Data Flow Reversal II

Call Tree Reversal

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen University

Contents

Objective and Learning Outcomes

DAG Reversal

Call Tree

Call Tree Reversal
- NP Completeness
- Implementation

Summary and Next Steps

Outline

Objective and Learning Outcomes

DAG Reversal

Call Tree

Call Tree Reversal
- NP Completeness
- Implementation

Summary and Next Steps

Objective

- Introduction to Call Tree Reversal as a special case of Data Flow Reversal

Learning Outcomes

- You will understand
  - Formulation of Call Tree Reversal
  - NP-completeness of Call Tree Reversal
- You will be able to
  - Design call tree reversals
  - Evaluate costs of call tree reversals

Outline

Objective and Learning Outcomes

DAG Reversal

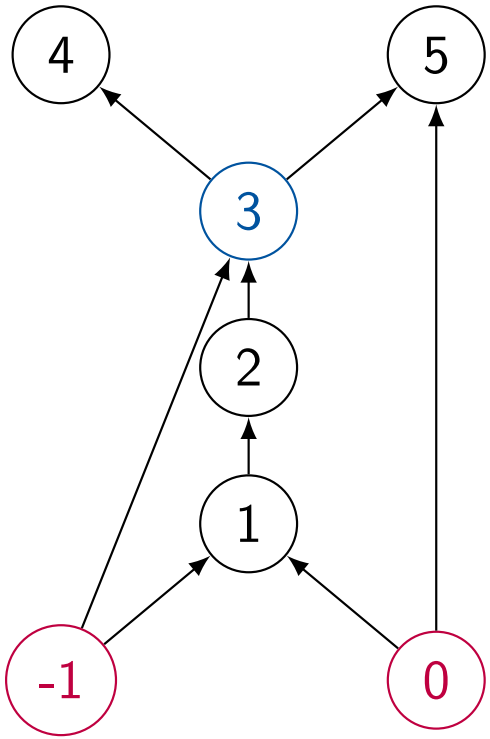
Call Tree

Call Tree Reversal
- NP Completeness
- Implementation

Summary and Next Steps

Given a DAG and two integers $C, \overline{MEM} > 0$, is there a data flow reversal that uses at most $MEM \leq \overline{MEM}$ memory units and yields a computational cost of $COST \leq C$?

The computational cost ($COST$) is defined as the sum of the number of elemental function evaluations (OPS) to be performed in addition to a single evaluation of the program (requiring $|Z \cup Y|$) and the number of write accesses to persistent memory.



DAG Reversal is NP-complete.

▶ U. Naumann: *DAG Reversal is NP-Complete*. Journal of Discrete Algorithms, Elsevier 2010.

Outline

Objective and Learning Outcomes

DAG Reversal

Call Tree

Call Tree Reversal
- NP Completeness
- Implementation

Summary and Next Steps

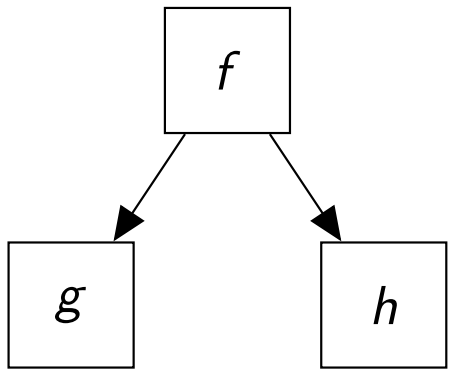
Persistently stored values are also referred to as checkpoints.

The vertex set $V = X \cup Z \cup Y$ of a DAG $G = (V, E)$ decomposes into sources X, intermediate vertices Z and sinks Y such that $X \cap Z \cap Y = \varnothing$.

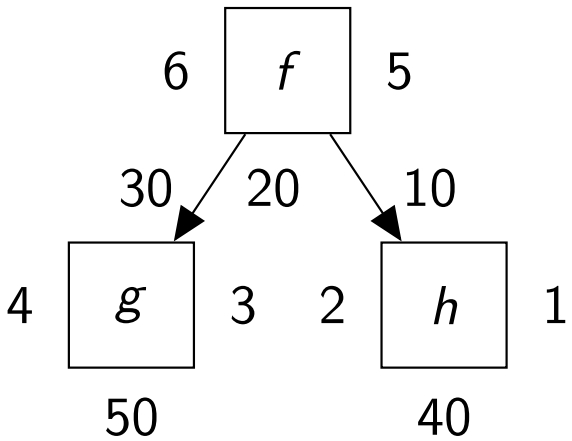
A call tree $T = (N, A)$ over a DAG G captures the interprocedural data flow of the program represented by G. Nodes in N correspond to subprograms. Subprogram calls are marked by arcs in A.

Subprograms correspond to subgraphs of G.

W.l.o.g., data flow reversal of subprograms in store-all (alternatively, recompute-all) mode is assumed to be feasible.



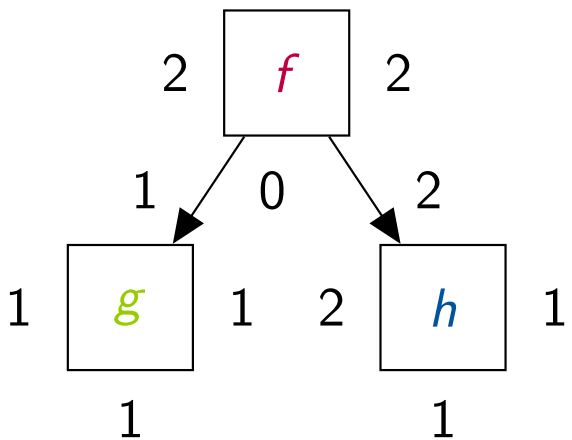
Call trees are annotated with sizes of checkpoints and of local DAGs as follows:



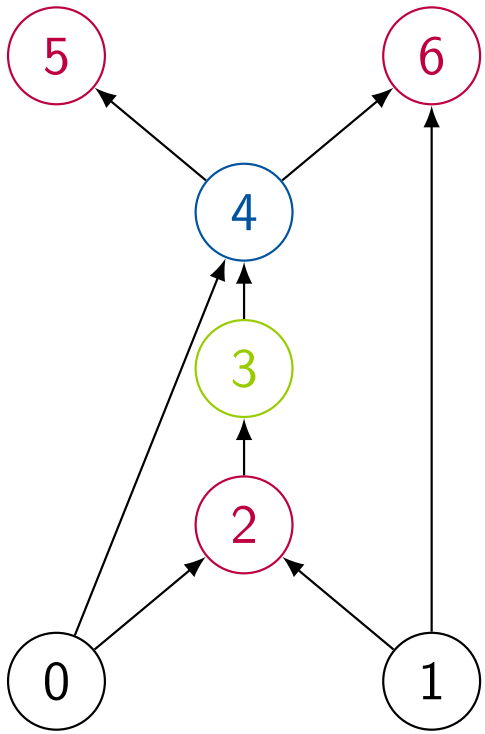
- f calls g before h
- sizes of argument checkpoints: $f : 6$, $g : 4$, $h : 2$
- sizes of result checkpoints: $f : 5$, $g : 3$, $h : 1$
- sizes of DAGs $f : 60$, $g : 50$, $h : 40$
- sizes of local DAGs in f :
  - before call of g : 30
  - in between calls of g and h : 20
  - after call of h : 10

For simplicity, we assume identical values for DAG sizes (memory occupied in local store-all mode) and numbers of elemental functions evaluations performed by (parts of) subprograms (e.g, number of vertices in DAGs), e.g, if $G = (V = X \cup Z \cup Y, E)$ is the DAG of f, then $|X| = 6$, $|Y| = 5$ and $|Z| = 60 - 5 = 55$.

Call trees correspond to partitionings / colorings of the DAG vertices. The mapping is not unique in either direction, e.g,

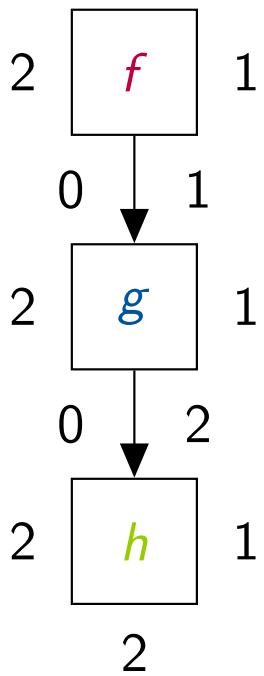



```
1  float v[7];
2
3  void f() {
4    v[2]=v[0]+v[1];
5    g();
6    h();
7    v[5]=sin(v[4]);
8    v[6]=v[4]+v[1];
9  }
10
11 void g() {
12   v[3]=sin(v[2]);
13 }
14
15 void h() {
16   v[4]=v[0]+v[3];
17 }
```

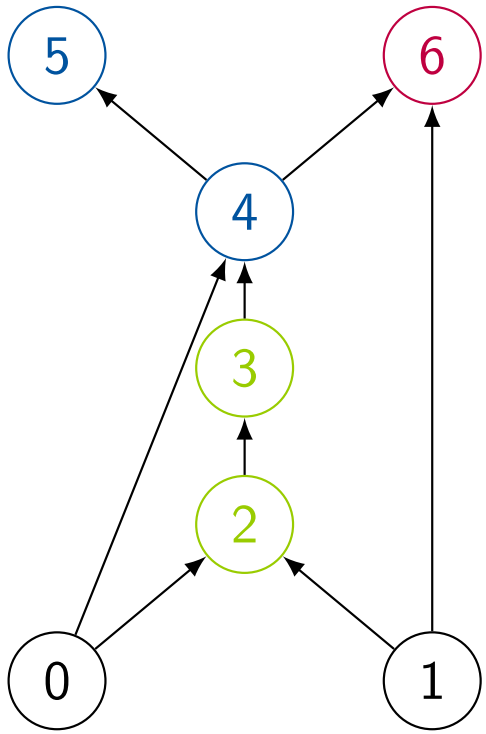


Call Tree

Example 2

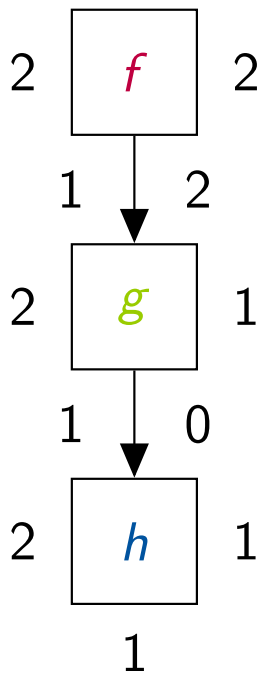



```
1   float v[7];
2
3   void f() {
4     g();
5     v[6]=v[4]+v[1];
6   }
7
8   void g() {
9     h();
10    v[4]=v[0]+v[3];
11    v[5]=sin(v[4]);
12  }
13
14  void h() {
15    v[2]=v[0]+v[1];
16    v[3]=sin(v[2]);
17  }
```

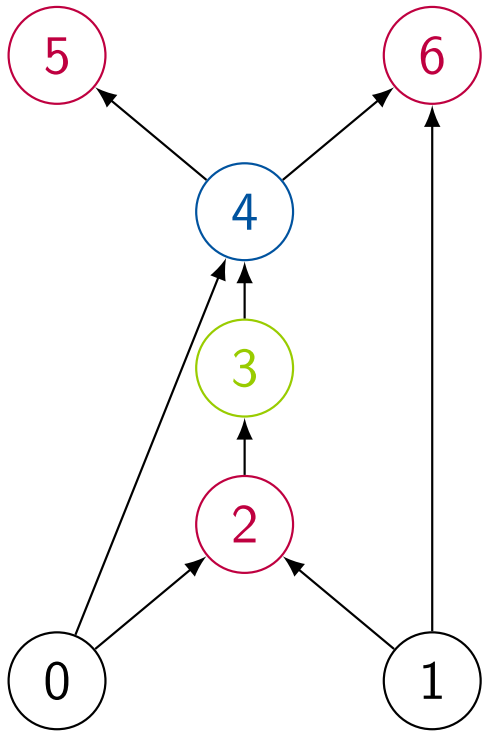


Call Tree

Example 3




```
1   float v[7];
2
3   void f() {
4     v[2]=v[0]+v[1];
5     g();
6     v[5]=sin(v[4]);
7     v[6]=v[4]+v[1];
8   }
9
10  void g() {
11    v[3]=sin(v[2]);
12    h();
13  }
14
15  void h() {
16    v[4]=v[0]+v[3];
17  }
```



Outline

Objective and Learning Outcomes

DAG Reversal

Call Tree

Call Tree Reversal
- NP Completeness
- Implementation

Summary and Next Steps

A call tree reversal is a data flow reversal with checkpoints restricted to subsets of arguments and results of subprograms.

Call Tree Reversal:

Given an annotated call tree over a DAG and two integers $C, \overline{MEM} > 0$. Is there a call tree reversal that uses at most $\overline{MEM}$ memory and costs at most C?

Call Tree Reversal is NP-complete.

- U. Naumann: *Call Tree Reversal is NP-Complete*. In Bischof et. al.: Advances in Automatic Differentiation, LNCSE 64, Springer, 2008.

run primal

run augmented primal (record local DAG)

run adjoint (reverse local DAG)

store argument checkpoint

restore argument checkpoint

store result checkpoint

restore result checkpoint

... and combinations thereof.

A result checkpointing of a call tree $T = (N, A)$ over a DAG $G = (V, E)$ is a data flow reversal which recovers all values (represented by $V = X \cup Z \cup Y$) in reverse order by storing only subsets of results of subprograms (represented by N) in addition to all values represented by X and by recomputing the other values from the stored ones.

Result Checkpointing:

Given a call tree T over a DAG $G = (V = X \cup Z \cup Y, E)$ and integers $\overline{MEM}, C > 0$. Is there a result checkpointing for T that uses at most $\overline{MEM}$ persistent memory units while costing at most $\overline{MEM} + C$)?

Result Checkpointing is NP-Complete

Proof

Reduction: DAG $G = (V = X \cup Z \cup Y, E) \rightarrow$ Call Tree $T = (N, A)$

We define a bijection between DAG Reversal and Result Checkpointing (Result Checkpointing inherits the computational complexity of DAG Reversal) as follows:

- Vertices in $Z \cup Y$ represent calls to multivariate scalar functions ϕ_i, $i = |X|, \ldots, |Z \cup Y| - 1$, operating on a global memory space $\mathbf{v} \in \boldsymbol{R}^{\mu}$, $\mu \leq |V|$ (elemental subprograms).
- $|Z_i| = 0$ for $G_i = (V_i = X_i \cup Z_i \cup Y_i, E_i)$ being the DAG of ϕ_i.
- Sequences of subprogram calls need to be reversed in recompute-all mode in order to not exceed the persistent memory bound $\overline{MEM}$.

Verification:

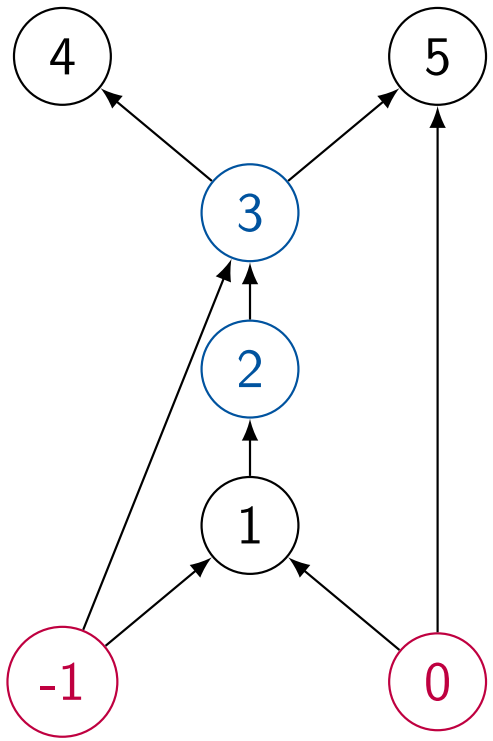
- A given solution to Result Checkpointing is trivially verified in polynomial time by counting the numbers of evaluations of elemental subprograms and of write accesses to memory.

Call Tree Reversal is NP-Complete

Proof

Example for reduction from DAG Reversal to Result Checkpointing ($\overline{MEM} = 4$):

```
1  float v[3];
2
3  void f1() { v[2]=v[0]+v[1] }
4
5  void f2() { v[2]=sin(v[2]) }
6
7  void f3() { v[2]=v[0]+v[2] }
8
9  void f4() { v[0]=sin(v[2]) }
10
11 void f5() { v[1]=v[2]+v[1] }
```



Call Tree Reversal comprises Result Checkpointing. Hence, it cannot be easier.

The following perspective on Call Tree Reversal was designed with the objective to restrict the search space such that results can be implemented in a rather straight forward fashion.

We

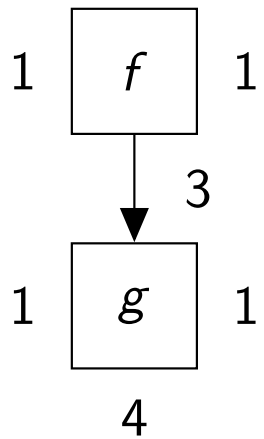
1. store all arguments of any subprogram (or none)
2. store all results of any subprogram only to avoid recomputation.

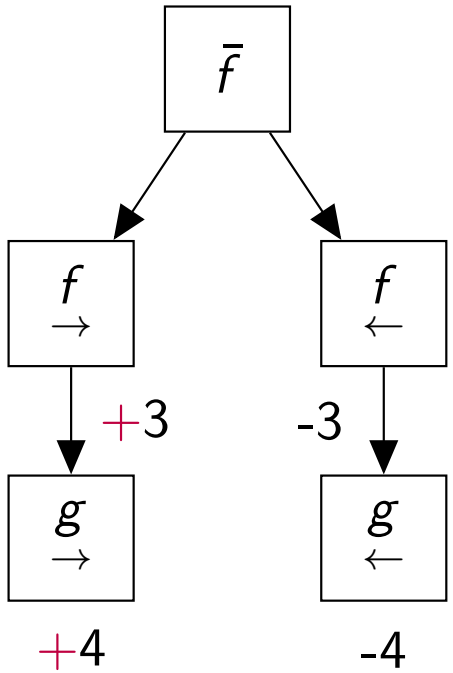
Hence, the reversal problem for a call tree $T = (N, A)$ asks for all arcs whether to store the arguments of their target nodes (1) or not (0), i.e,

$$R : E \to \{0, 1\} .$$

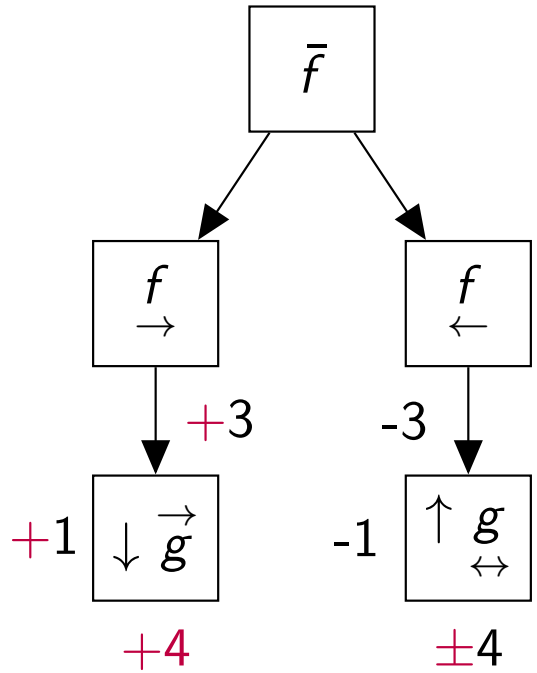
Moreover, nodes to become subject to result checkpointing need to be specified.

Early Recording costs (7,7)





Late Recording costs (4,12)

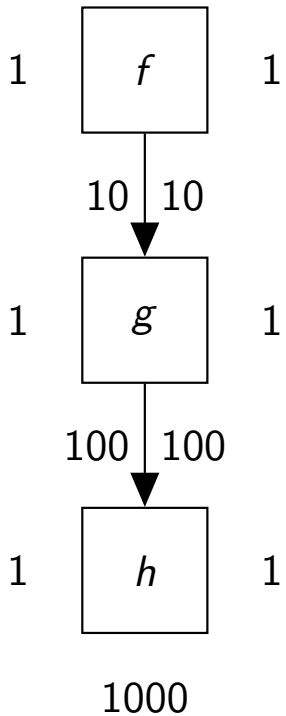


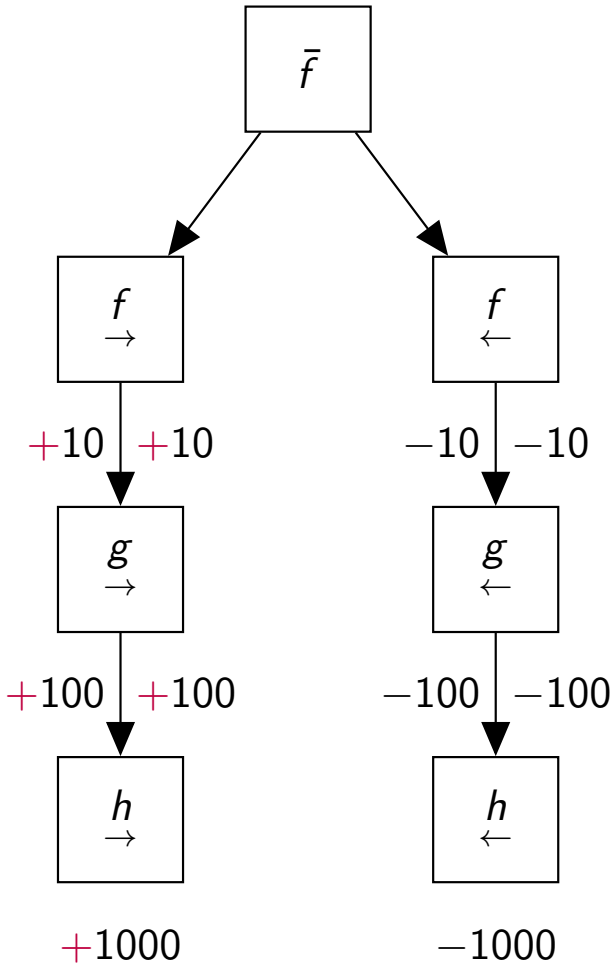
Color coding accesses to memory (black) and contributions to *COST*; mixed colors mark contributions to both, e.g, +4 marks increase in memory requirement by four units (write access). Missing labels are equal to zero.

Note: $COST = OPS+$ write accesses to memory $-|V|$.

Call Tree Reversal

Global Early Recording

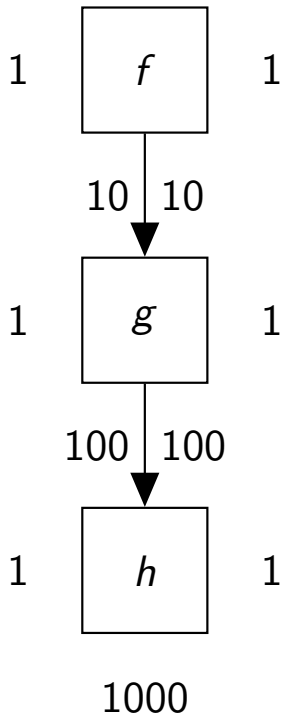


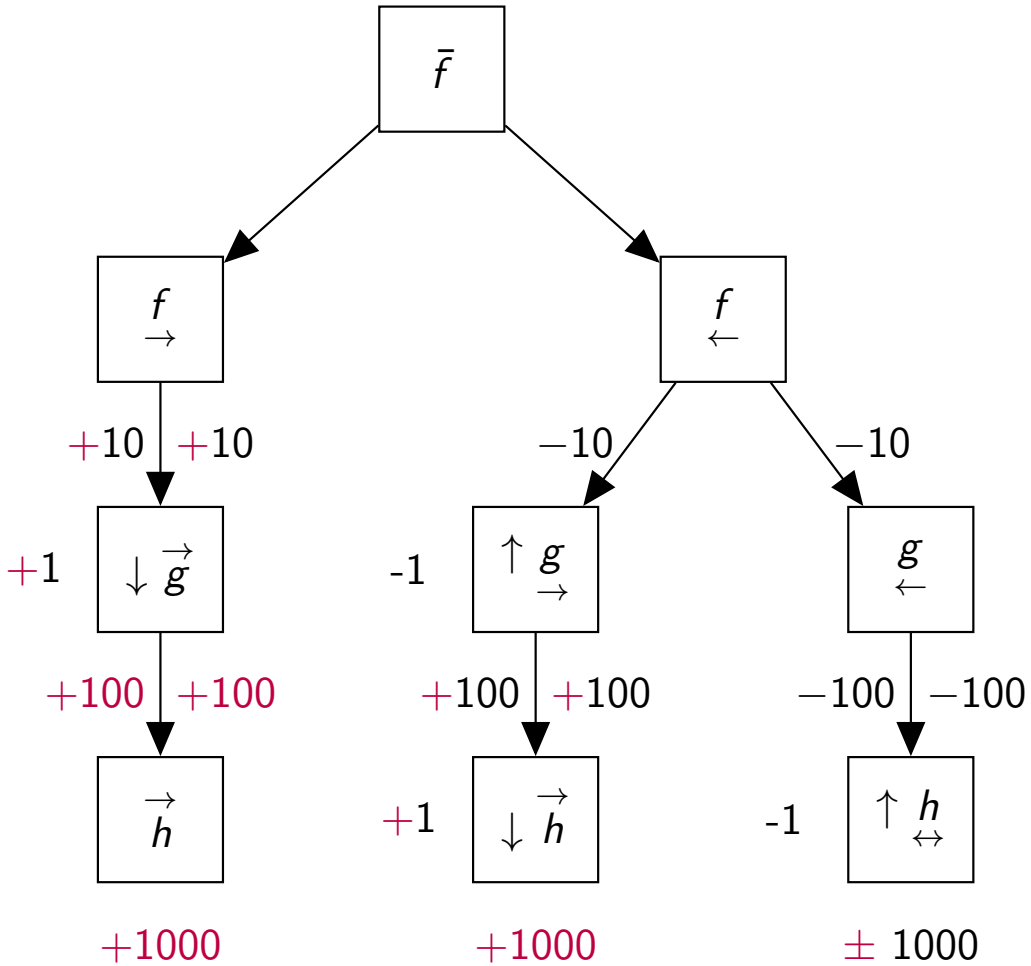


$$R = \{((f, g), 0), ((g, h), 0)\} \text{ costs } (1220, 1220)$$

Call Tree Reversal

Global Late Recording

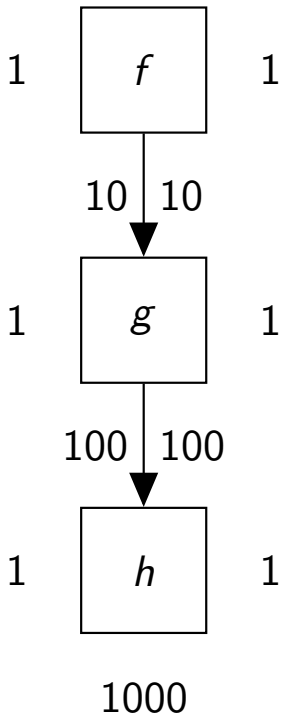


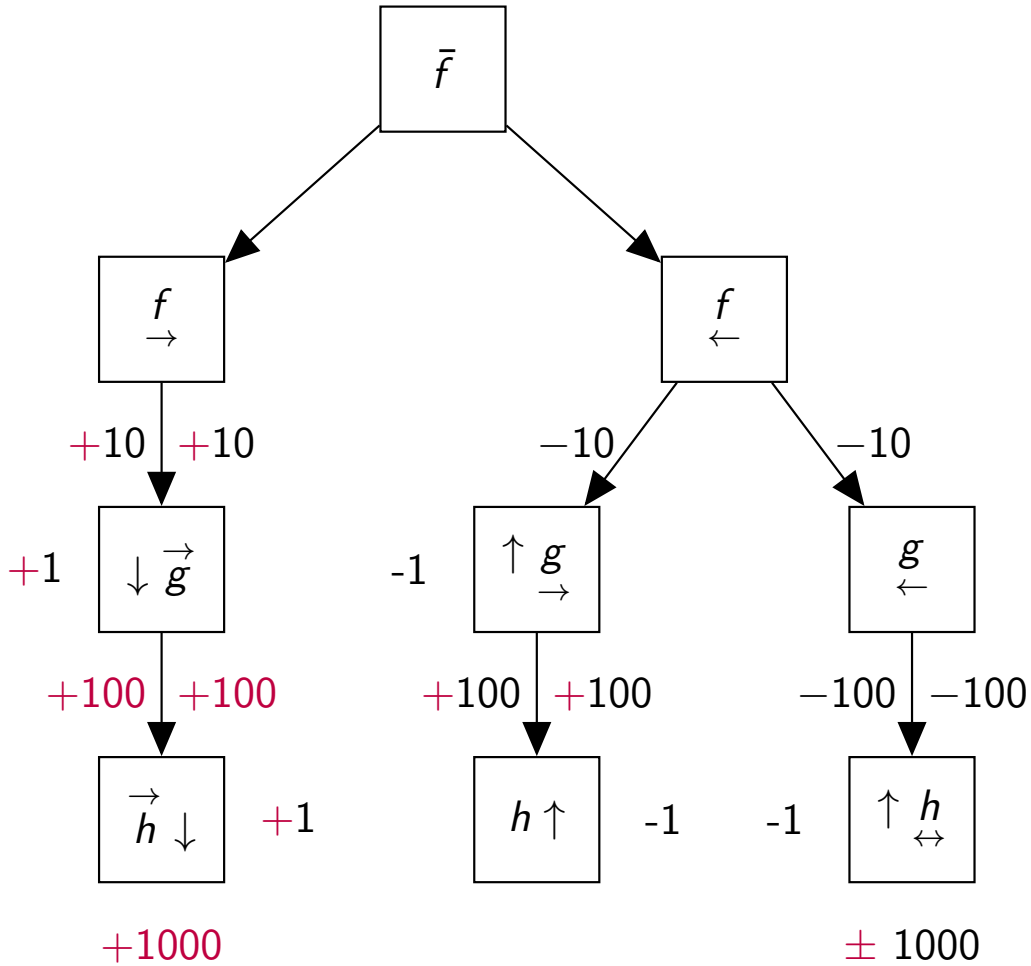


$$R = \{((f,g),1),((g,h),1)\} \text{ costs } (1110,3422)$$

Call Tree Reversal

Result Checkpointing



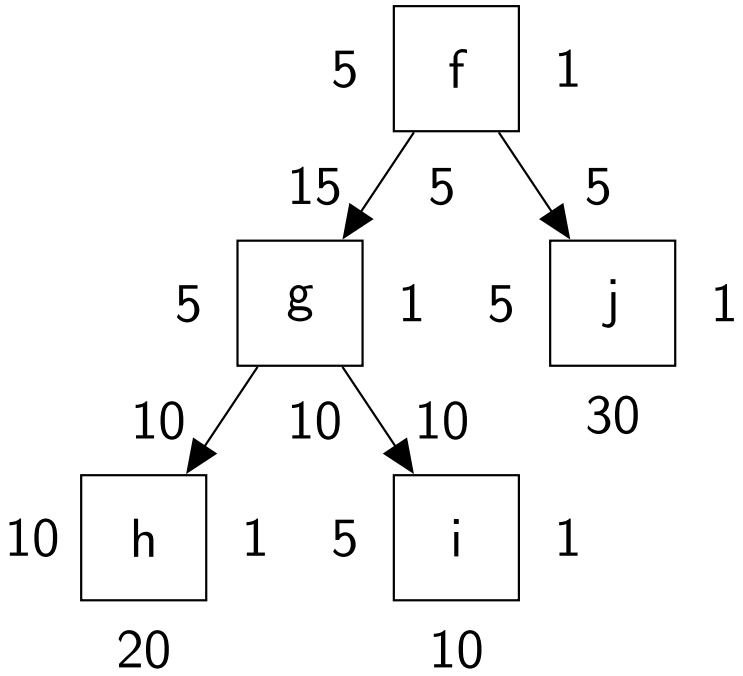


$R = \{((f,g),1),((g,h),1)\} \cup \{h\}$ costs (1110,2422)

Starting from $R = \{\forall a \in A : \ 0\}$ switch call reversal modes from 0 to 1 in increasing / decreasing orders of local DAG sizes.

Starting from $R = \{\forall a \in A : \ 1\}$ switch call reversal modes from 1 to 0 in increasing / decreasing orders of local DAG sizes.

Exercises: Consider the remaining two options for the above example and ...



Outline

Objective and Learning Outcomes

DAG Reversal

Call Tree

Call Tree Reversal
- NP Completeness
- Implementation

Summary and Next Steps

Summary

- Introduction to Call Tree Reversal as a special case of Data Flow Reversal
- NP-completeness of Call Tree Reversal
- Design call tree reversals and their costs

Next Steps

- Practice design of call tree reversals and evaluation of their costs.
- Continue the course to find out more ...