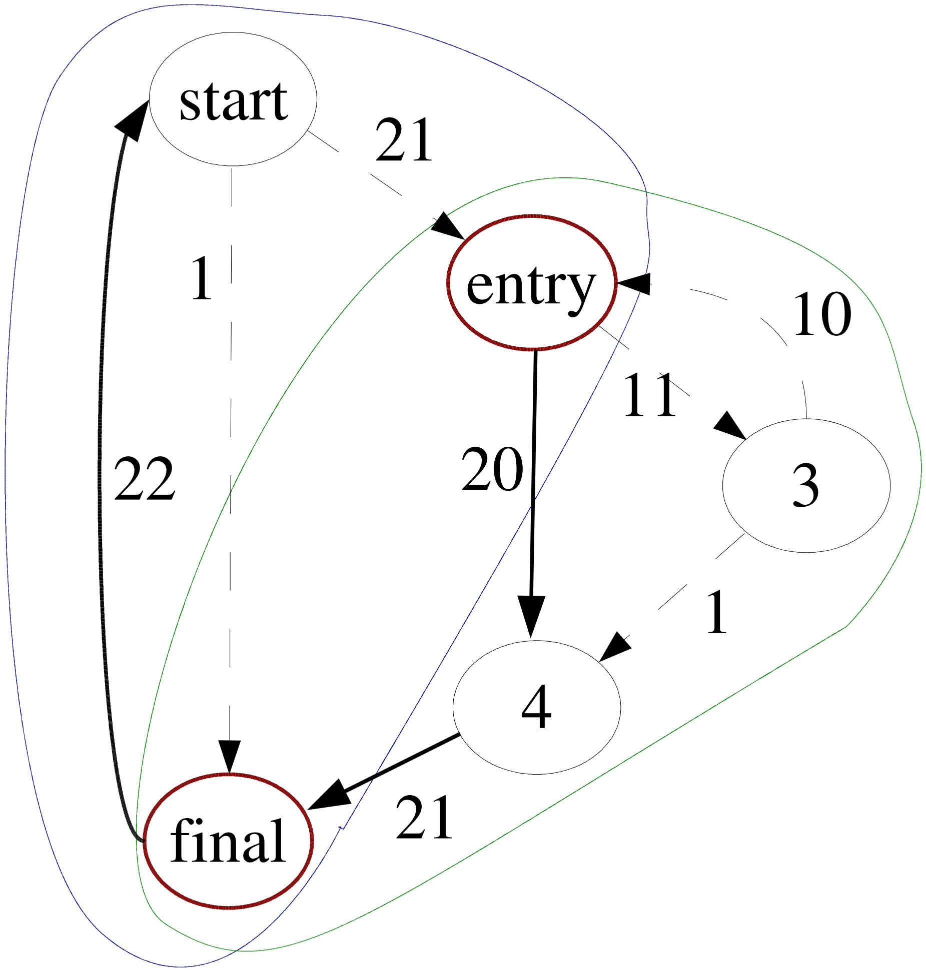
decomposition: case 4



a total weight of the covering is 75 here

- we try an optimal covering of the green graph that is not allowed to enter the entry node and but is allowed to enter the final one and is not allowed to contain a path from the entry node to the final one
- together with an optimal covering of the blue graph that is not allowed to enter the final node but is allowed to enter the entry one and is allowed to contain a path from the final node to the entry one
- we escape considering cycles by this trick

decomposition: case 5



a total weight of the covering is 63 here

- we try an optimal covering of the green graph that is not allowed to enter the entry node and but is allowed to enter the final one and is <u>allowed</u> to contain a path from the entry node to the final one
- together with an optimal covering of the blue graph that is not allowed to enter the final node but is allowed to enter the entry one and is <u>not allowed</u> to contain a path from the final node to the entry one
- we escape considering cycles by this trick

decomposition

the best of the five coverings described above is an optimal covering for the graph

using decomposition together with branch-and-bounds

- find all the hammocks and process them in an increasing order of their sizes
- for each hammock
  - find optimal coverings for the 5 cases described above using branch-and-bounds
  - replace the hammock by an edge from the entry to the final node marked with those 5 weights
- the whole graph is the biggest hammock

implementation and results

- the algorithm was implemented in OCaml, it uses the Pranlib control flow graph library (http://oops.tepkom.ru/projects/pranlib)
- it processes gcc assembler output
- tested on several open-source programs (gzip, bzip2 (de-)compressors, gnuchess chess player, awhttpd mini web-server), total of about 66 graphs with more than 80 nodes

implementation and results

- an average improvement of the exact algorithm over the usual approximation one is 3% in terms of processor cycles lost for unconditional jumps/pipeline flushes (the penalty function)
- the algorithm works as fast as the rest of the compilation process with exception of 9% graphs which don't have enough hammocks
- this is much better than what is achieved by the same branch-and-bounds without decomposition

table of all results

name	gcc time	opt time	number of big graphs	avg size of big graphs	max size of good graph	number of bad graphs	avg size of bad graphs
gzip	1.32	0.52	8	110.75	149	0	0
bzip2	2.24	7.87	16	240.25	648	1	648
awhttpd	0.78	1.39	2	84	84	0	0
gnuchess	6.681	12.84	40	191.5	552	5	210.2

Future work

- we may try to use more complicated subgraphs than hammocks to decompose the problem
- for example, we may remove all the cross edges from the graph, find hammocks in the result, add the edges back and consider the resulting fragments

Thank you