Optimizing for Space and Time Usage with Speculative Partial Redundancy Elimination

Bernhard Scholz, University of Sydney, Australia
Nigel Horspool, University of Victoria, Canada
Jens Knoop, Vienna University of Technology, Austria

Optimizing for Space and Time Usage with SPRE

Bernhard Scholz, University of Sydney, Australia
Nigel Horspool, University of Victoria, Canada
Jens Knoop, Vienna University of Technology, Austria

Overview

- SPRE is normally a speed optimization ...

- ... but SPRE may significantly increase program size.

- We present a new SPRE approach where the objective function is a linear combination of space and time.

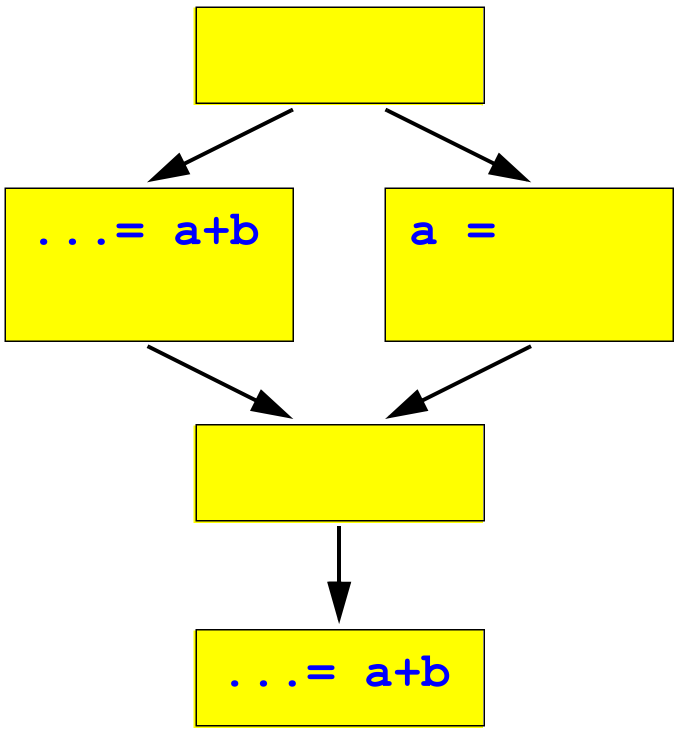
  (Problem maps to the well-known maximum flow problem in net-works.)

- An objective function which combines space and time can come close to the optimal result for both space and time when optimized sepa-rately.

Introduction

Partial Redundancy Elimination

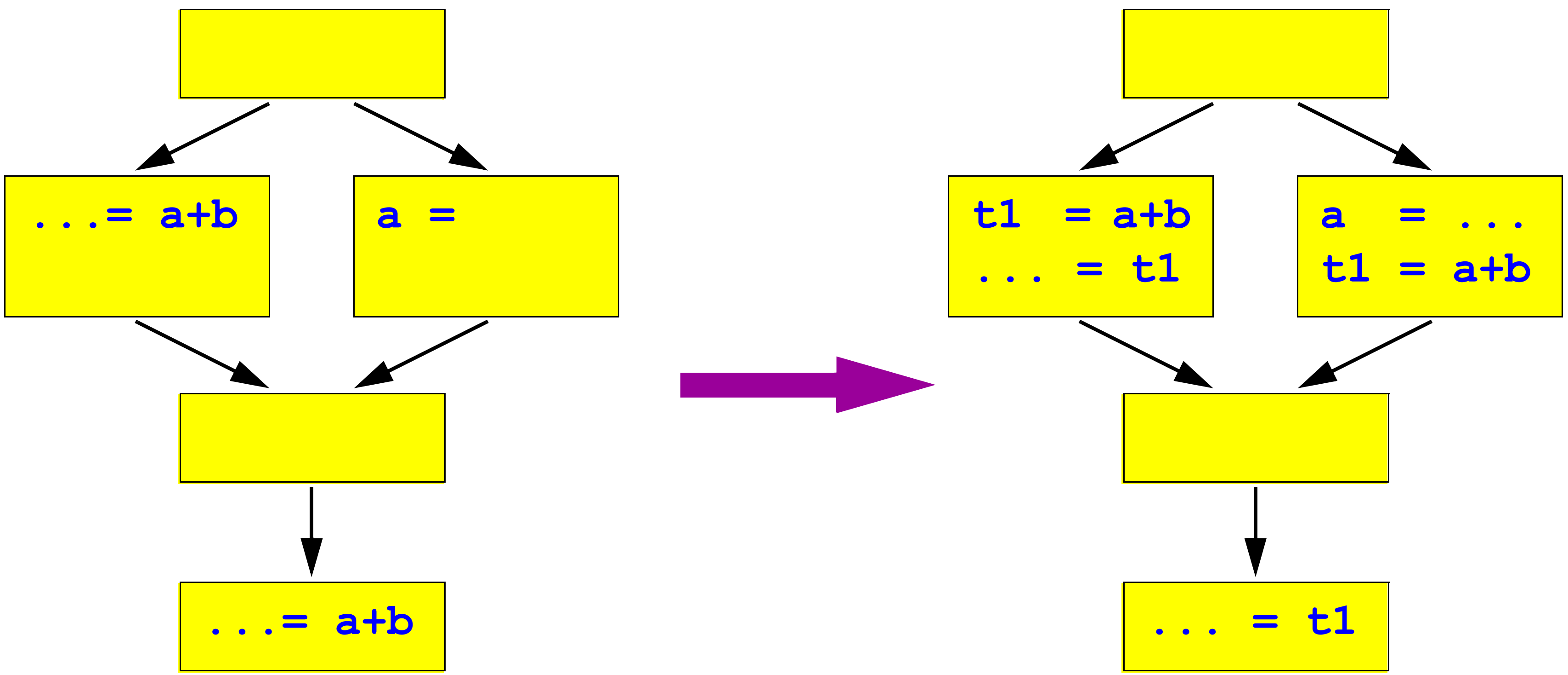
... is a generalization of code motion (Morel and Renvoise, 1979)



Introduction

Partial Redundancy Elimination

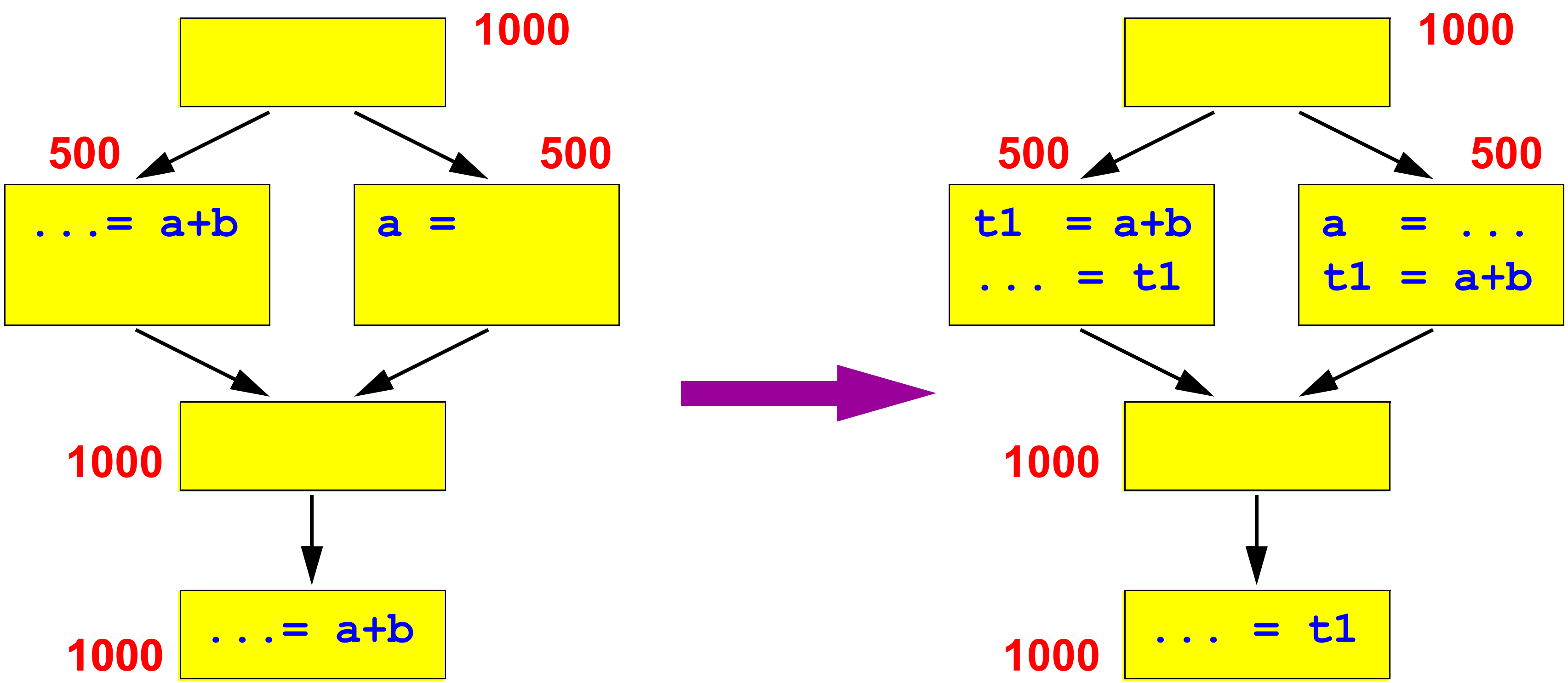
... is a generalization of code motion (Morel and Renvoise, 1979)



Introduction

Partial Redundancy Elimination

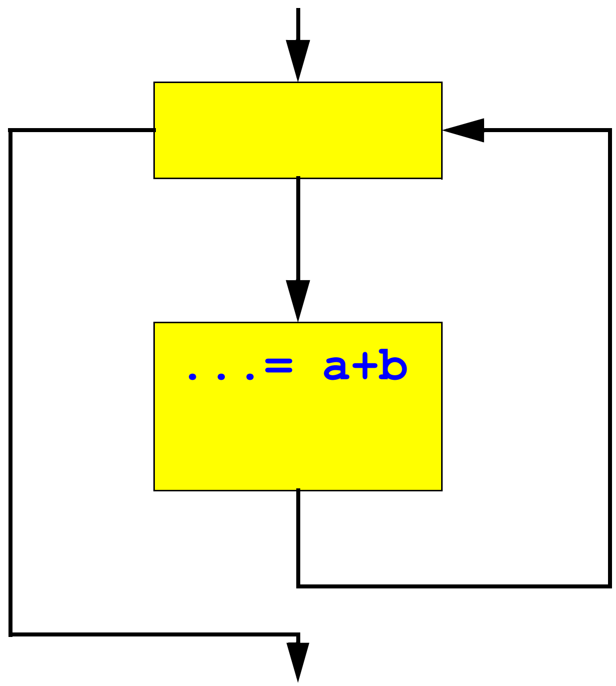
... is a generalization of code motion (Morel and Renvoise, 1979)



Partial Redundancy Elimination

... is also very conservative. An expression *e* can be inserted at a point P only if every path starting from P uses *e*. This restriction guarantees:

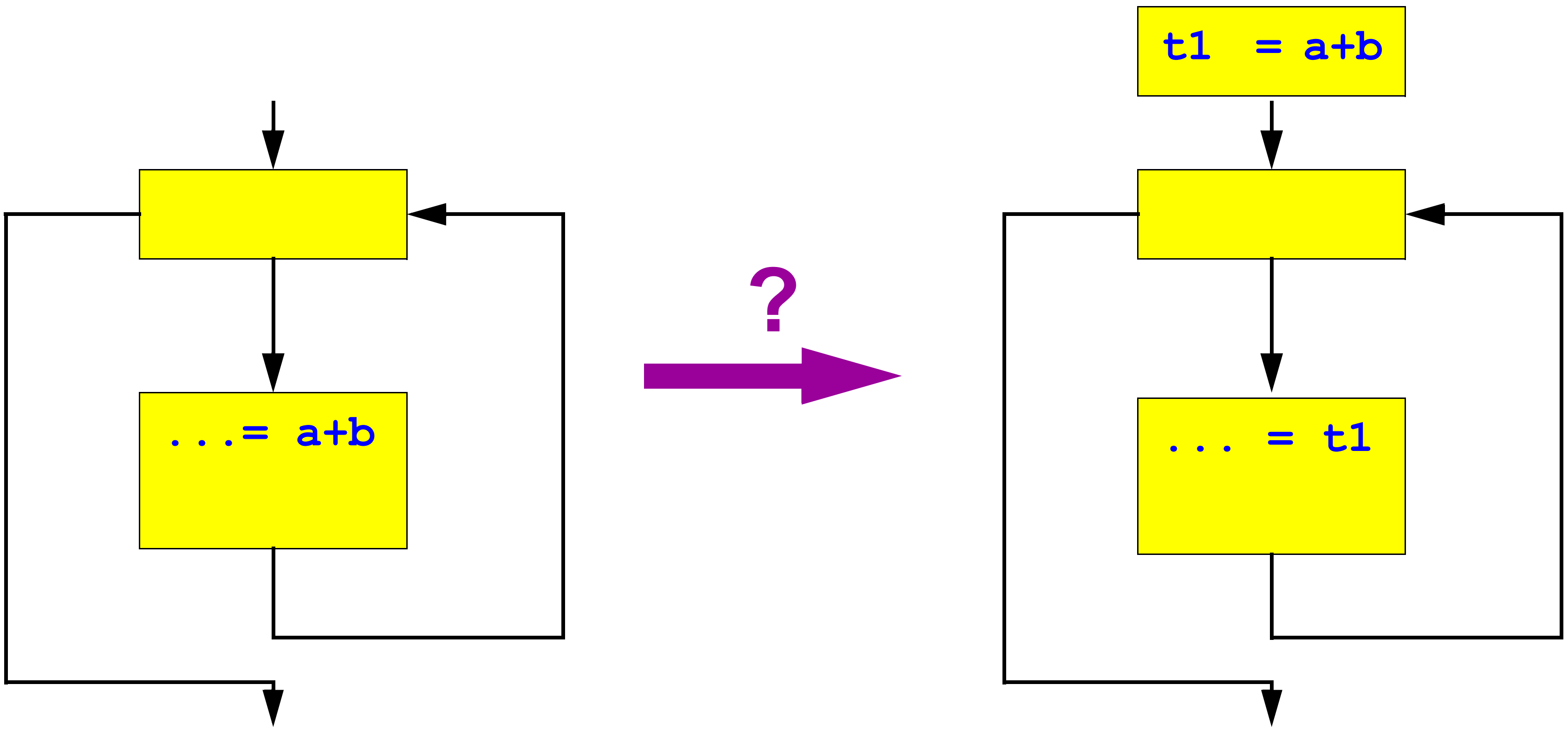
- safety, and
- optimality (no more evaluations of the expression than before).



Partial Redundancy Elimination

... is also very conservative. An expression *e* can be inserted at a point P only if every path starting from P uses *e*. This restriction guarantees:

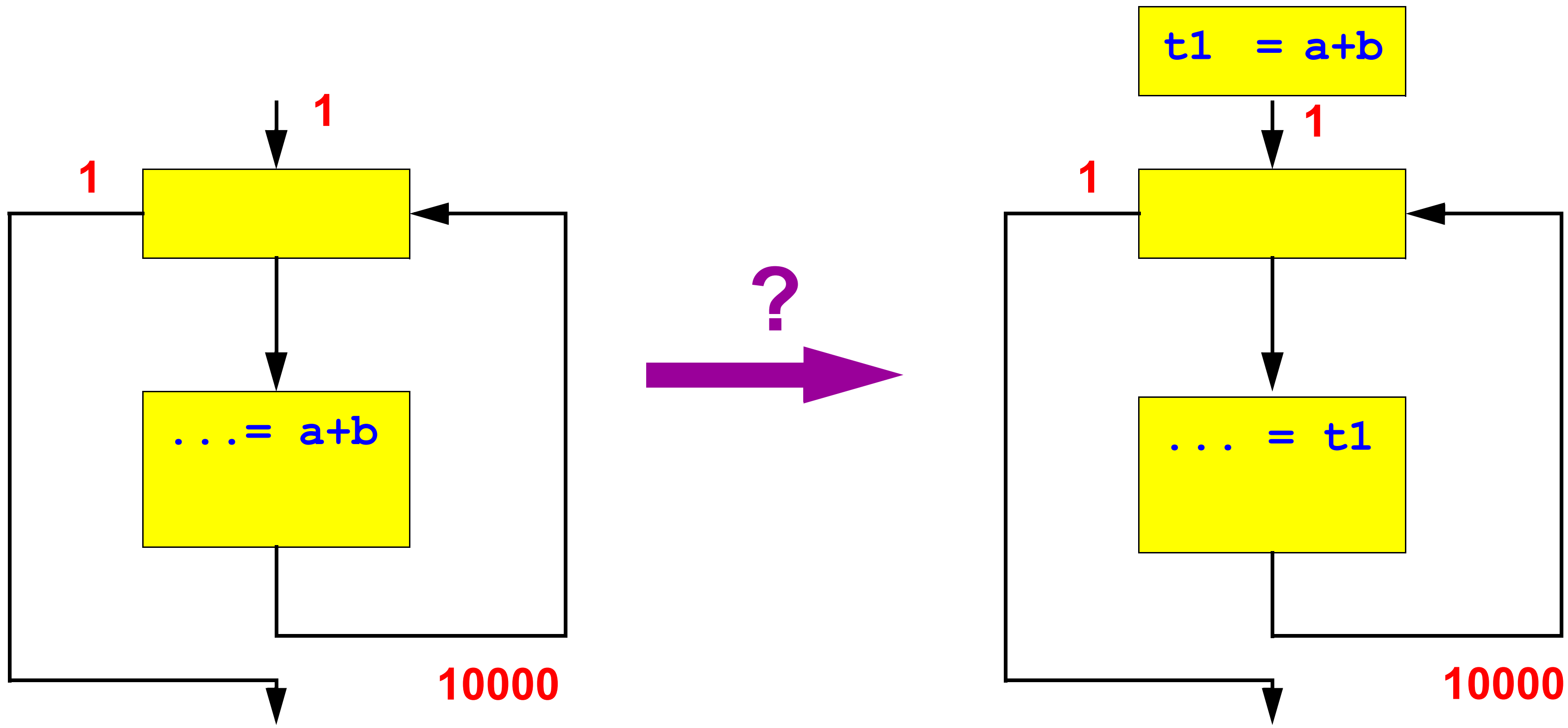
- safety, and
- optimality (no more evaluations of the expression than before).



Partial Redundancy Elimination

... is also very conservative. An expression *e* can be inserted at a point P only if every path starting from P uses *e*. This restriction guarantees:

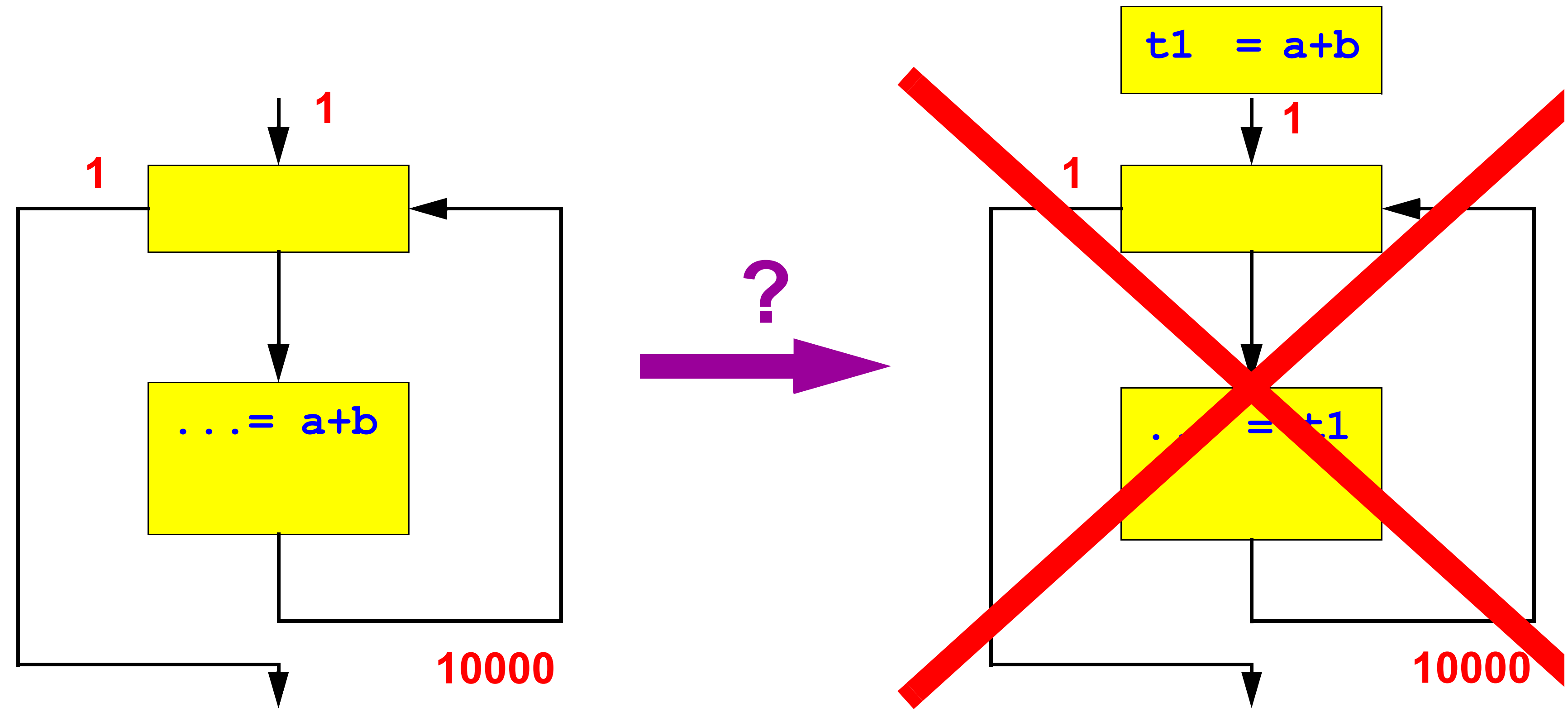
- safety, and
- optimality (no more evaluations of the expression than before).



Partial Redundancy Elimination

... is also very conservative. An expression *e* can be inserted at a point P only if every path starting from P uses *e*. This restriction guarantees:

- safety, and
- optimality (no more evaluations of the expression than before).



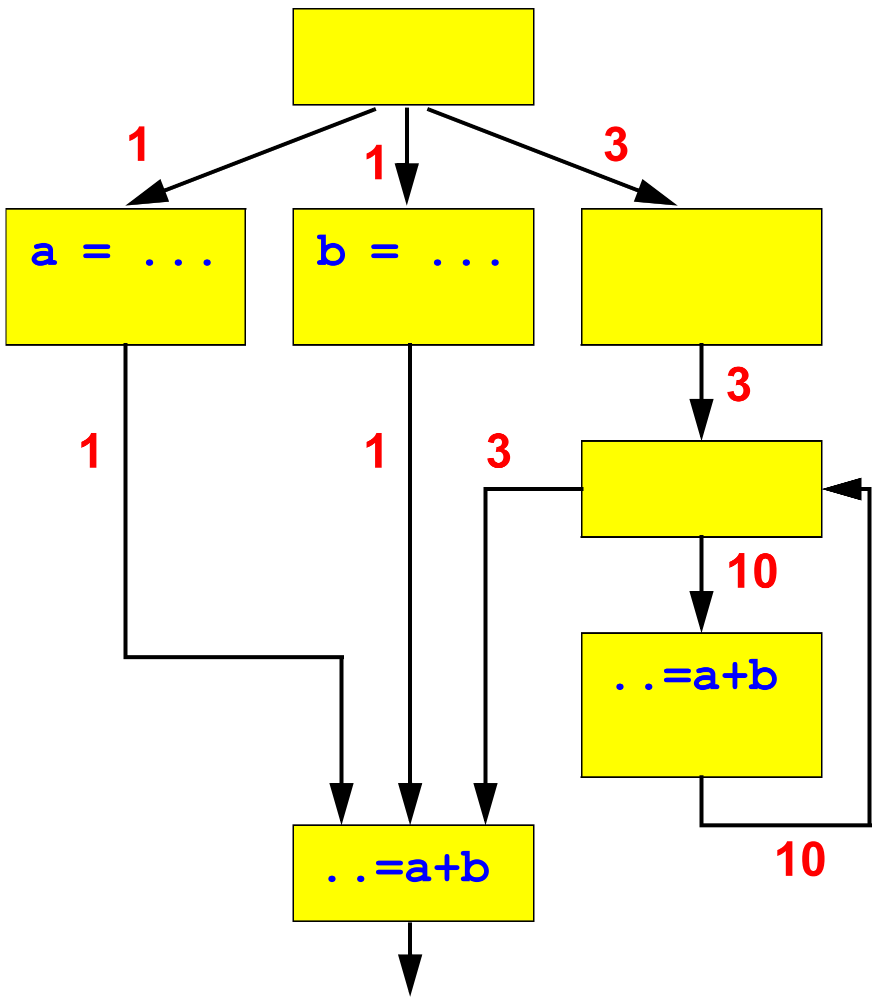
Speculative PRE

- An evaluation of *e* can be inserted anywhere as long as it is *safe* to do so, and
- we speculatively compute *e* in the hope that the value will be useful later.

Using probabilistic information (from execution profiles or elsewhere), the optimality goal becomes minimization of the *expected* number of evaluations.

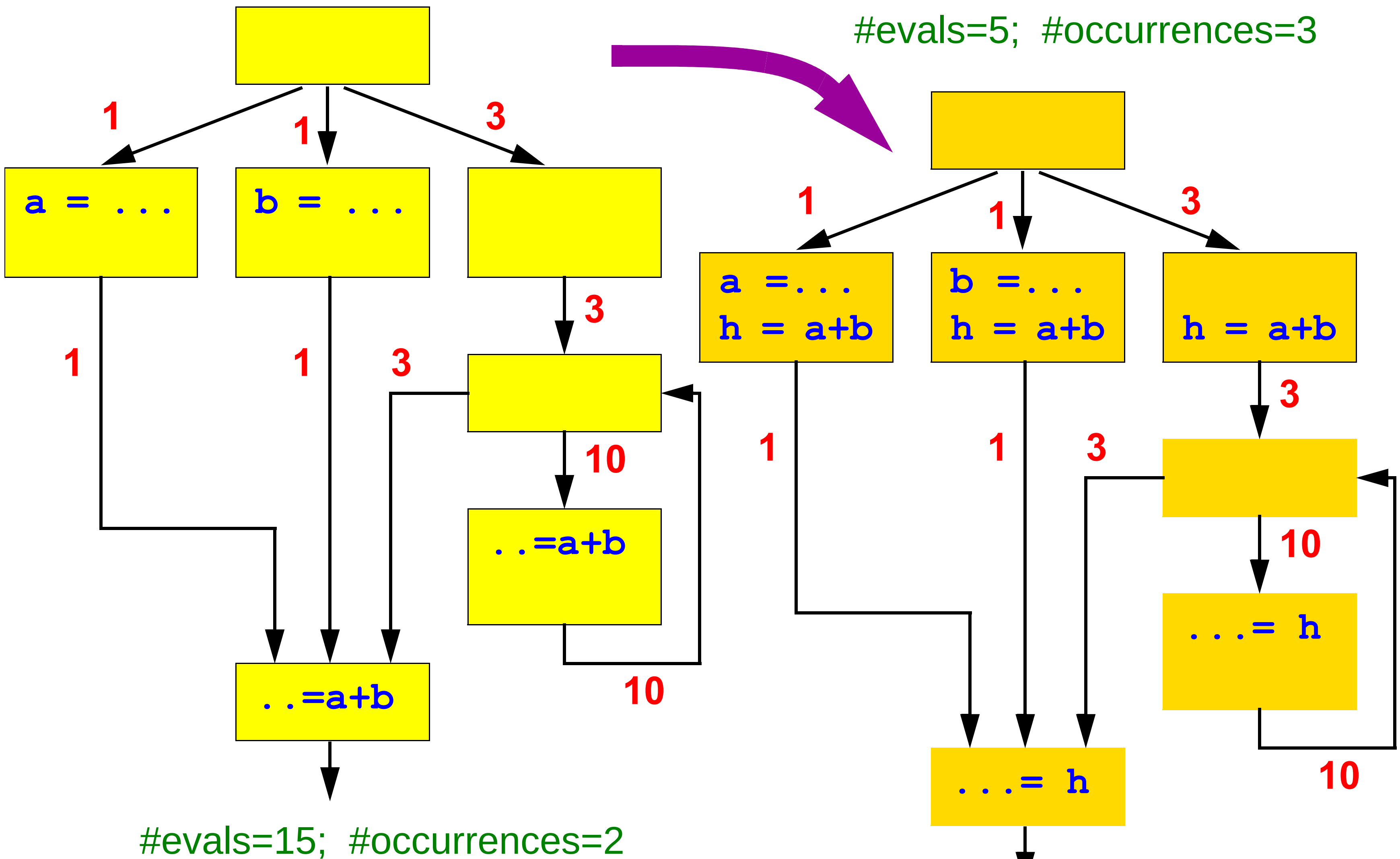
Cai and Xue presented a SPRE algorithm in 2003 which finds time-optimal solutions.

SPRE Example

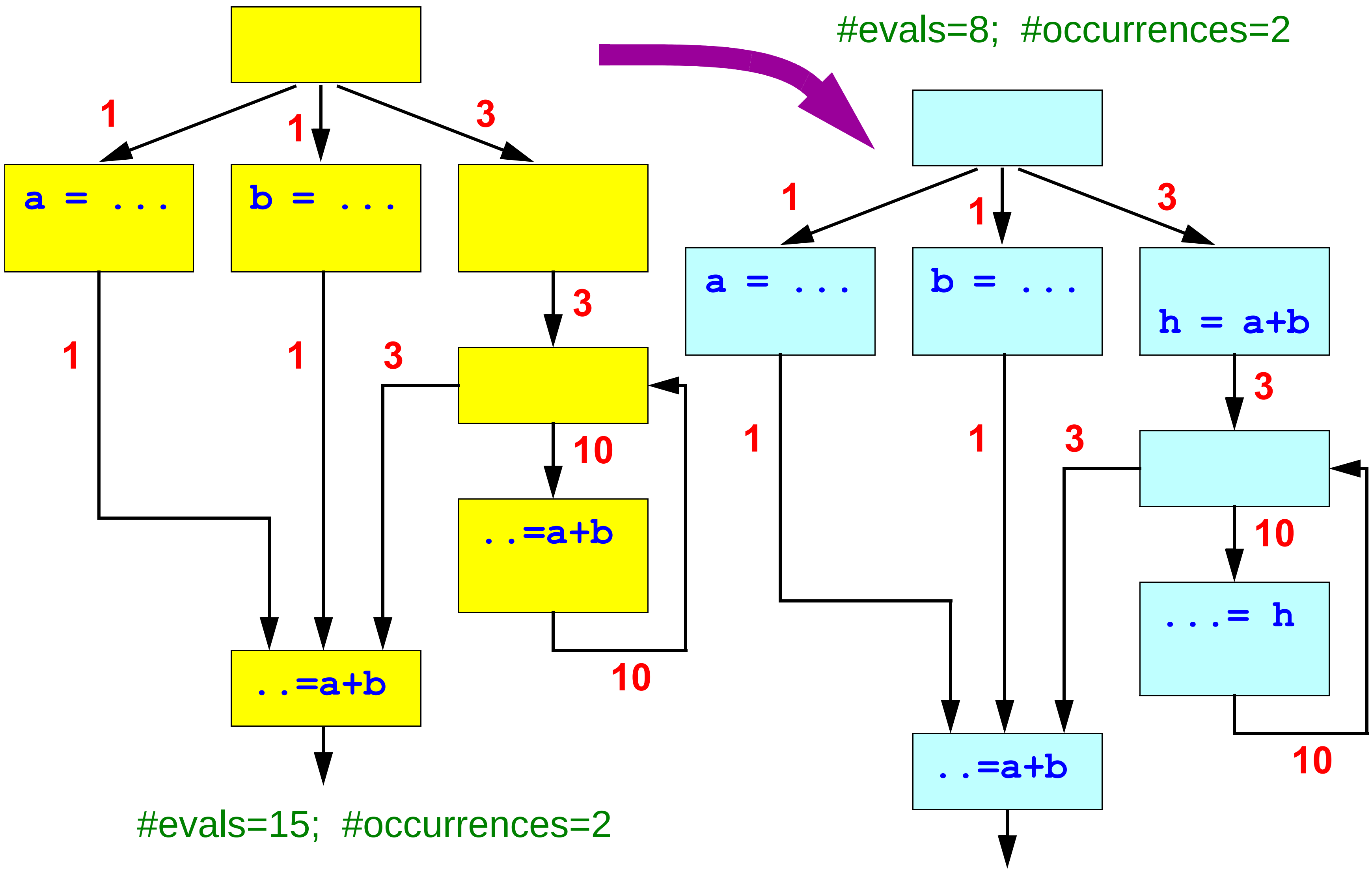


#evals=15; #occurrences=2

SPRE Example – optimized for time



SPRE Example – optimized for space



Overview of Algorithm

- The problem is decomposed into local transformations on each block in the flow graph. (For convenience only, we consider each simple statement to be a block.)
- For an expression `a+b`, we have three kinds of block:

  a NULL block which neither computes `a+b` nor assigns to `a` or `b`;
  a COMP block which computes `a+b`;
  a MOD block which assigns to `a` or `b` (and does not compute `a+b`).

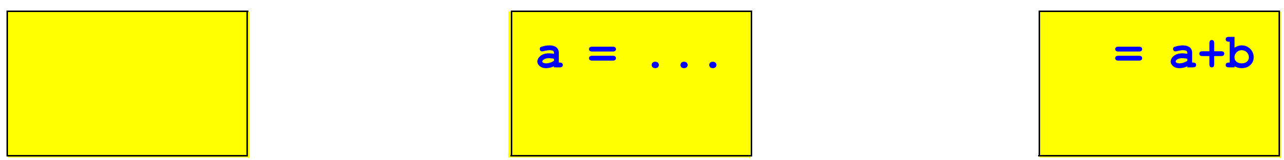
- Each local transformation incurs a cost (or a benefit); the cost is a linear combination of the code size and the expected execution frequency of the node.
- We map the costs and the constraints into a network flow problem. We use a maximum flow algorithm to find the combination of local transformations that achieves the lowest total cost (or greatest total benefit).

Local Transformations

For an expression `a+b`, the transformation of a block is driven by

- availability/unavailability of `a+b` on entry,
- whether we want `a+b` to be available on exit.

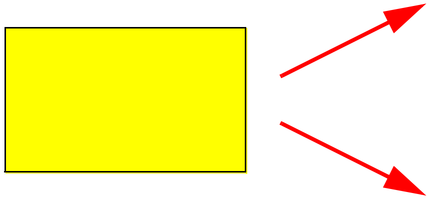
The three kinds of block are diagrammed like this:



NULL block **MOD block** **COMP block**

Local Transformations

NULL block:

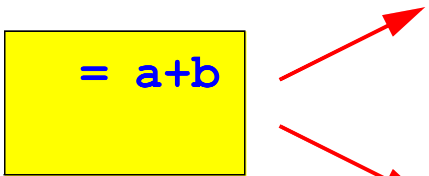


Transformations	a+b available on exit	a+b unavailable on exit
a+b available on entry	Cost = 0	Cost = 0
a+b unavailable on entry	`h = a+b` Cost = ???	Cost = 0

Static cost of `h=a+b` is 1.
Dynamic cost of `h=a+b` is the execution frequency of the node.

Local Transformations

COMP block:



Transformations	a+b available on exit	a+b unavailable on exit
a+b available on entry	`= h` Cost = 0	`= h` Cost = 0

a+b unavailable on entry	`h = a+b`
	`= h`
	Cost = ???
		`= a+b`
		Cost = ???

Local Transformations

MOD block:

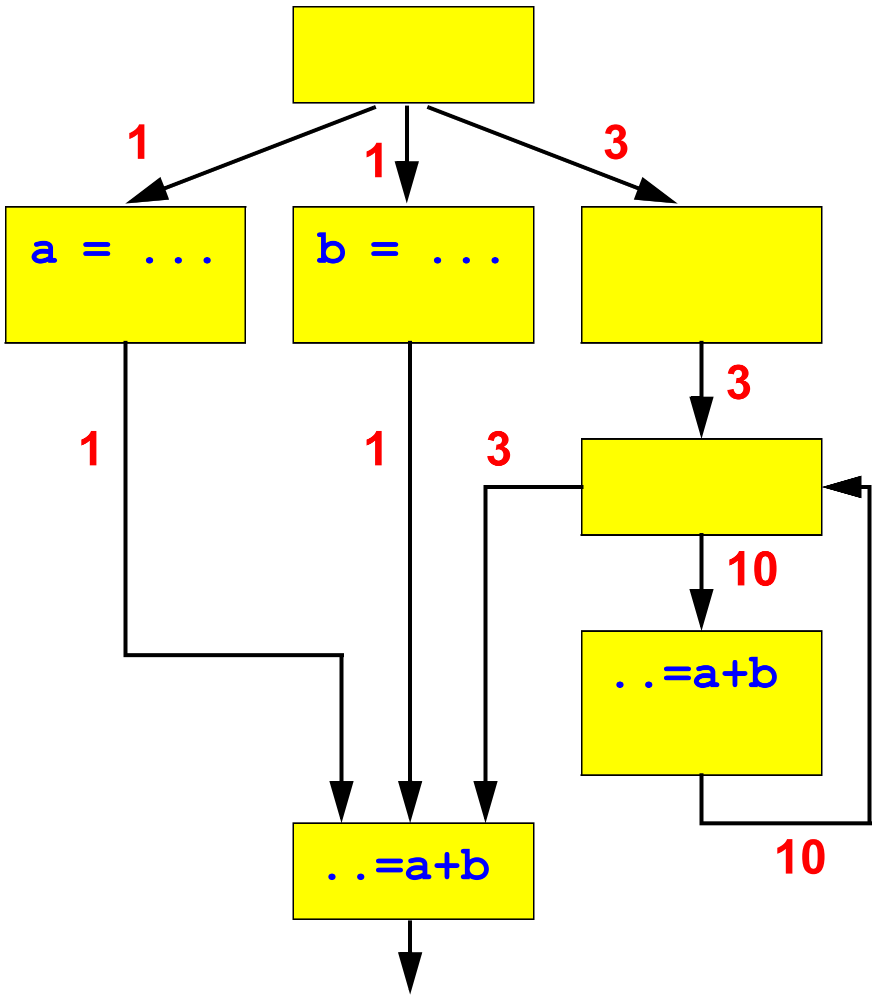
```
a =
```

Transformations	a+b available on exit	a+b unavailable on exit

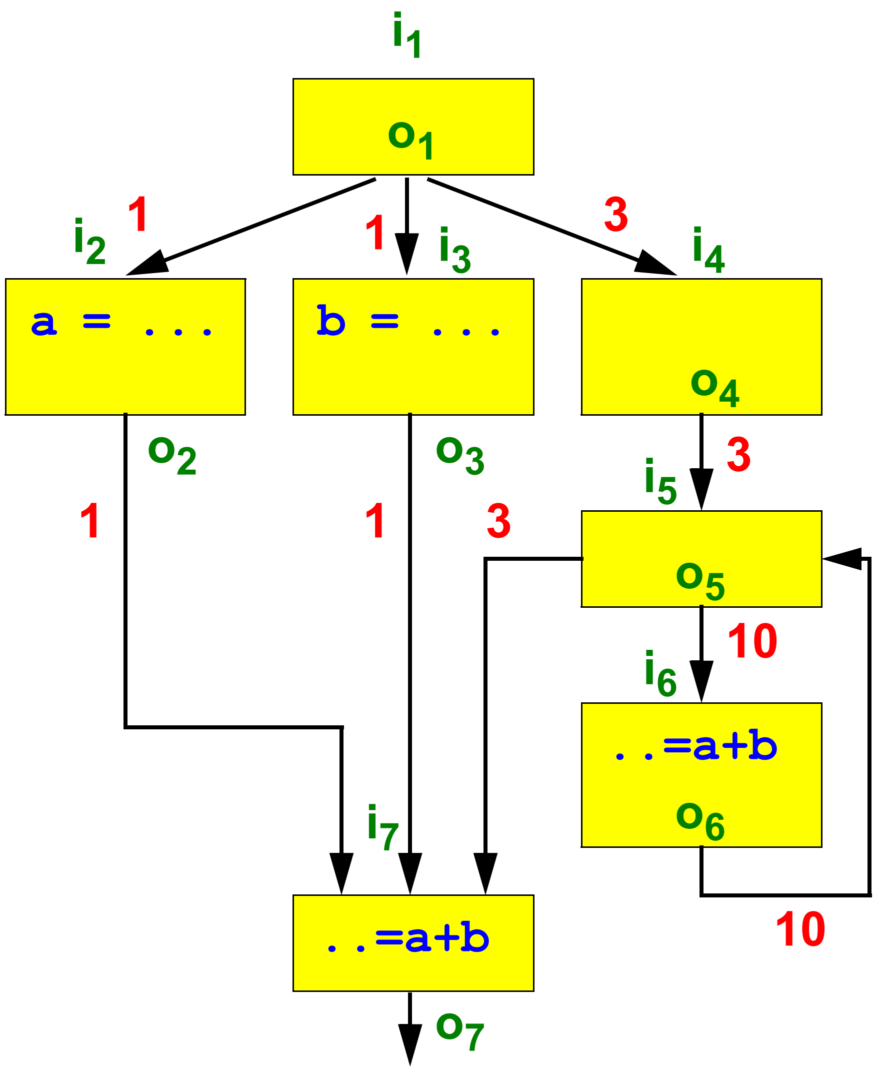
a+b available on entry	`a =`
	`h = a+b`
	Cost = ???
		`a =`
		Cost = 0

a+b unavailable on entry	`a =`
	`h = a+b`
	Cost = ???
		`a =`
		Cost = 0

Searching for an Optimal Solution ...

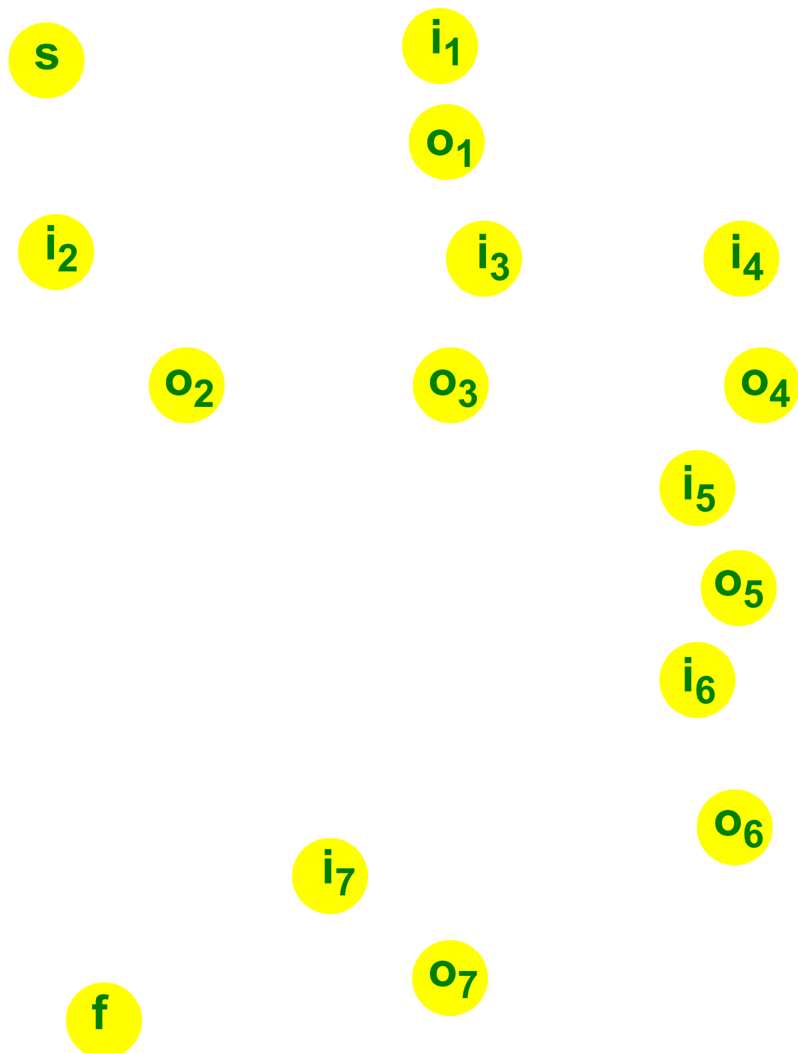


Searching for an Optimal Solution ...



- Labels i_1 ... i_7, o_1 ... o_7 denote all the places where expression `a+b` might be made available or left unavailable.
- A = set of labels where `a+b` is available in the optimal solution; ~A is the complement set.
- There are constraints on the partitioning of labels into the A and ~A sets, which we express in a flow network.

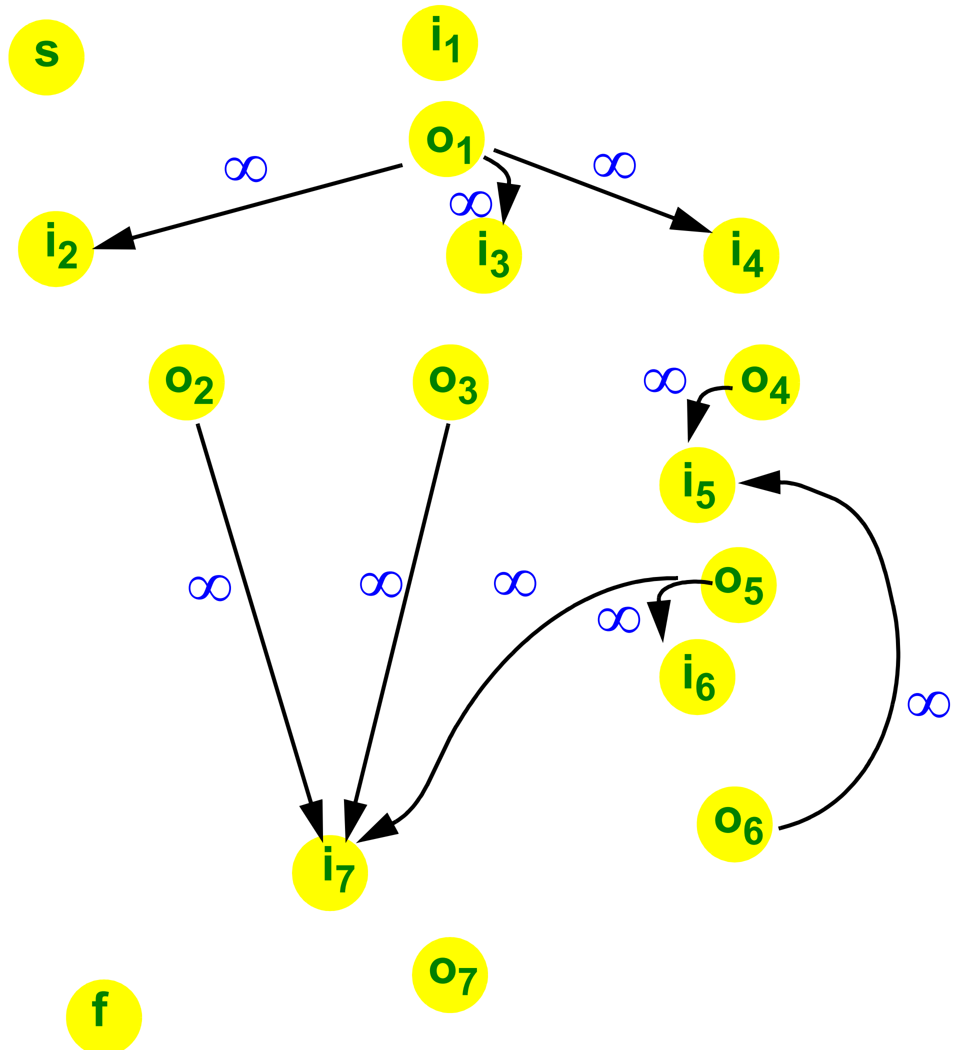
Searching for an Optimal Solution …



The labels are the nodes of the network.

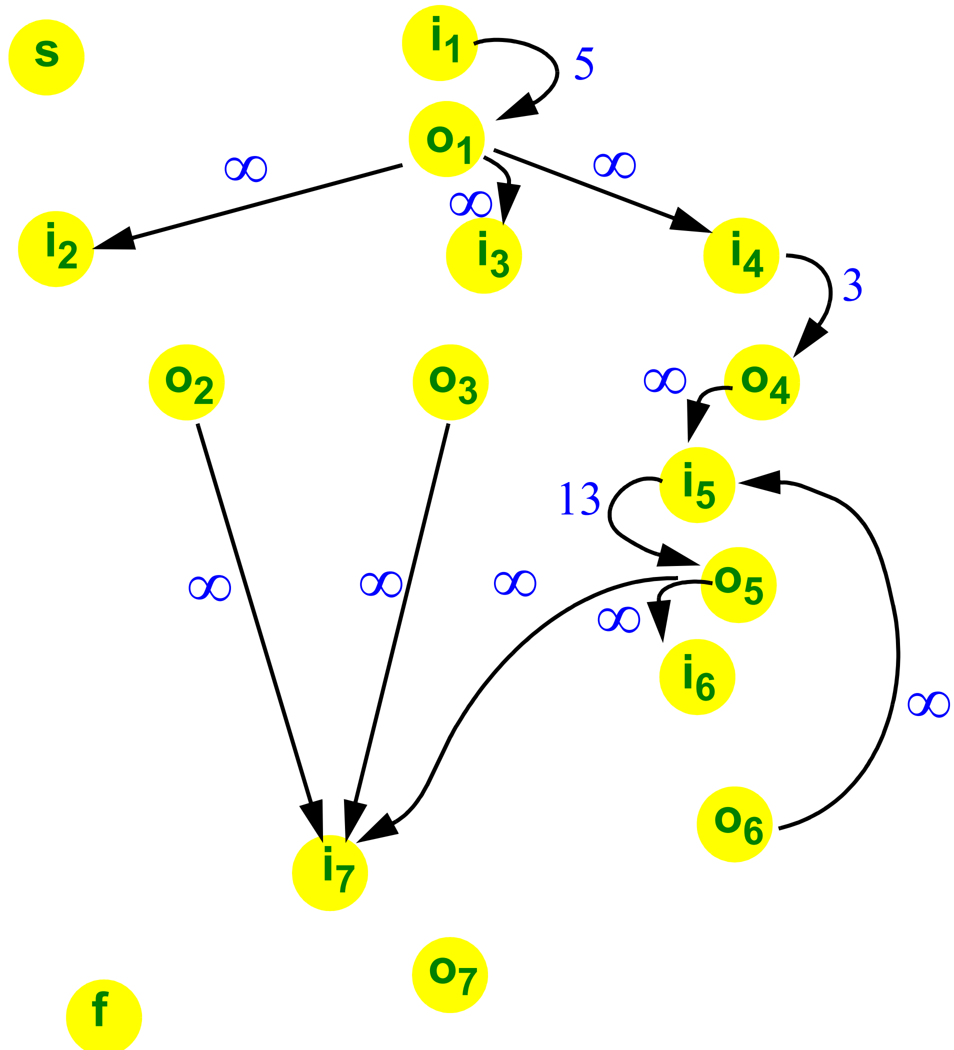
We add two more nodes *s* and *f*
(for start and finish).

Searching for an Optimal Solution …



We add edges with infinite capacity wherever two labels must have the same assignment (both in A or both in ~A), as when they are connected by an edge in the original flow graph.

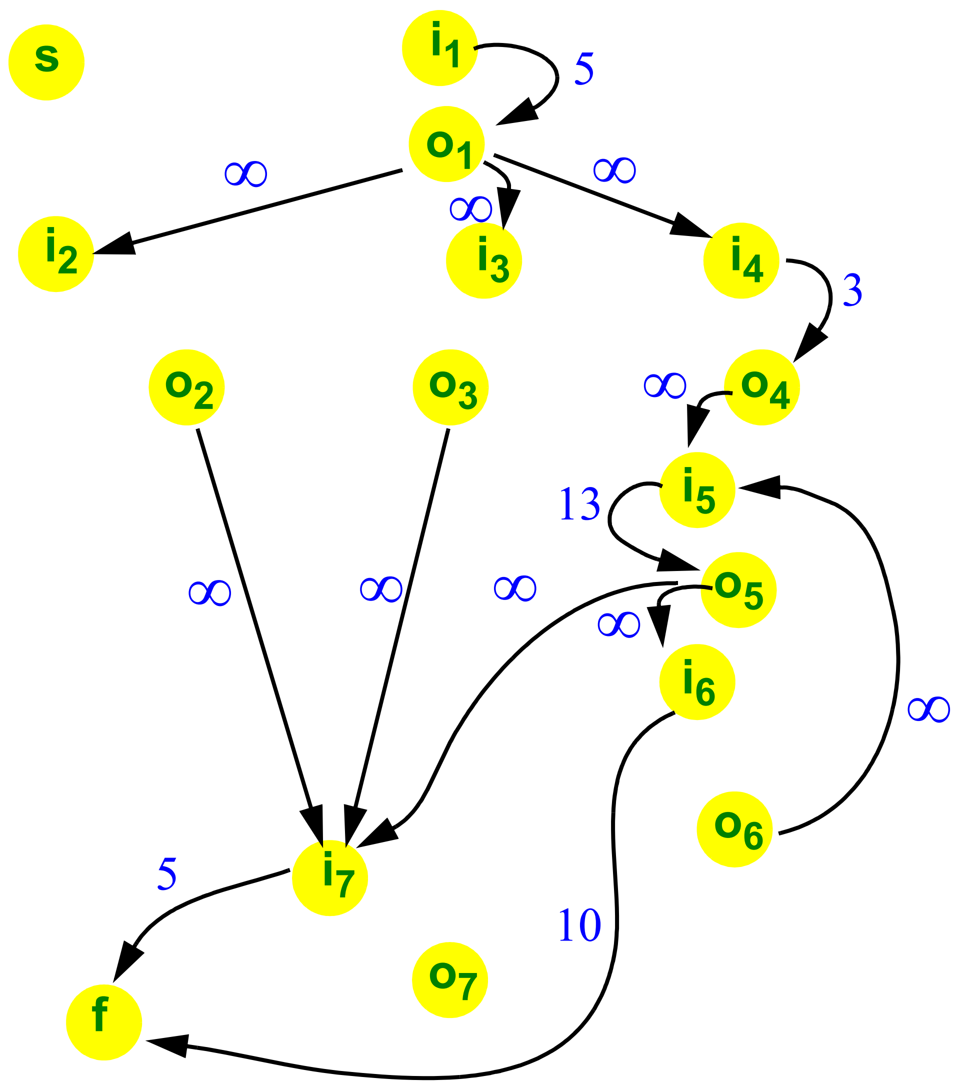
Searching for an Optimal Solution …



For each NULL block, we create an edge from its input label to its output label with capacity equal to that block's execution **frequency**.

Speed optimization

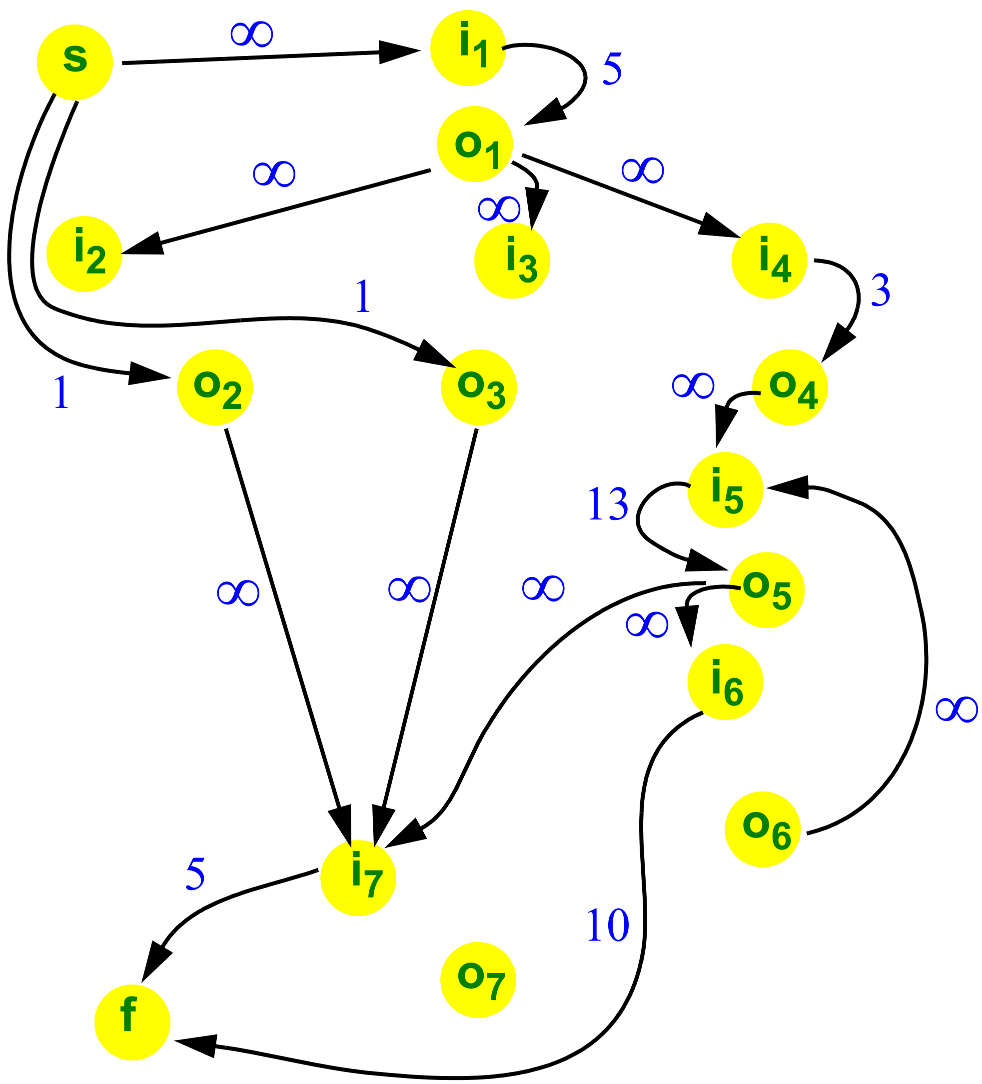
Searching for an Optimal Solution ...



For each COMP block, we add an edge from its input label to the *f* label with capacity equal to that block's execution frequency.

Speed optimization

Searching for an Optimal Solution ...

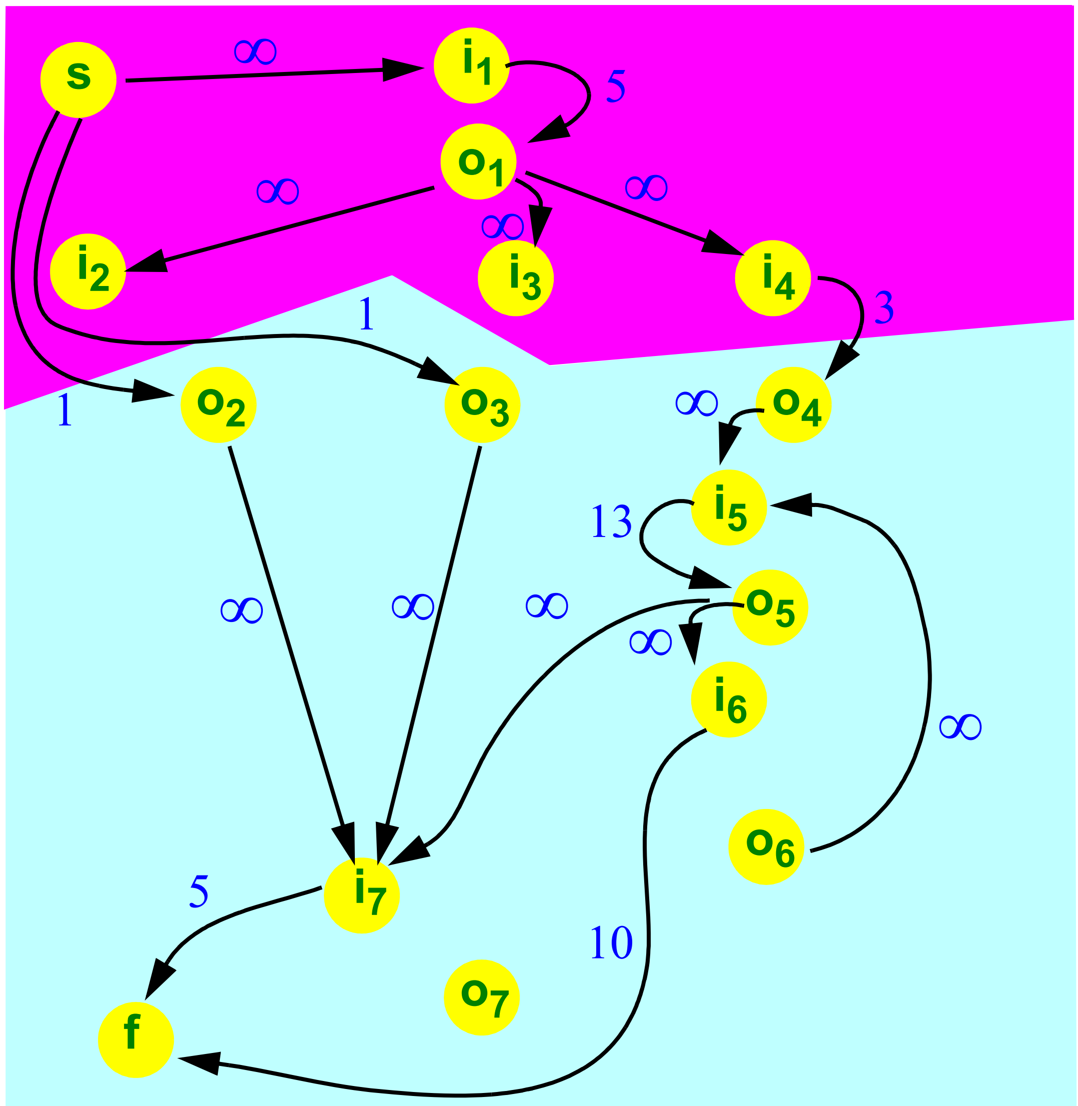


For each MOD block, we add an edge from *s* to its output label with capacity equal to that block's execution frequency.

And add an edge from *s* to the input label of the entry point with infinite capacity.

Speed optimization

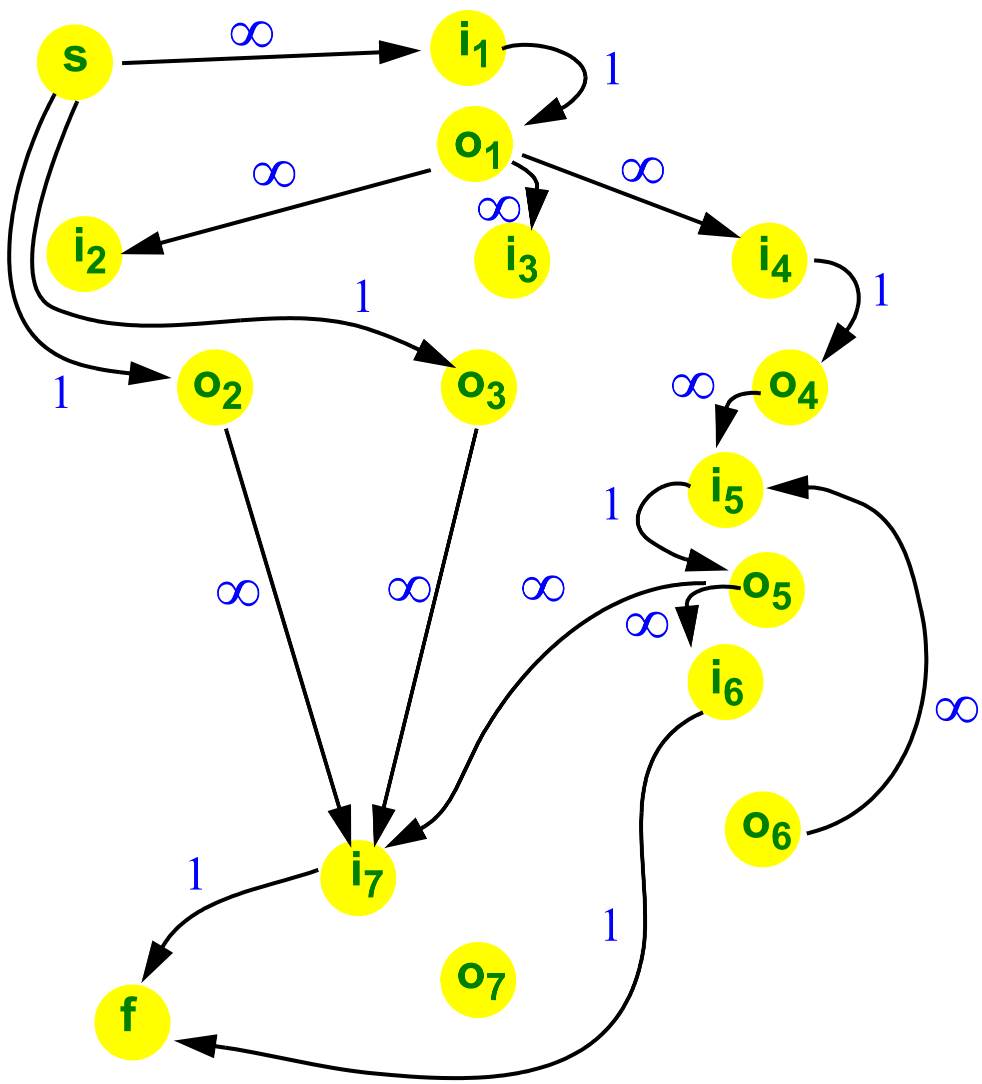
Searching for an Optimal Solution ...



The min cut – giving a maximum flow of 5

<u>**Speed optimization**</u>

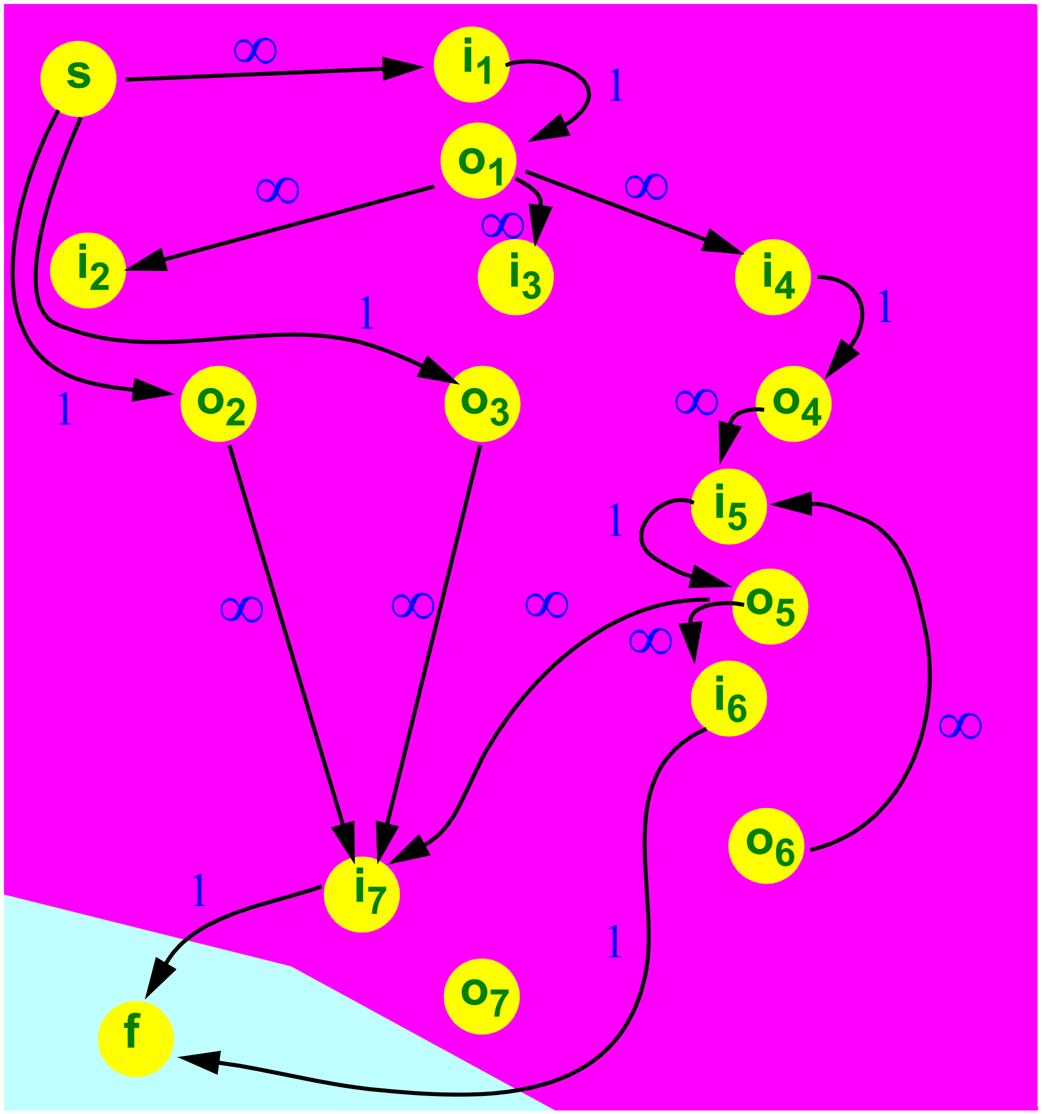
Searching for an Optimal Solution ...



If we want to optimize for space then we use 1 instead of execution frequency for the edge capacities.

Space optimization

Searching for an Optimal Solution ...



The min cut – giving a maximum flow of 2

Space optimization

Some Results

Number of (static) occurrences of expressions (space):

expressed as a ratio between original count and the count after optimizing

Benchmark	SPRE + Cost Model			PRE
	time	mix	space	
099.go	2.72	0.87	**0.85**	0.92
124.m88ksim	2.17	**0.91**	**0.91**	0.99
126.gcc	23.04	0.96	**0.92**	0.98
129.compress	2.01	**0.94**	**0.94**	0.97
130.li	2.83	0.94	**0.93**	0.97
132.ijpeg	2.35	0.97	**0.96**	0.99
134.perl	56.51	0.96	**0.90**	0.99
147.vortex	1.15	**0.91**	**0.91**	1.04

(The best ratios in each row are shown in **red**)

Some Results

Number of dynamic evaluations of expressions (time):

expressed as a ratio between original count and the count after optimizing

Benchmark	SPRE + Cost Model time	mix	space	PRE
099.go	**0.81**	**0.81**	0.88	0.84
124.m88ksim	**0.97**	**0.97**	1.00	0.98
126.gcc	**0.93**	**0.93**	1.23	0.95
129.compress	**0.90**	**0.90**	0.98	0.92
130.li	**0.96**	**0.96**	1.11	0.97
132.ijpeg	**0.98**	**0.98**	1.03	0.99
134.perl	**0.97**	**0.97**	1.53	0.98
147.vortex	**0.95**	**0.95**	1.14	0.96

Note: The mix model is time optimal in the experiments and close to being space optimal.

Conclusions & Other Points

- Optimizing for speed alone can cause significant increases in program size for little extra benefit.

- Much smaller networks can be constructed by applying some straightforward simplifications.

- An analysis can be performed for only one expression at a time, making this computationally expensive.
  However the overhead is still only ~4% of compilation time in our gcc implementation.

- PRE and SPRE significantly increase register pressure; incorporating some estimate of register pressure into the objective function would be a useful direction for further research.

Any Questions?