# Model-Driven Design of Performance Requirements

A. García-Domínguez, I. Medina-Bulo and M. Marcos-Bárcena

University of Cádiz

QSIC '11
July 13th, 2011

# Motivation

## Non-functional requirements (NFR)

- Good software meets various NFR in addition to regular FR
- Examples of NFRs: performance, throughput, etc.

## Performance in workflows

- Workflows compose internal and external tasks into new task
- Clients impose Service Level Agreements (SLAs) on workflows
- We need to answer several questions on a workflow:
  - Can we meet the requirements set by the user?
  - How fast would each task need to be, in that case?
- It is cheaper to do this before implementing the workflow
- We could derive tests from these answers

# Information sources for performance data in the model

## In order of preference

1. Historical data of observed performance
2. Signed SLAs for external tasks
3. Expert knowledge about the task and the systems

## Challenges

- We might want to relax historical observations or SLAs "a bit". Formally, what is "a bit"?
- Expert knowledge may be partial, e.g. "this takes at least $x$ seconds" or "this takes roughly twice as long as task X"
- How do we combine all this information?

# Our approach

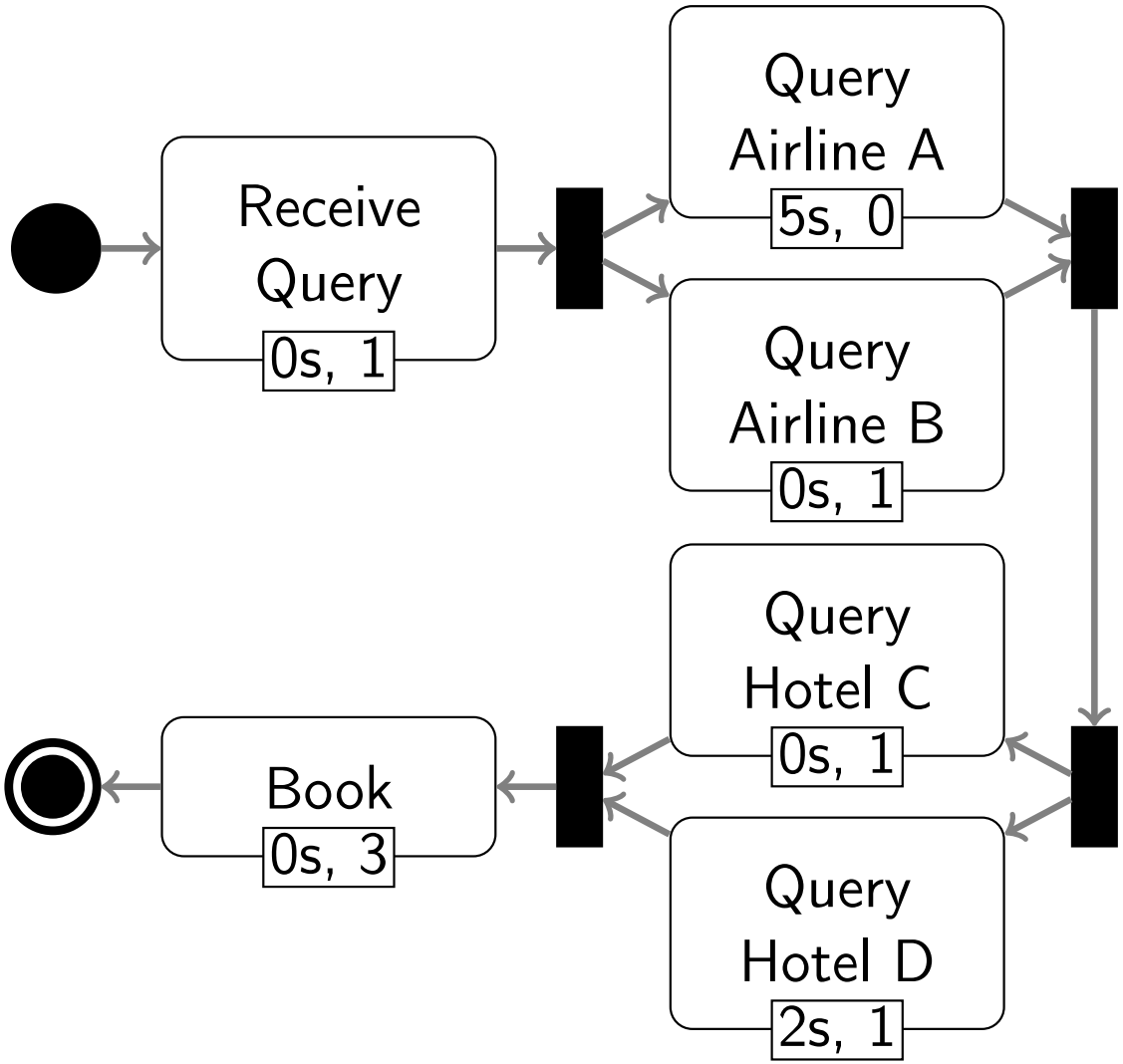
## Algorithms: exhaustive and incremental

- They answer the previous questions on a workflow model, by:
  - Inferring time limits for every task
  - Notifying the user when this is unfeasible
- Tests show that both obtain the same results
- Performance widely differs

## Workflow models

- All paths must finish within a global time limit
- The inferred time limit for each task is the sum of:
  - Fixed part: minimum time limit
  - Variable part: proportional to its weight

# Example: simplified travel search



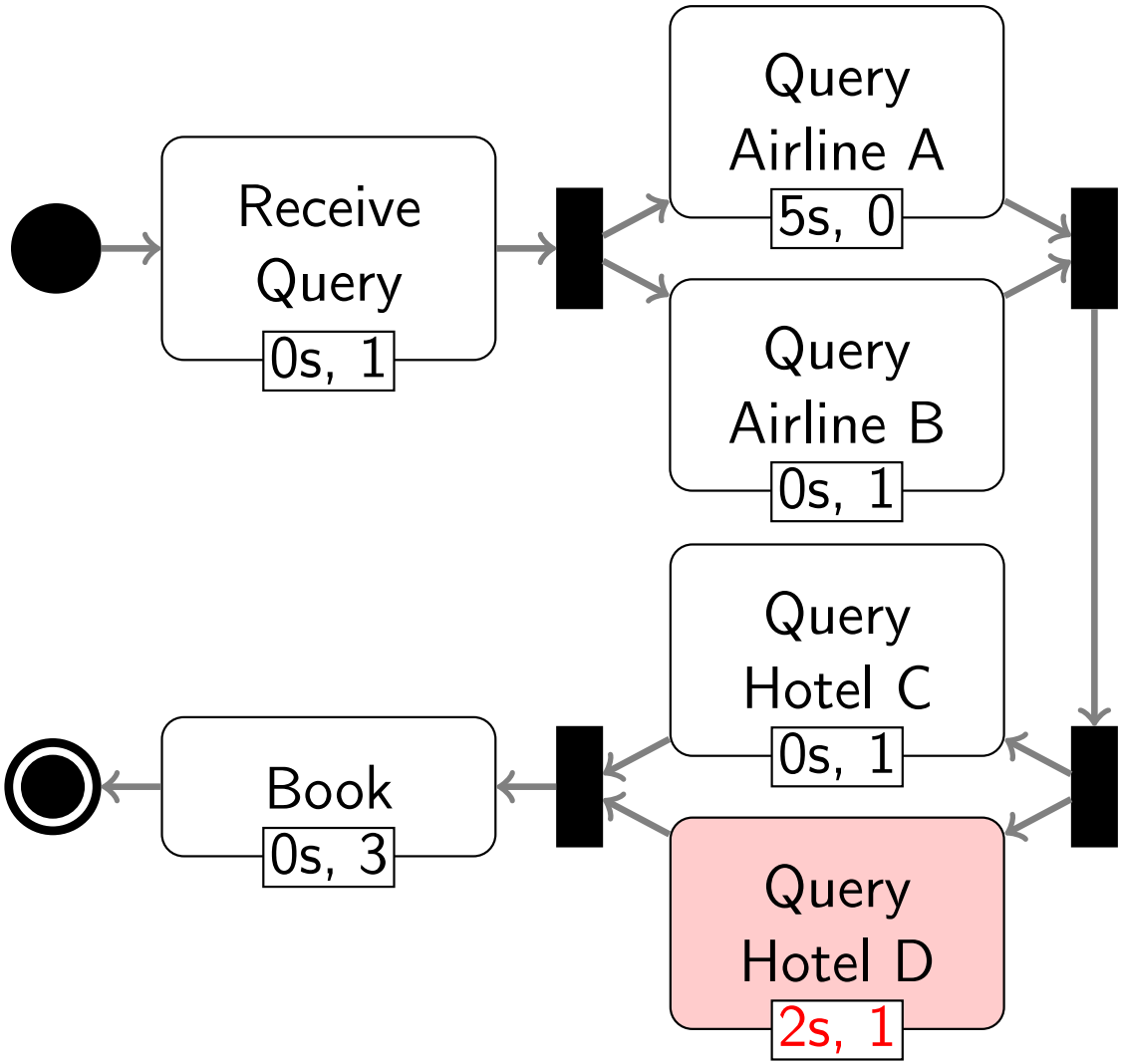

## Requirements

- DAG
- 1 source
- 1+ sinks
- $(m, w)$ pairs for each node

## Info sources

- Historical data
- Signed SLAs
- Experience

# Example: simplified travel search



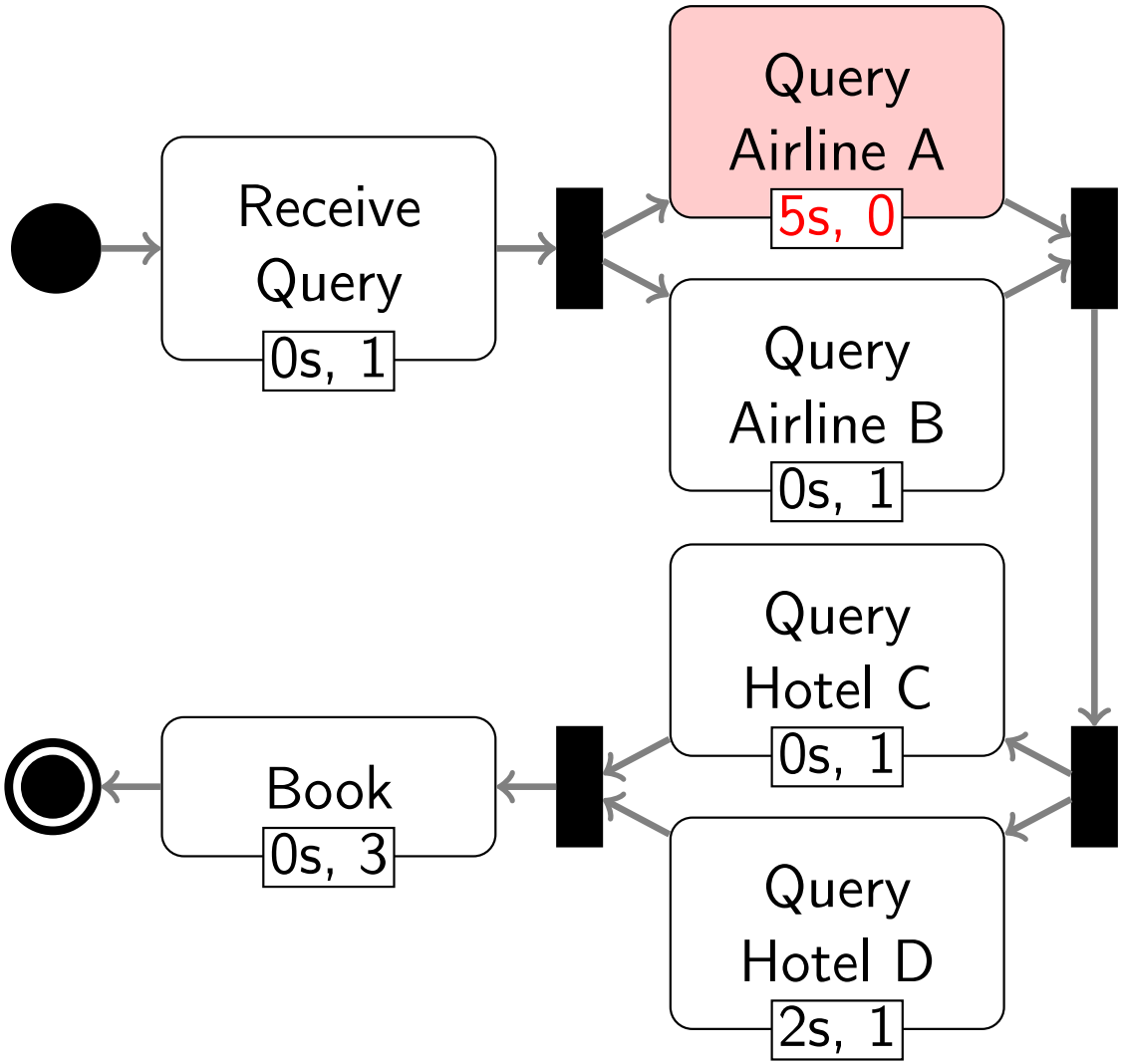

## Requirements

- DAG
- 1 source
- 1+ sinks
- $(m, w)$ pairs for each node

## Info sources

- Historical data
- Signed SLAs
- Experience

# Example: simplified travel search



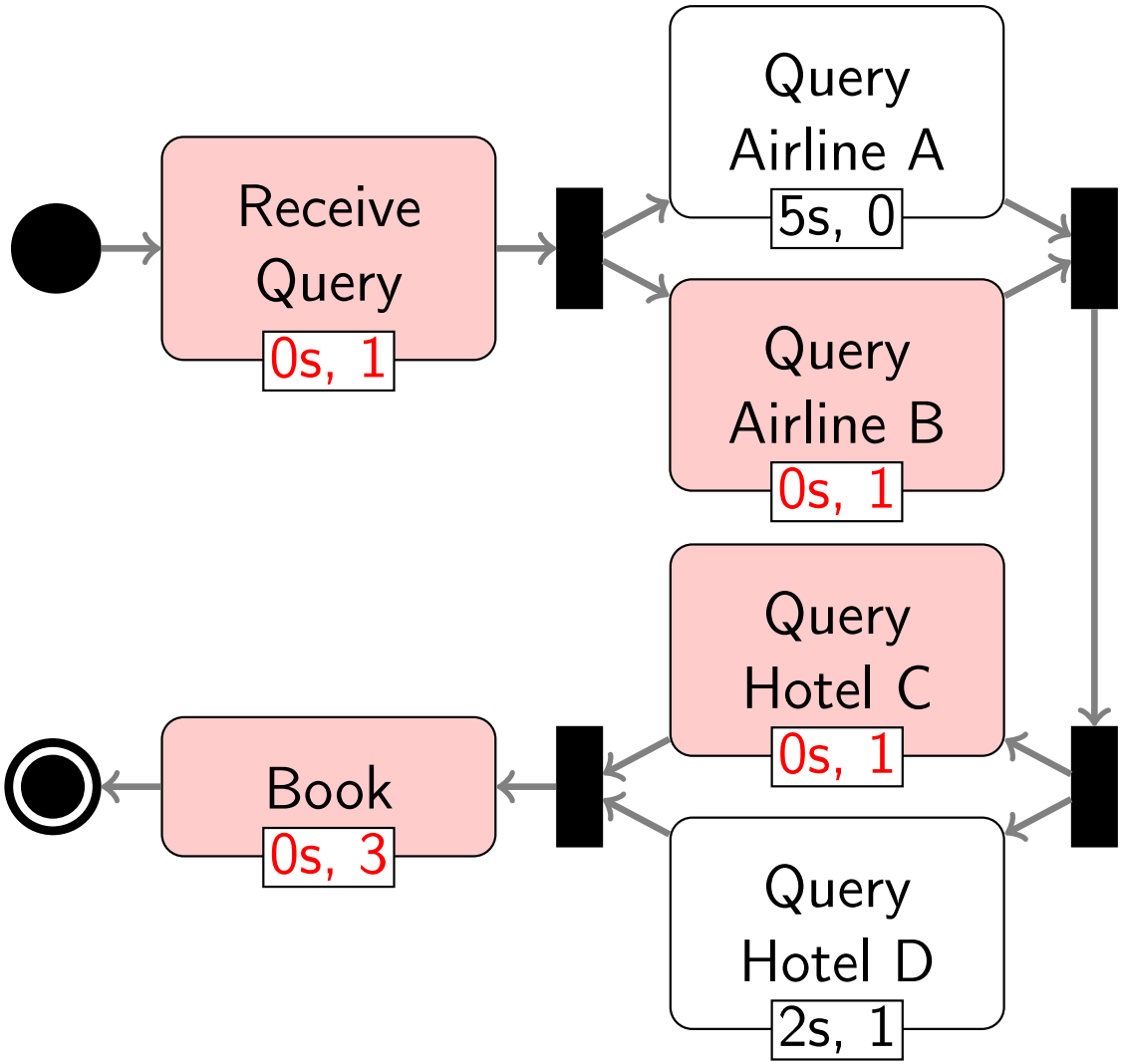

## Requirements

- DAG
- 1 source
- 1+ sinks
- $(m, w)$ pairs for each node

## Info sources

- Historical data
- Signed SLAs
- Experience

# Example: simplified travel search




## Requirements

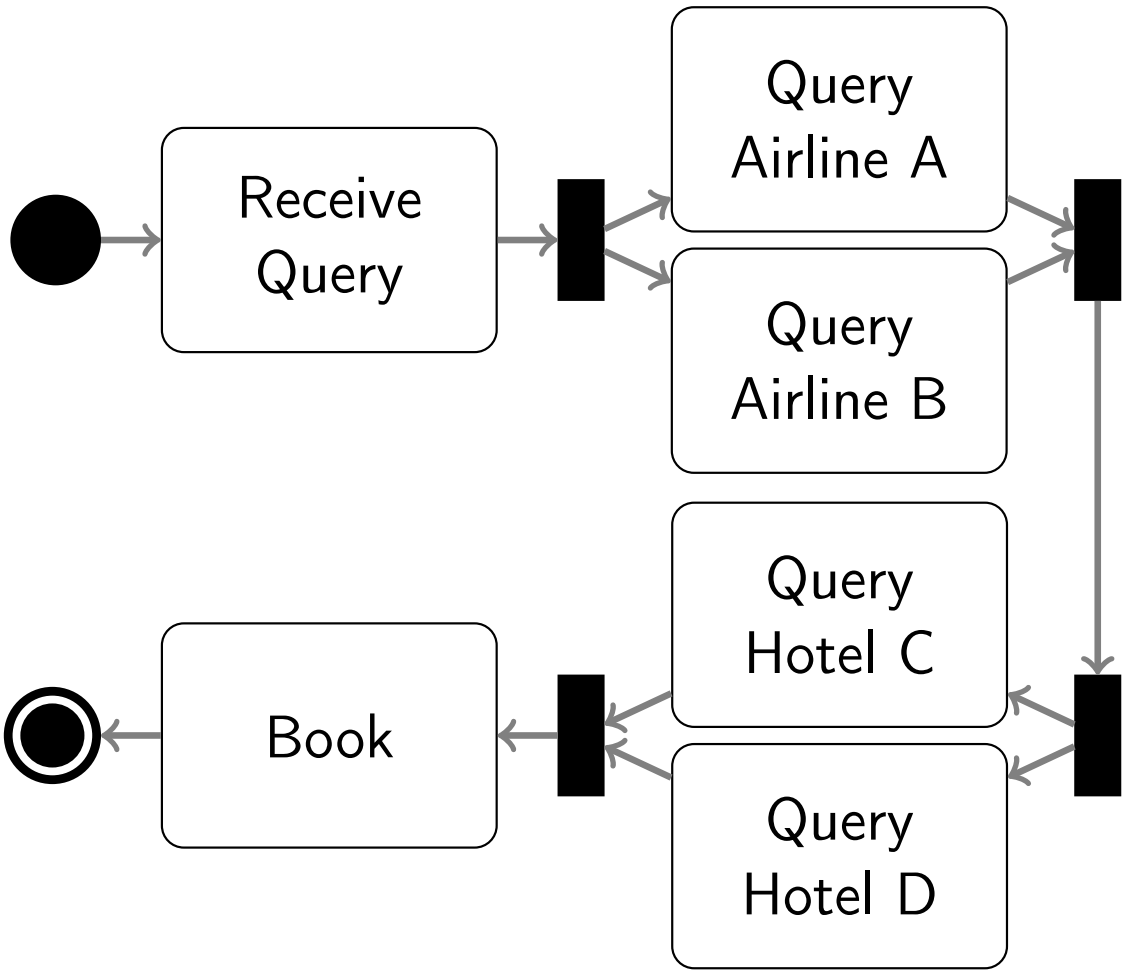
- DAG
- 1 source
- 1+ sinks
- $(m, w)$ pairs for each node

## Info sources

- Historical data
- Signed SLAs
- Experience

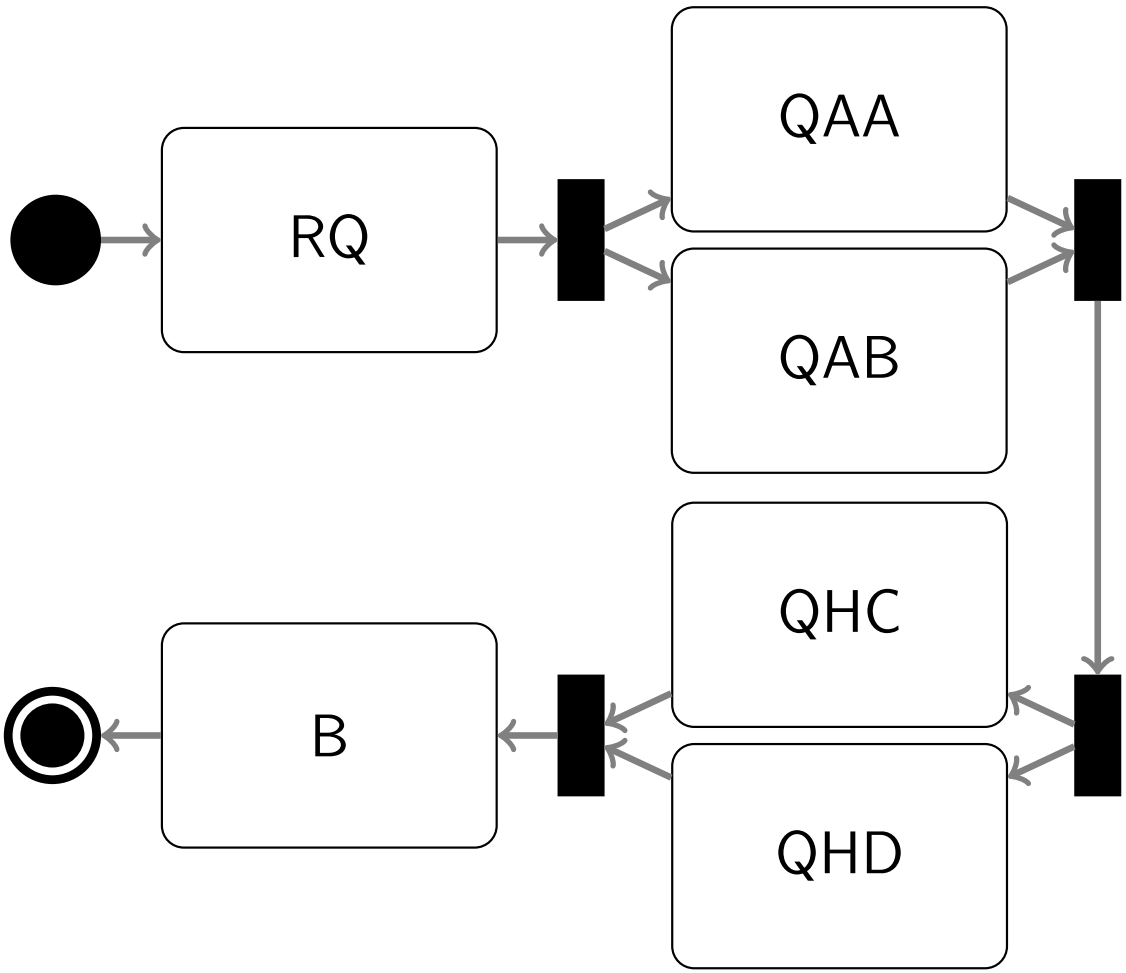
# Example: simplified travel search

We can use any DAG: we will simplify this to an equivalent graph.



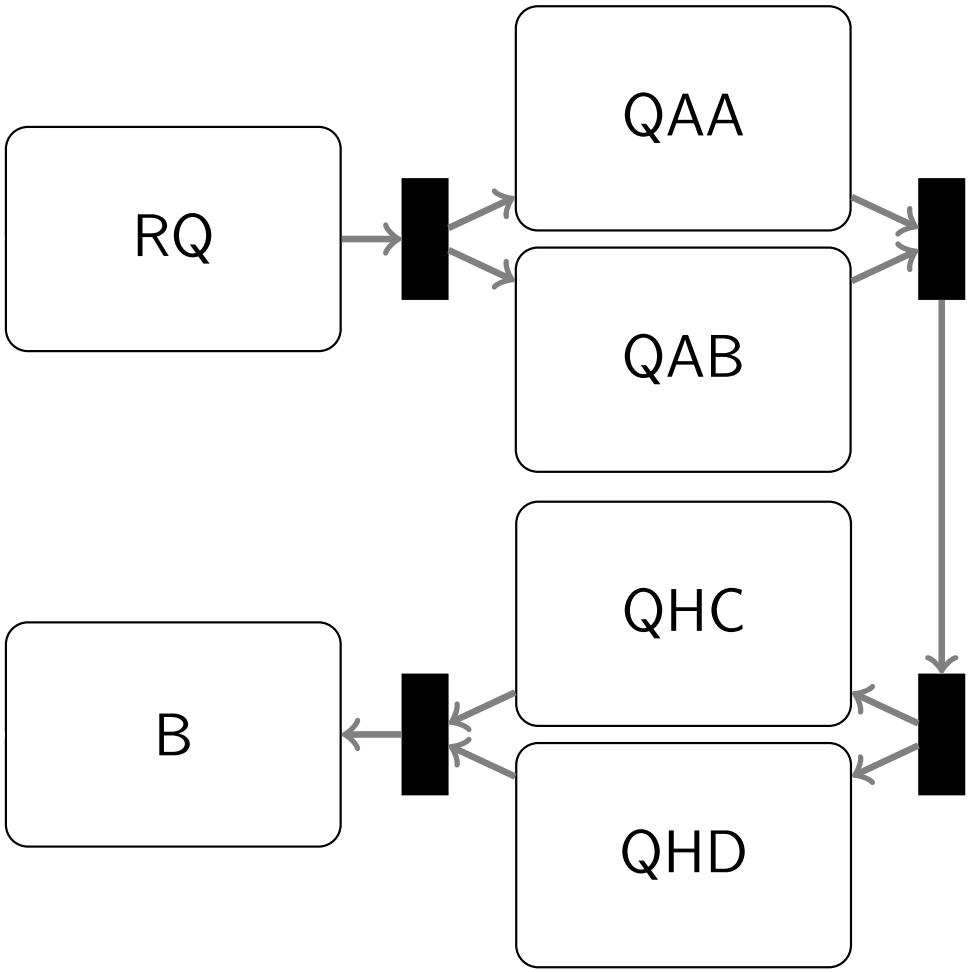
# Example: simplified travel search

To save space, we will shorten the names to their acronyms.



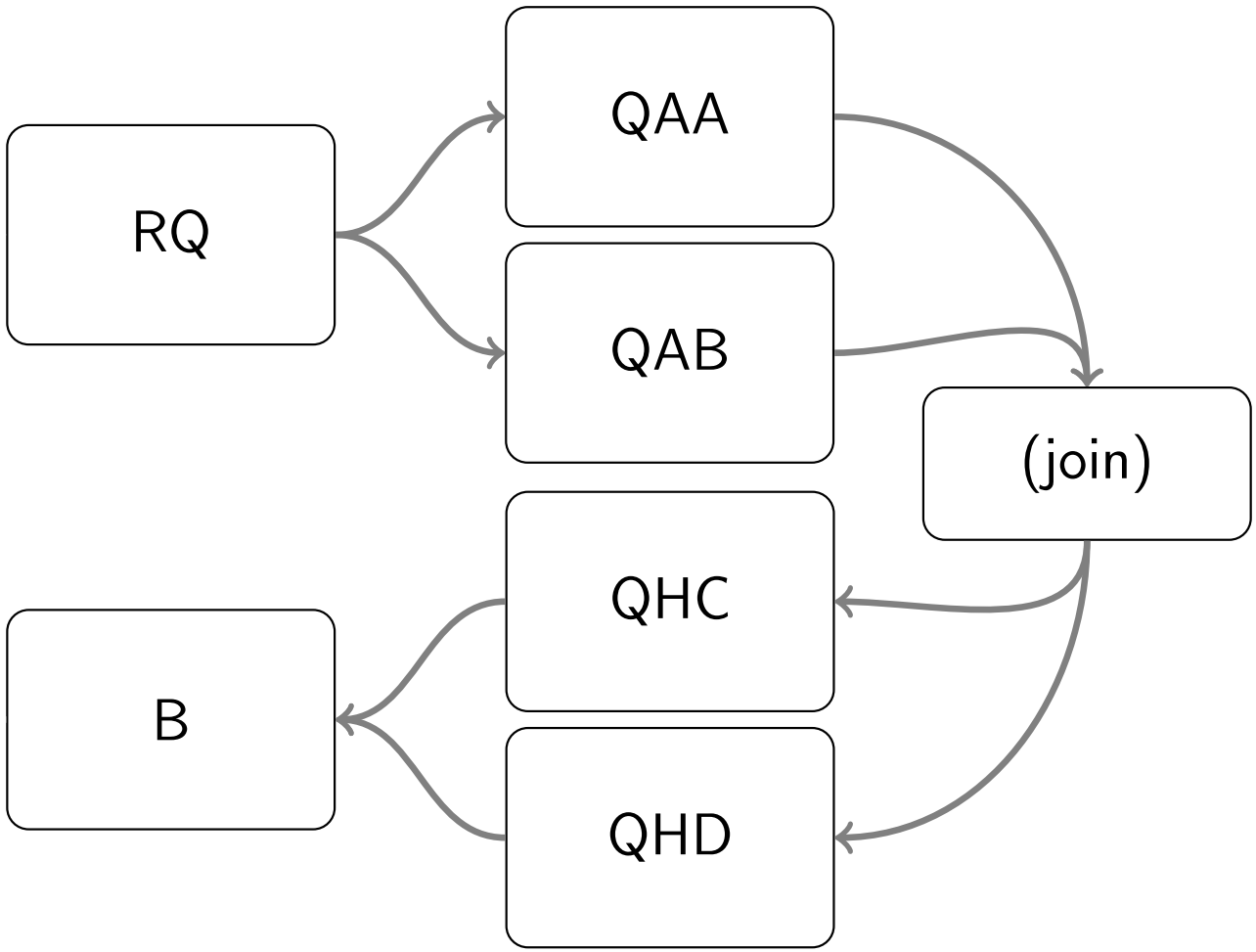
# Example: simplified travel search

Now, we remove the initial and final nodes.



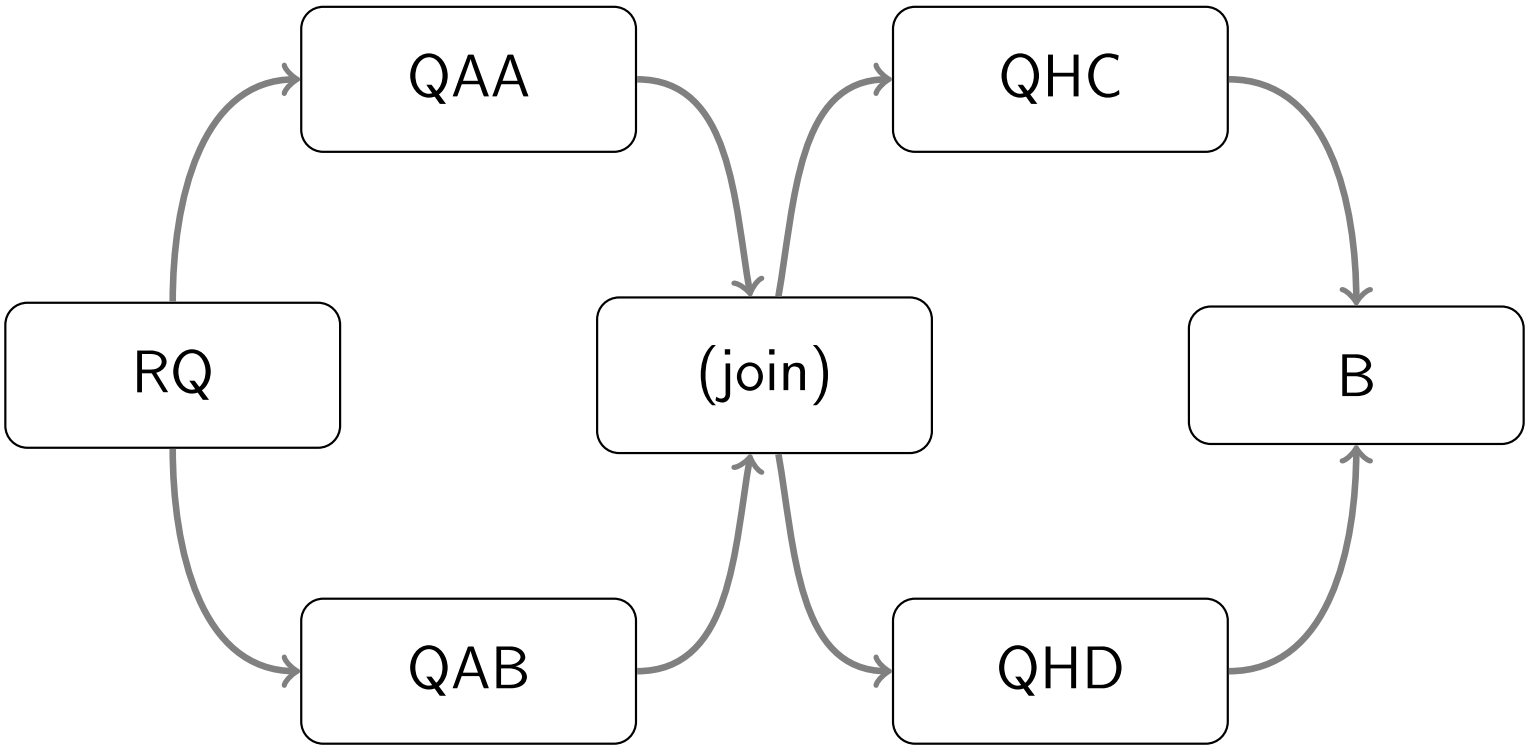
# Example: simplified travel search

We simplify forks and joins to edges and a virtual join node.



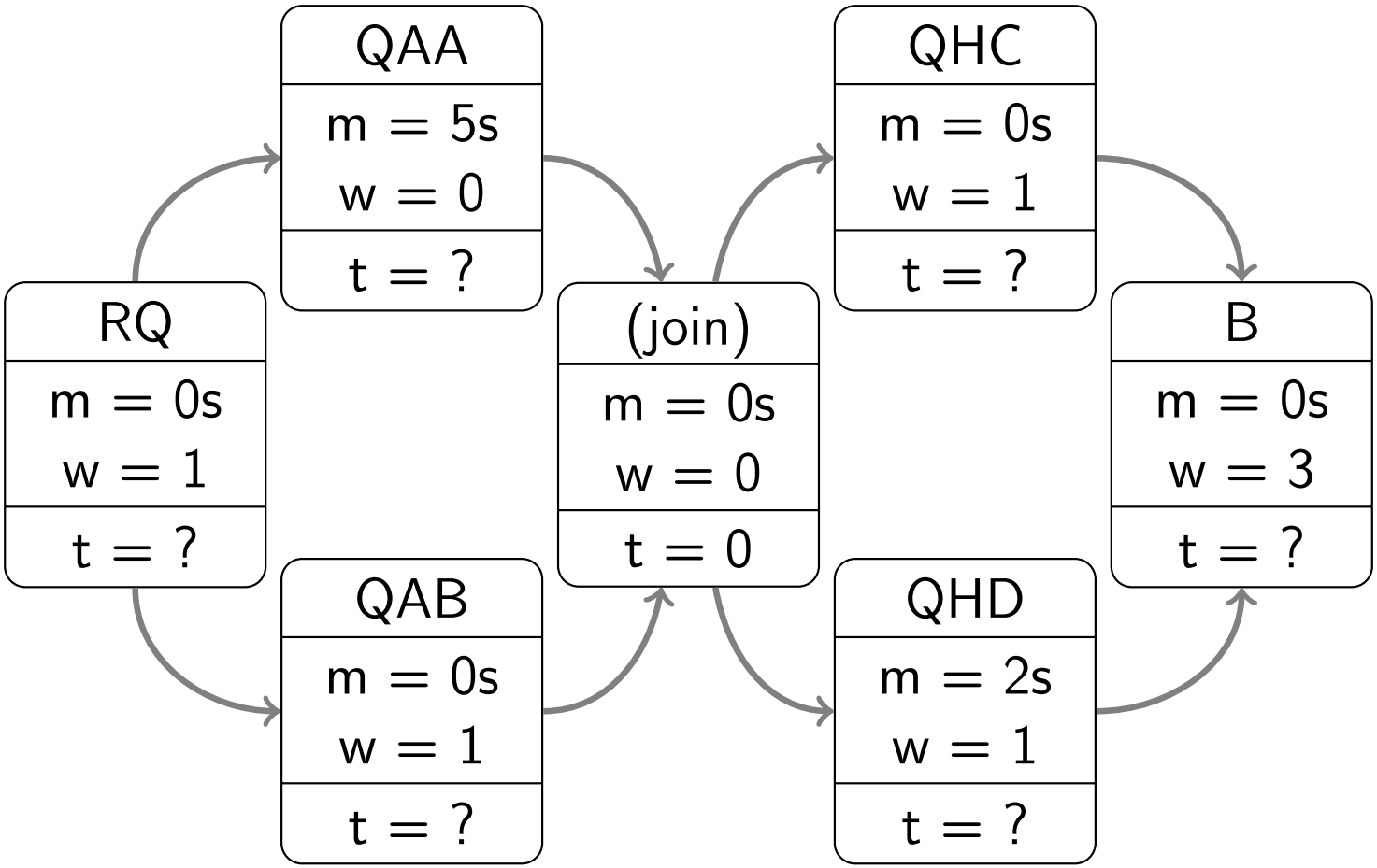
# Example: simplified travel search

We rearrange the nodes for readability.



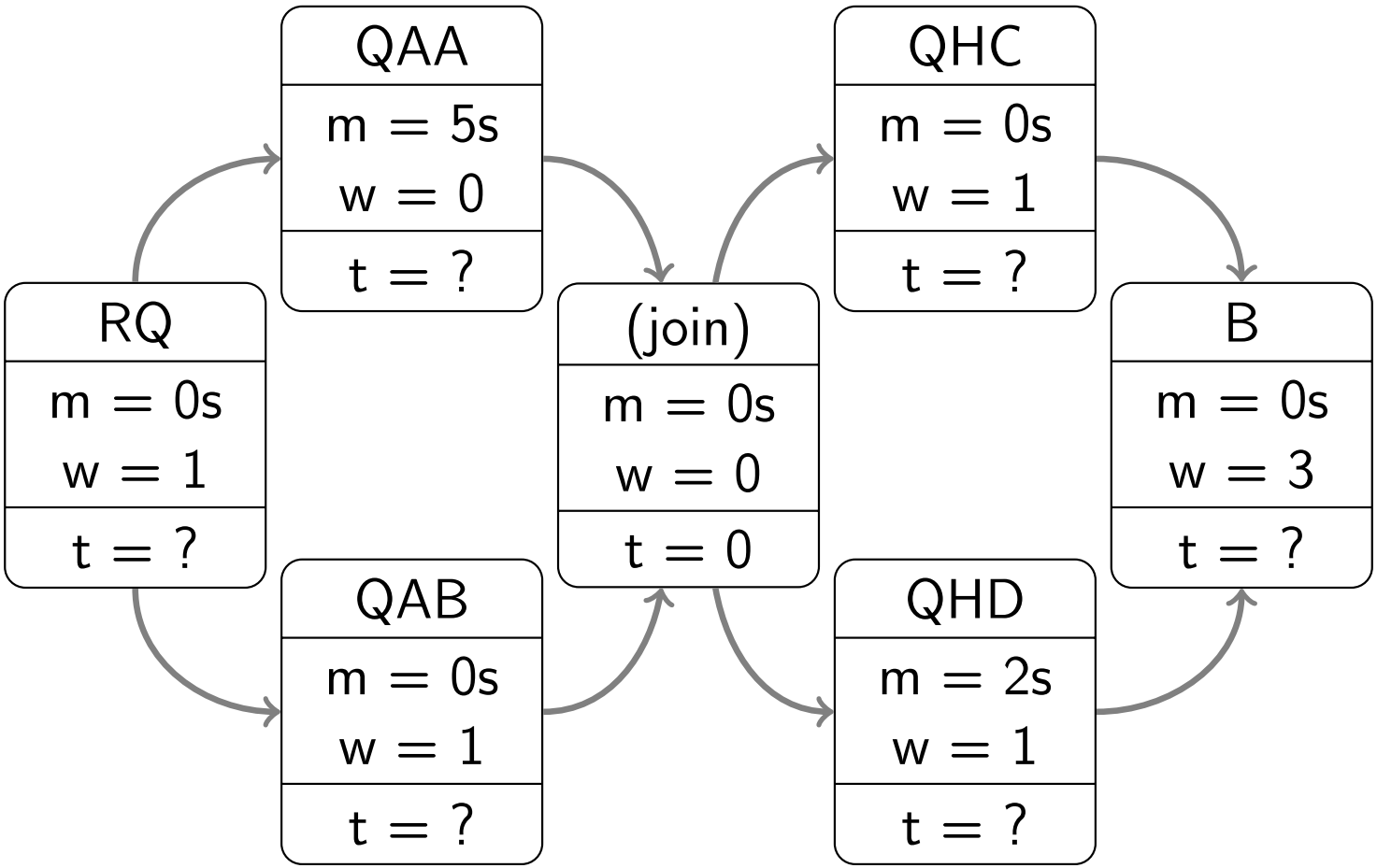
# Example: simplified travel search

We add attributes for $m$, $w$ and the inferred limit $t$.



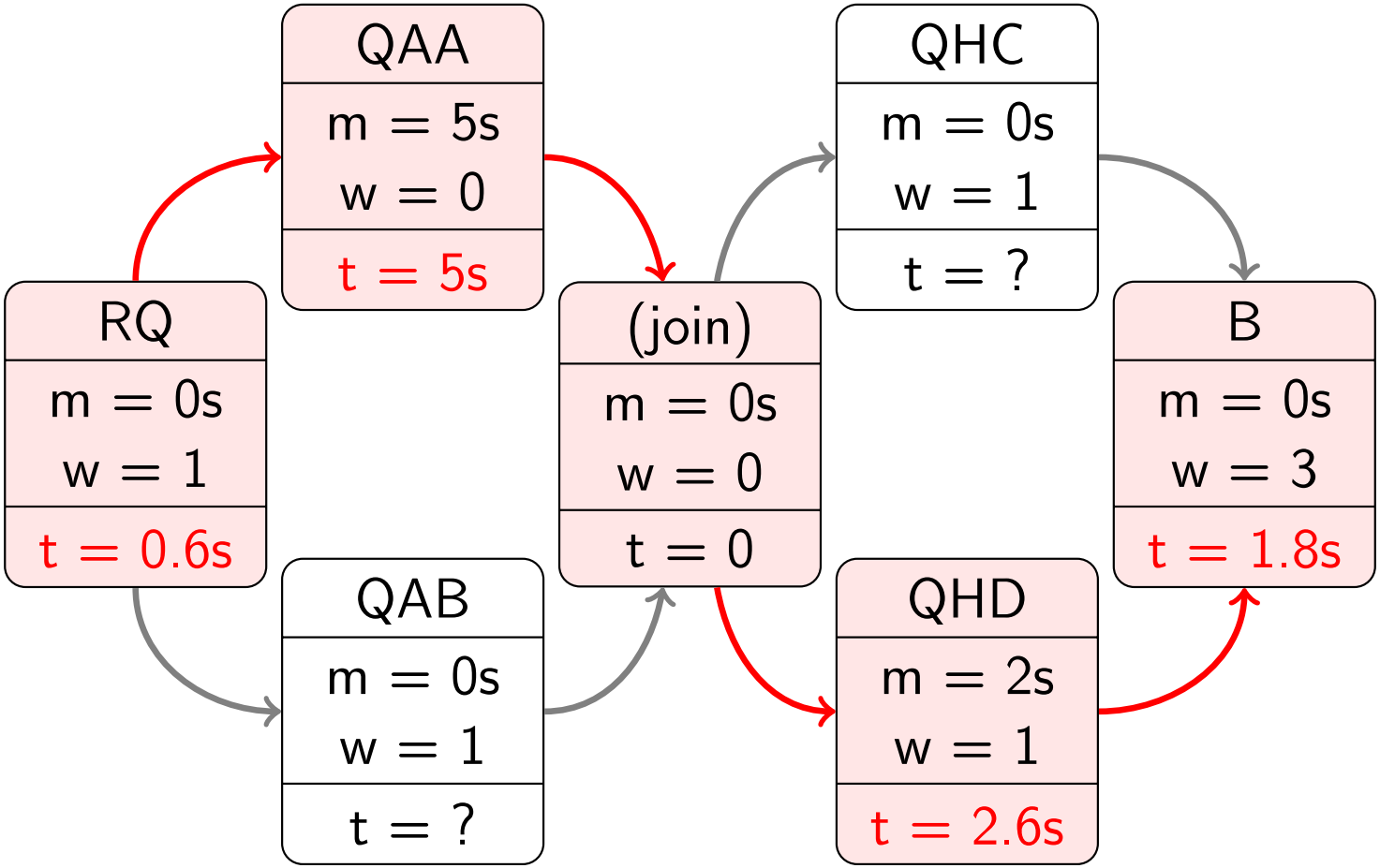
# Exhaustive algorithm: execution (limit = 10 seconds)

We will now run the exhaustive algorithm, with $L = 10s$.



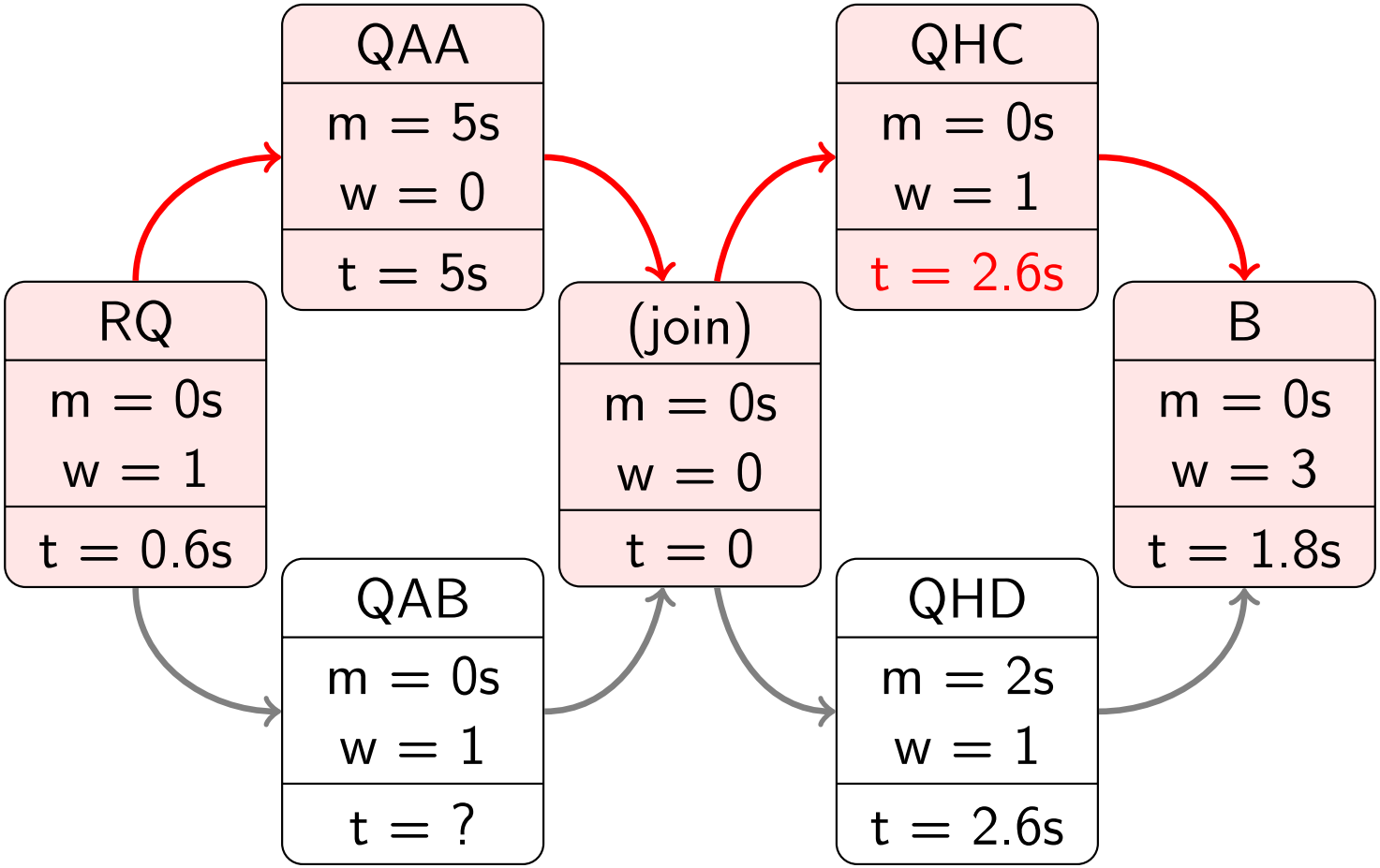
# Exhaustive algorithm: execution (limit = 10 seconds)

Strictest path: assign $(10 - 7)/5 = 0.6s$ per unit of weight.



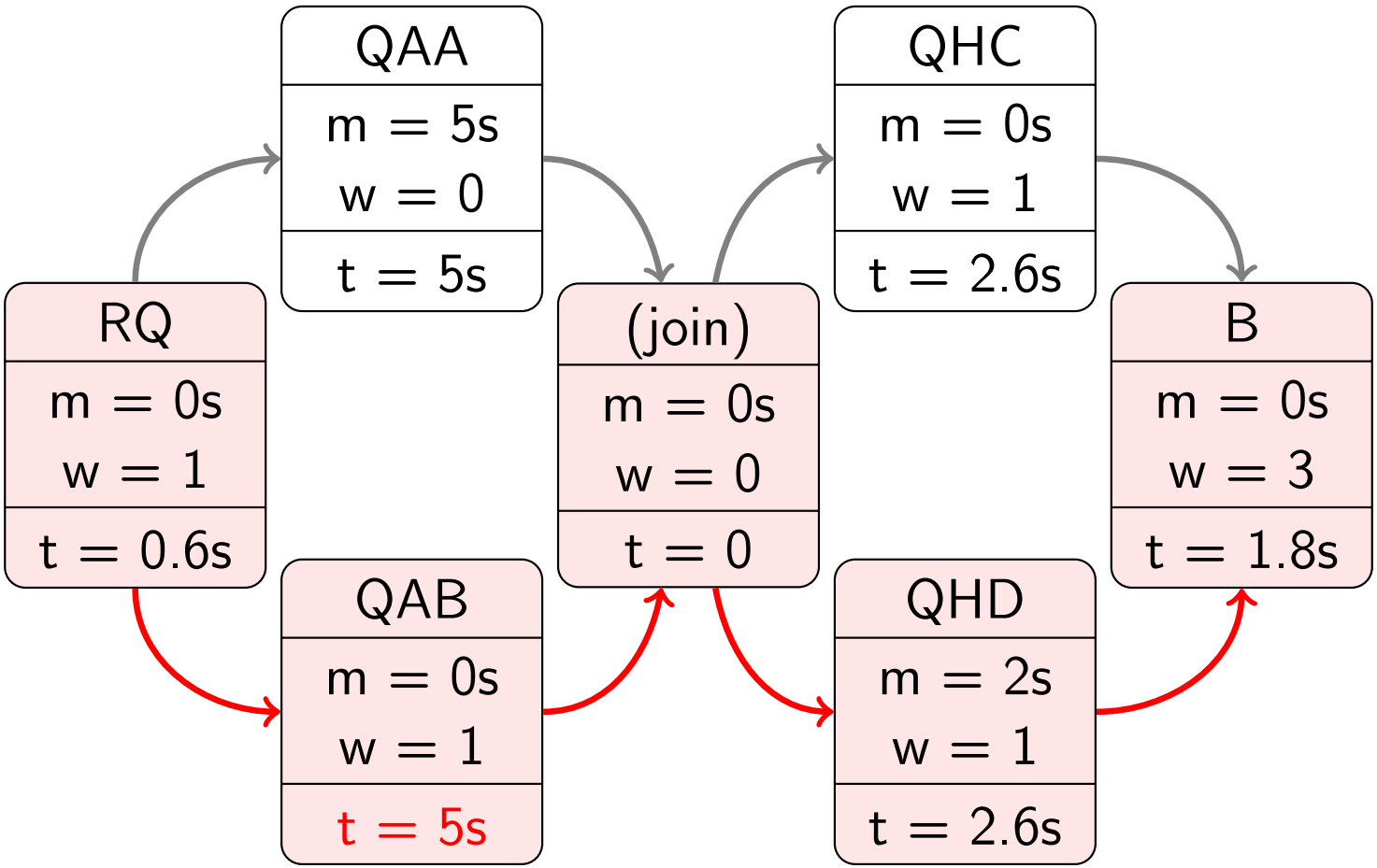
# Exhaustive algorithm: execution (limit = 10 seconds)

Second path: assign $(10 - 7.4)/1 = 2.6s$ per unit of weight.



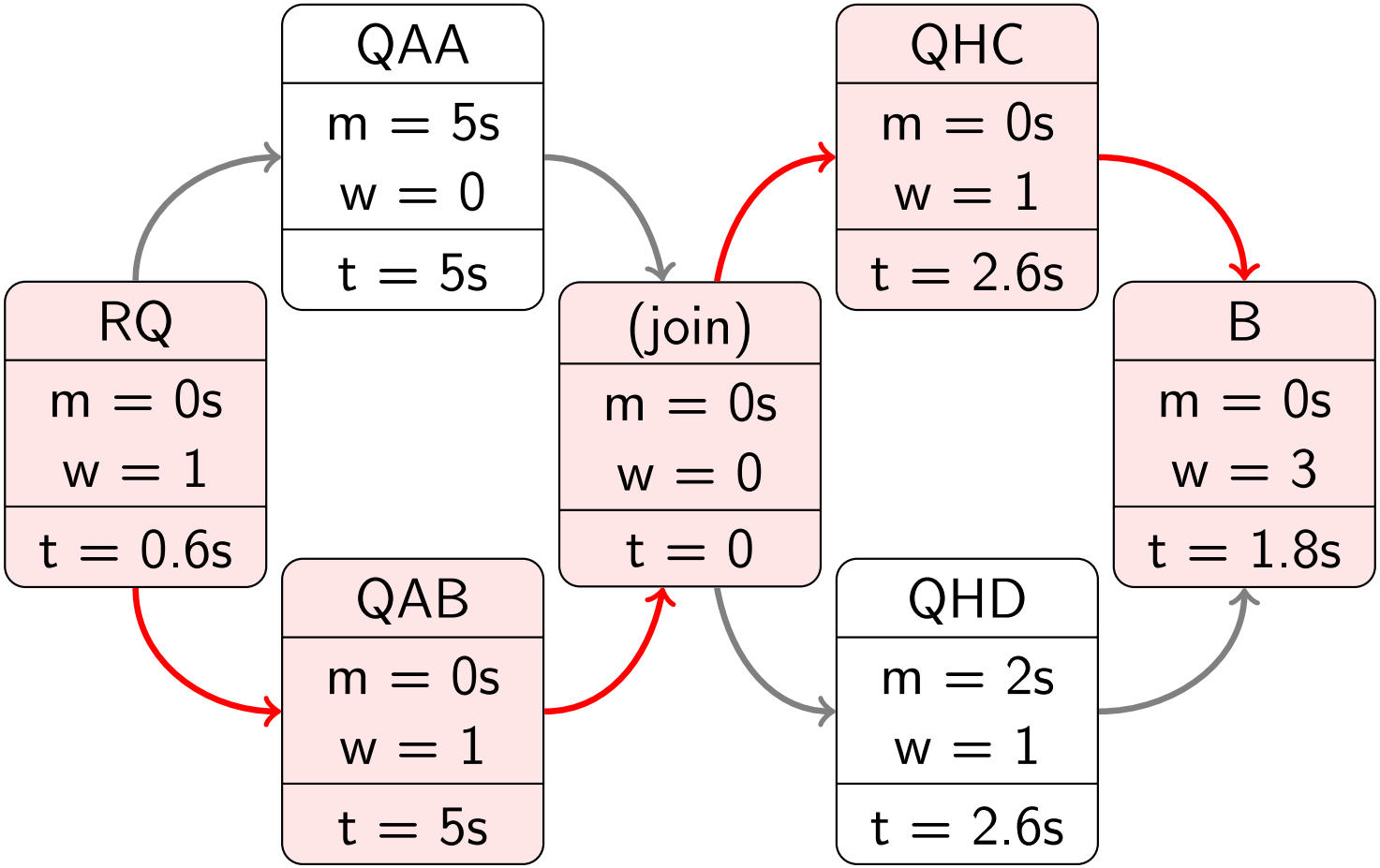
# Exhaustive algorithm: execution (limit = 10 seconds)

Third path: assign $(10 - 5)/1 = 5s$ per unit of weight.



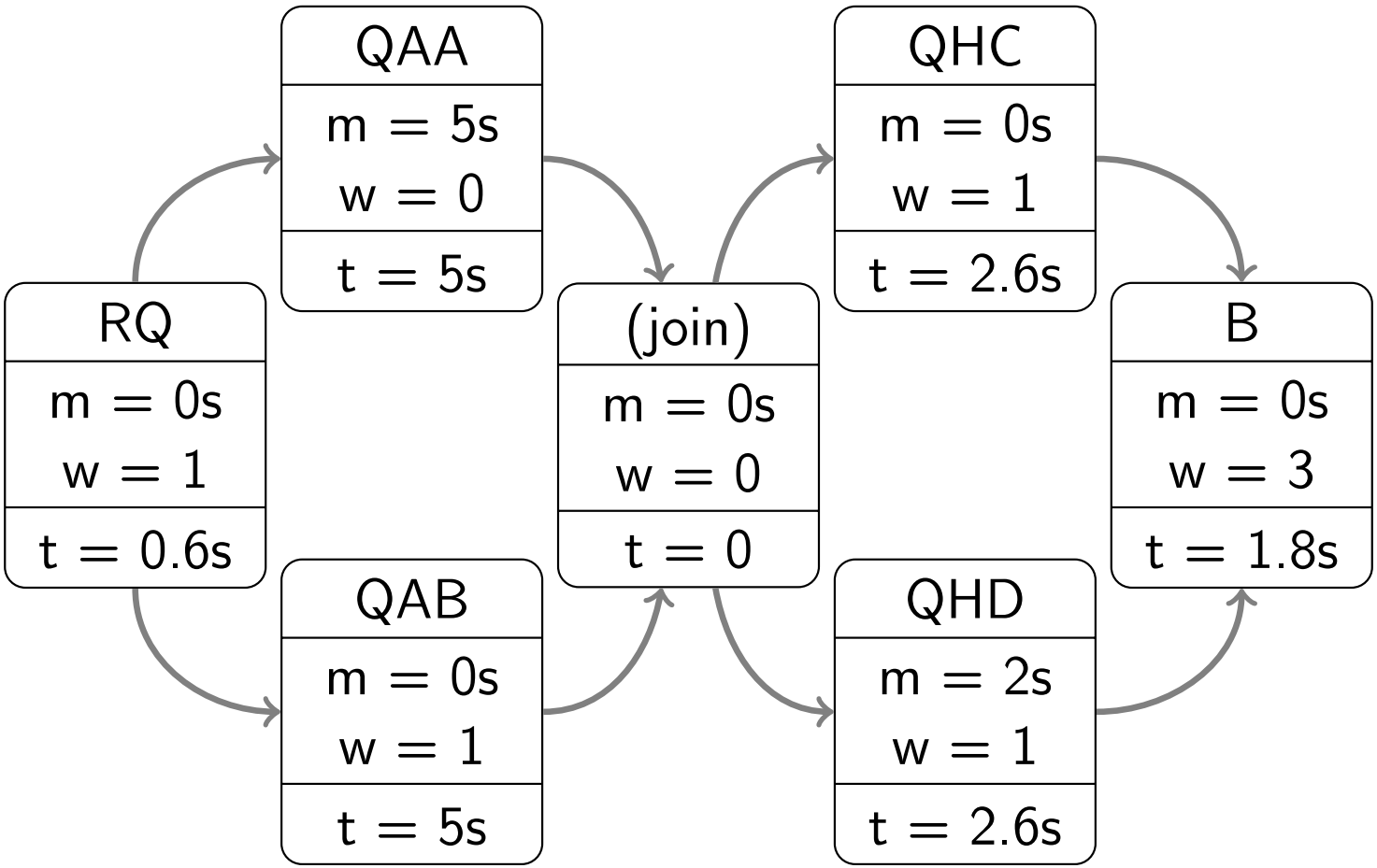
# Exhaustive algorithm: execution (limit = 10 seconds)

Last path: nothing to do.

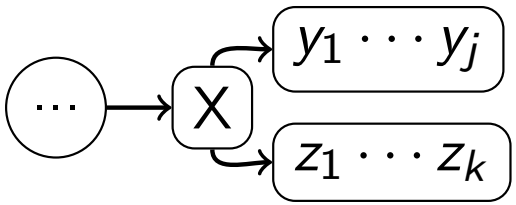


# Exhaustive algorithm: execution (limit = 10 seconds)

Problem: this algorithm is too expensive!



# Incremental algorithm: insights




Let $L$ be the global limit, and:

$$a = \sum m(y_i) \quad b = \sum w(y_i)$$
$$c = \sum m(z_i) \quad d = \sum w(z_i)$$

$y$ is always stricter than $z$ when:

$$a \leq c \wedge (b \leq d \vee a < L \wedge b > d \wedge (b - d)L \leq bc - ad)$$

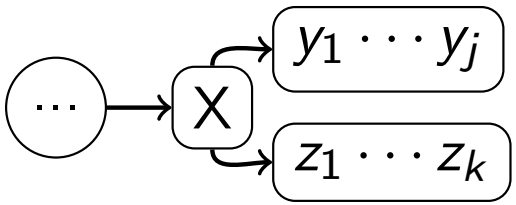
## Discard redundant paths

- We only want the strictest paths each node belongs to
- We compute them from the sinks and back to the source
- In every fork, we discard subpaths which are always less strict than some other

## Visit edges instead of paths

Instead of visiting each path, time flows through the edges and is consumed by the nodes.

# Incremental algorithm: insights




Let $L$ be the global limit, and:

$$a = \sum m(y_i) \quad b = \sum w(y_i)$$
$$c = \sum m(z_i) \quad d = \sum w(z_i)$$

$y$ is always stricter than $z$ when:

$$a \leq c \wedge (b \leq d \vee a < L \wedge b > d \wedge (b-d)L \leq bc - ad)$$

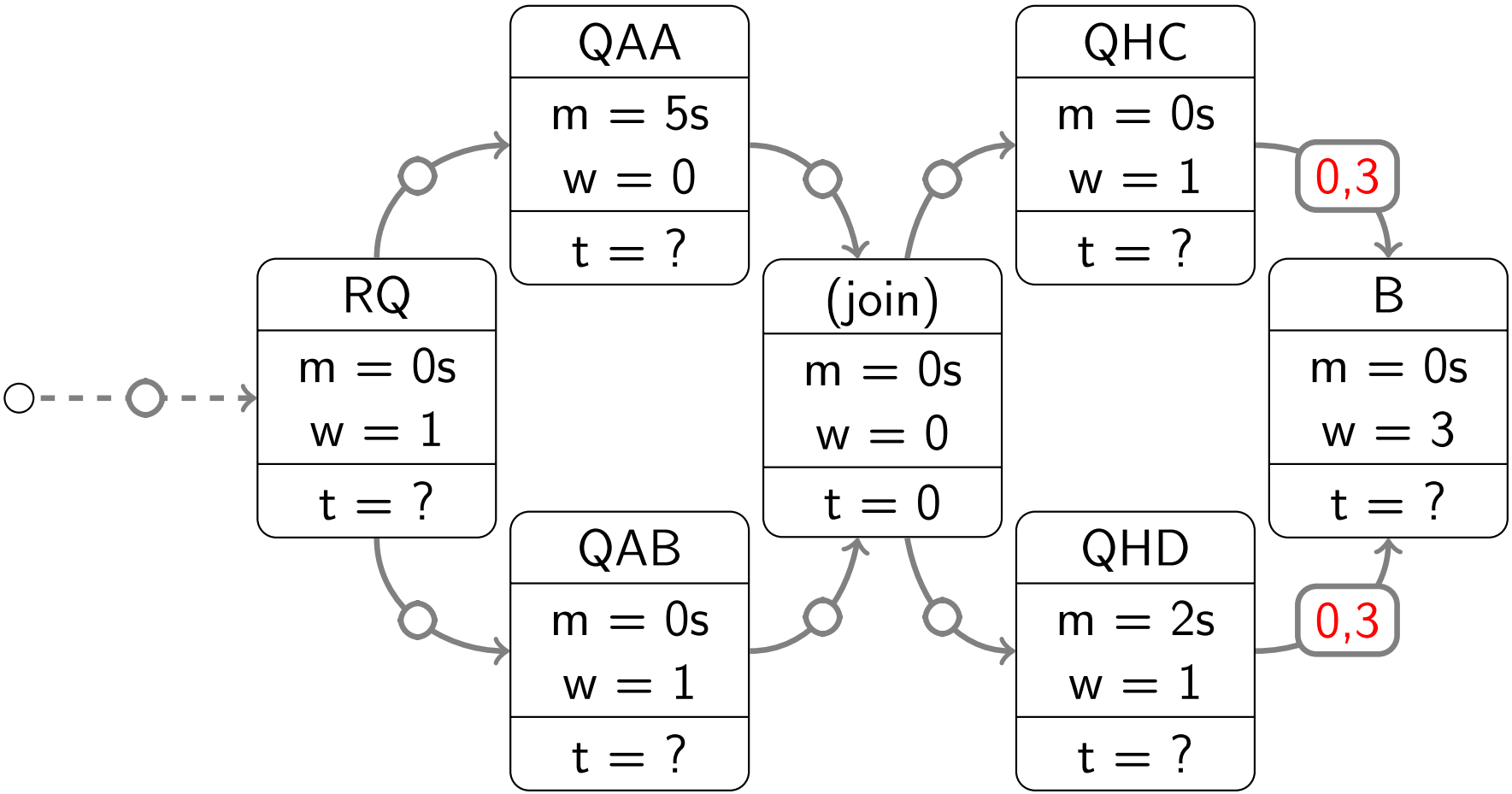
## Discard redundant paths

- We only want the strictest paths each node belongs to
- We compute them from the sinks and back to the source
- In every fork, we discard subpaths which are always less strict than some other

## Visit edges instead of paths

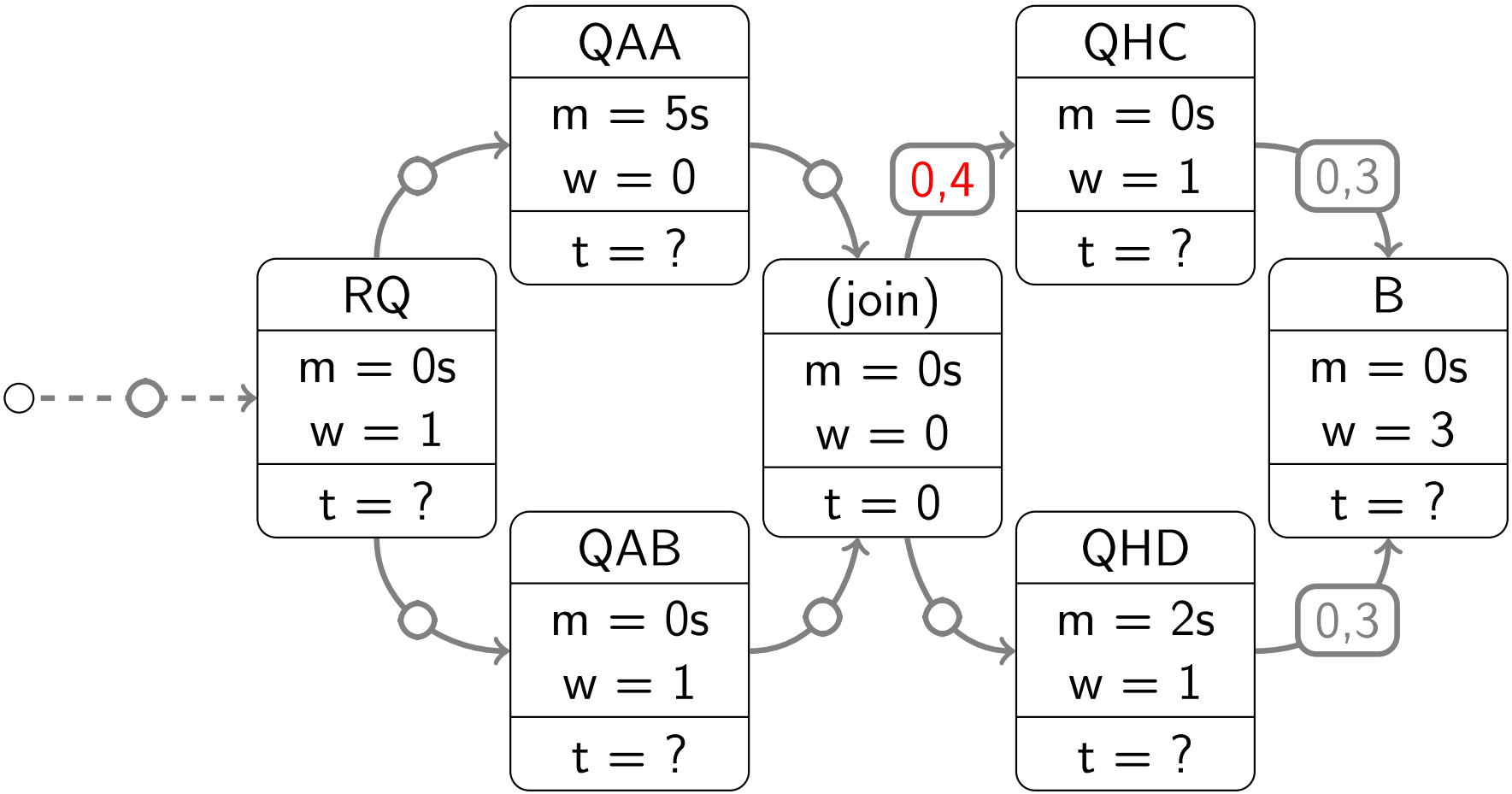
Instead of visiting each path, time flows through the edges and is consumed by the nodes.

# Incremental algorithm: execution (limit = 10 seconds)



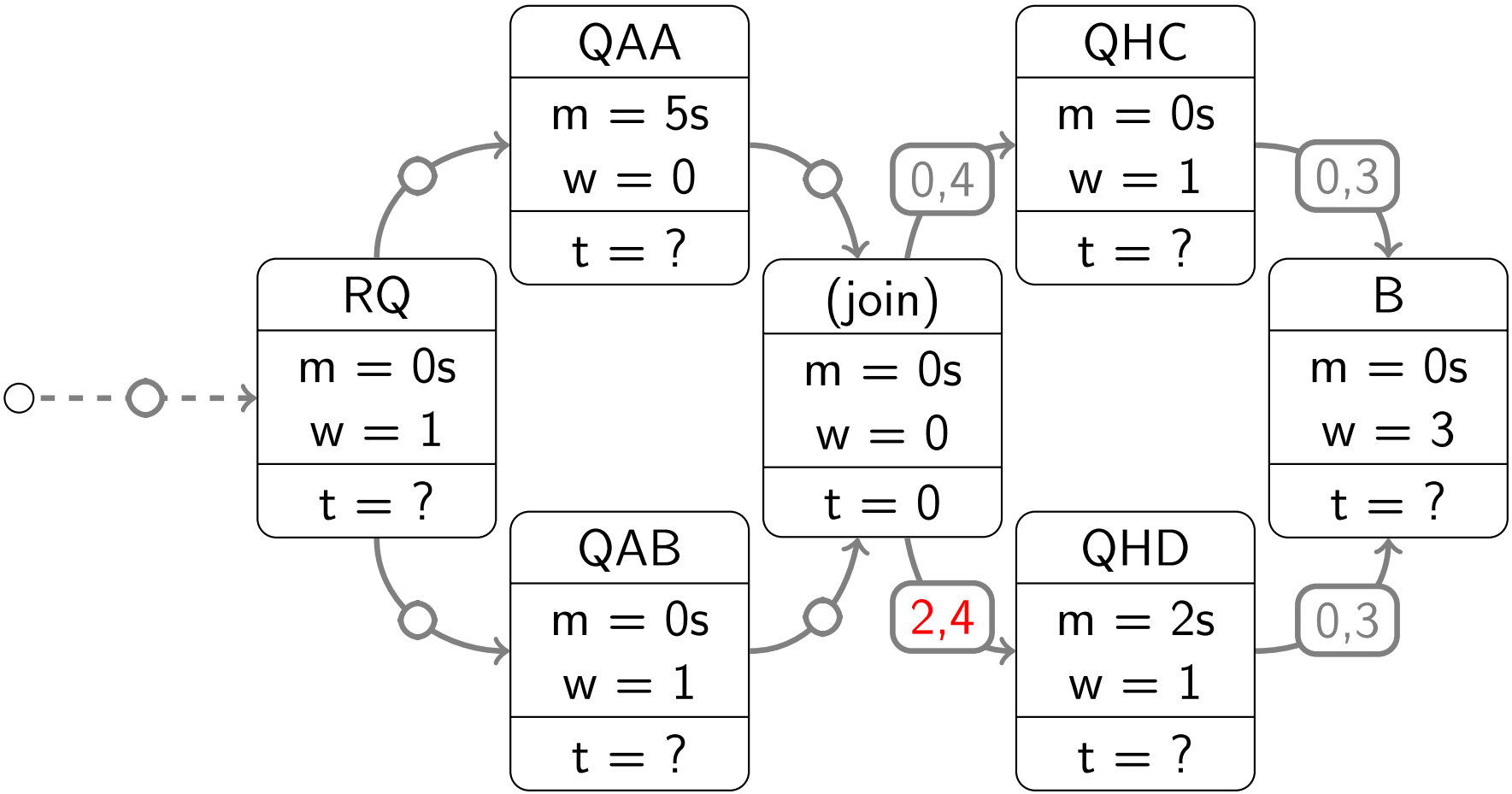

Before we can send the 10s through RQ, we need to compute $(m, w)$ for the strictest subpaths starting at each node. To do that, we traverse the graph in reverse topological order.

# Incremental algorithm: execution (limit = 10 seconds)



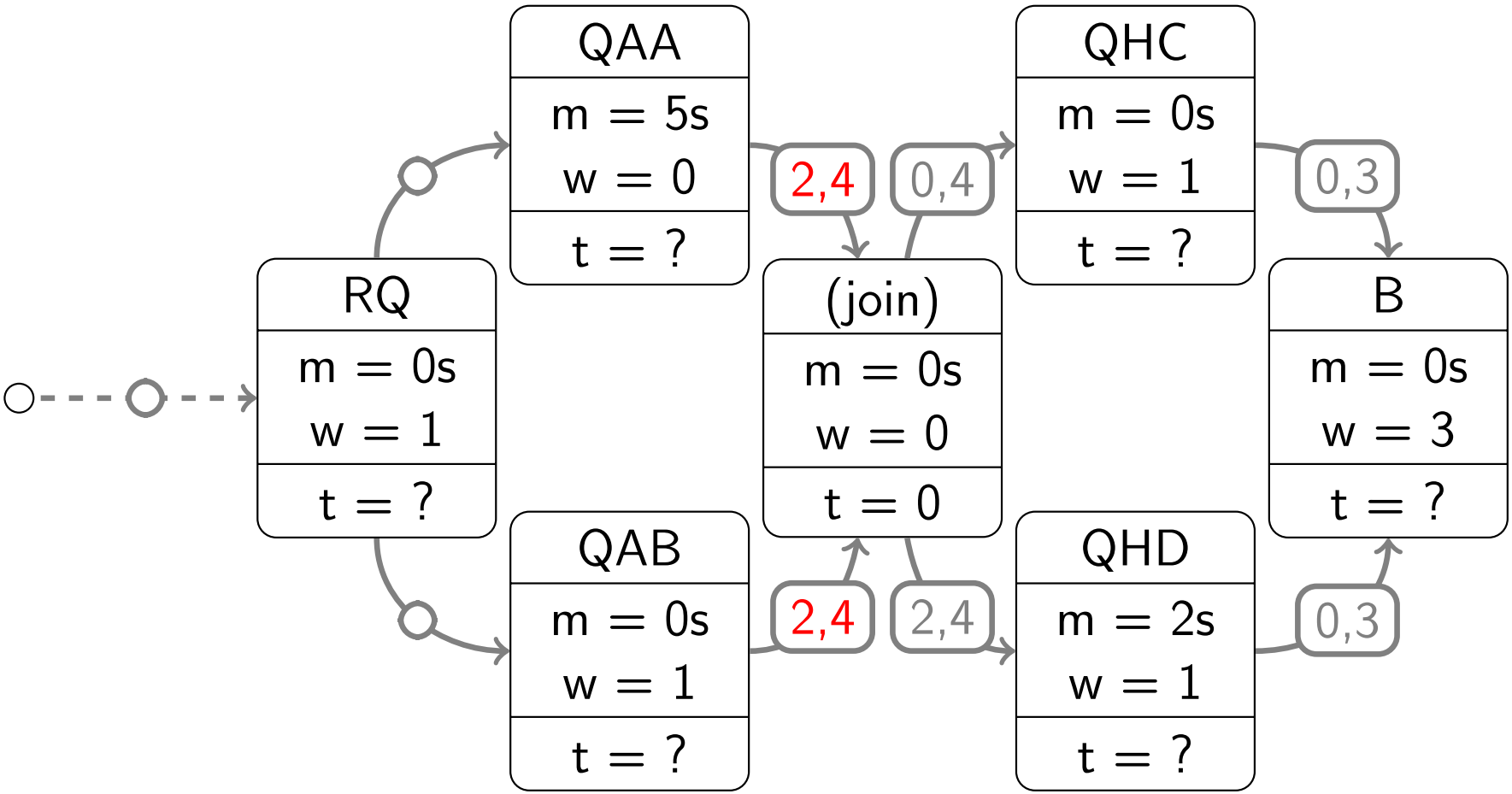

Before we can send the 10s through RQ, we need to compute $(m, w)$ for the strictest subpaths starting at each node. To do that, we traverse the graph in reverse topological order.

# Incremental algorithm: execution (limit = 10 seconds)



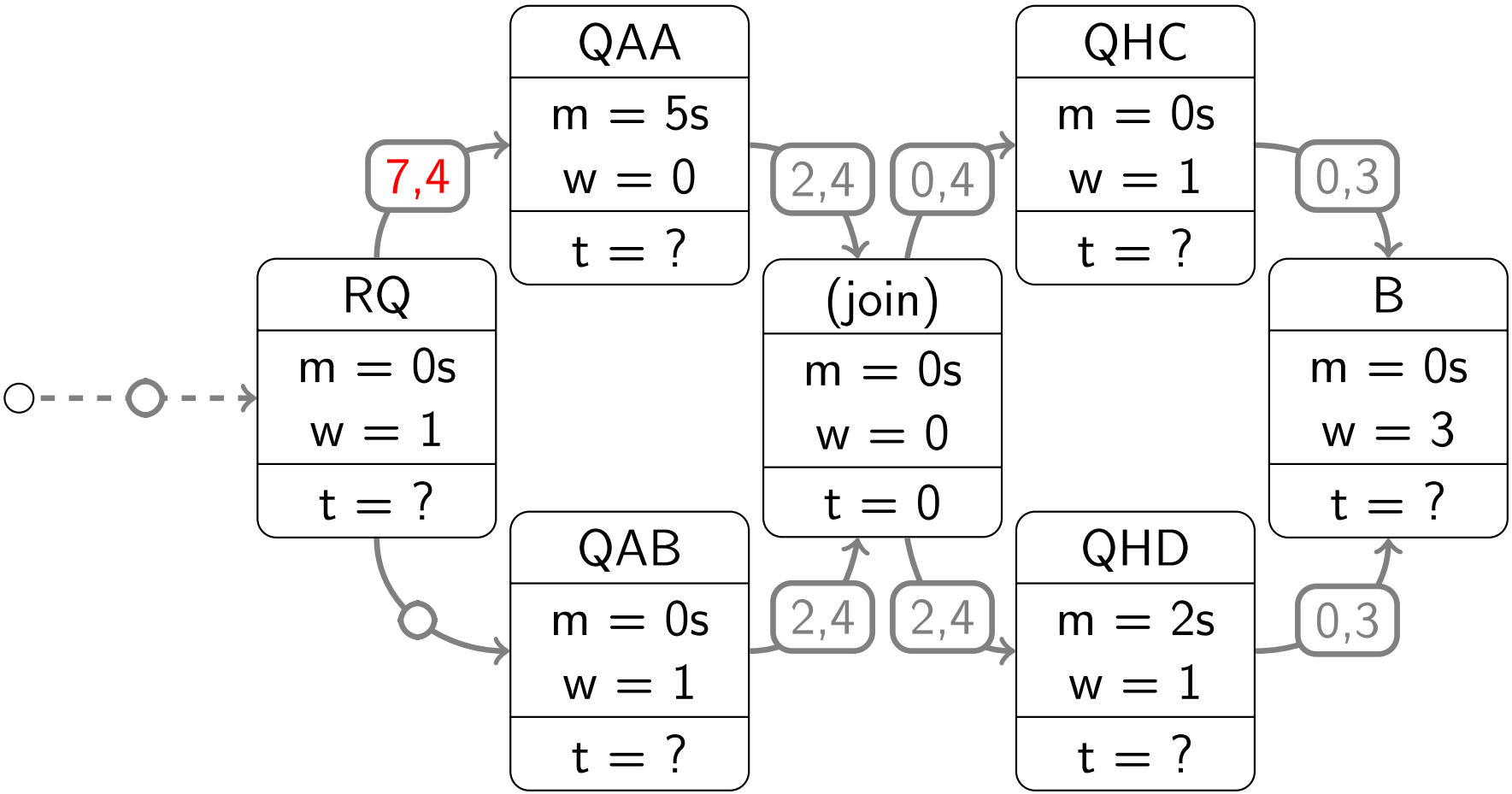

Before we can send the 10s through RQ, we need to compute $(m, w)$ for the strictest subpaths starting at each node. To do that, we traverse the graph in reverse topological order.

# Incremental algorithm: execution (limit = 10 seconds)



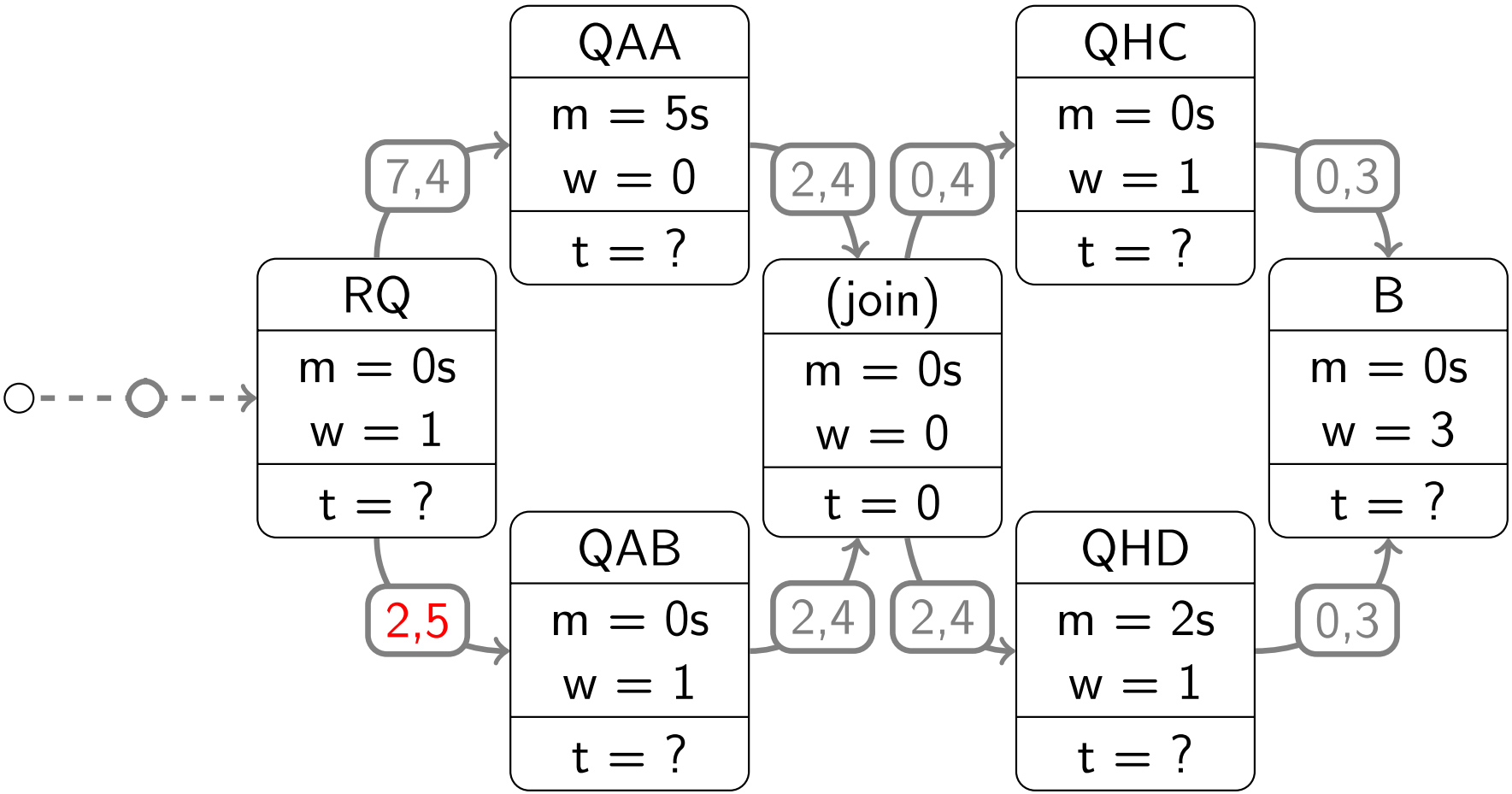

Before we can send the 10s through RQ, we need to compute $(m, w)$ for the strictest subpaths starting at each node. To do that, we traverse the graph in reverse topological order.

# Incremental algorithm: execution (limit = 10 seconds)



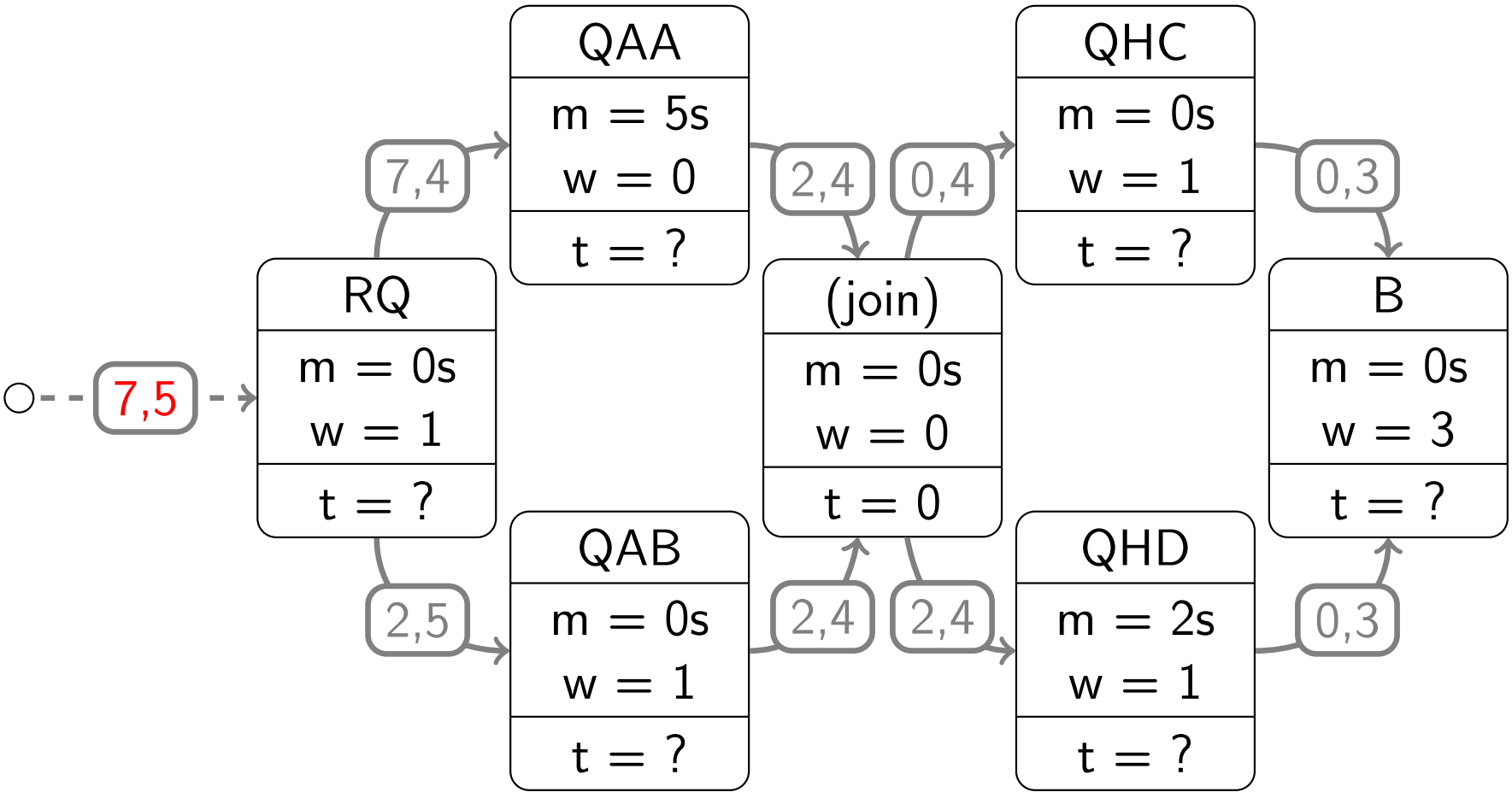

Before we can send the 10s through RQ, we need to compute $(m, w)$ for the strictest subpaths starting at each node. To do that, we traverse the graph in reverse topological order.

# Incremental algorithm: execution (limit = 10 seconds)



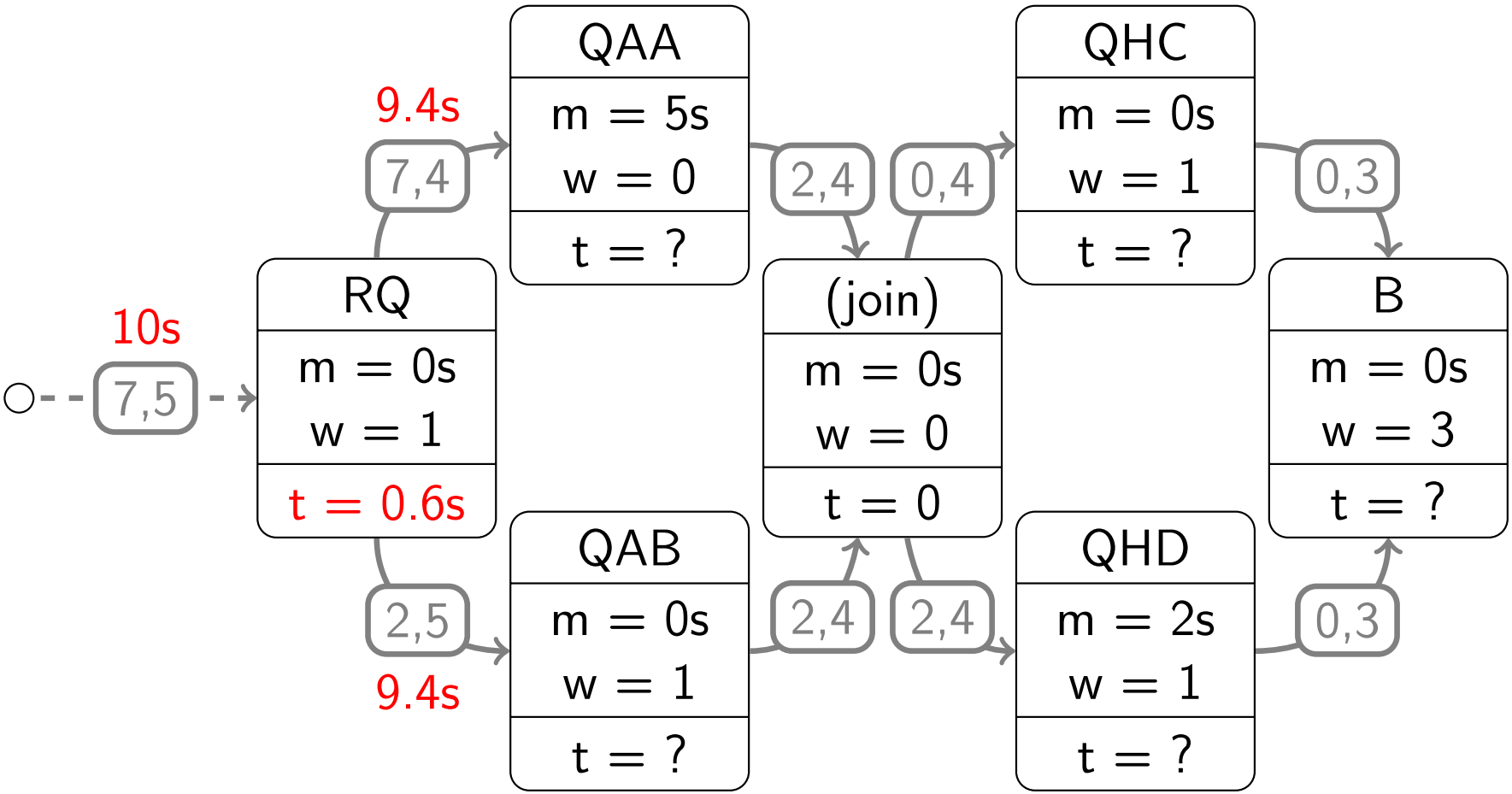

Before we can send the 10s through RQ, we need to compute $(m, w)$ for the strictest subpaths starting at each node. To do that, we traverse the graph in reverse topological order.

# Incremental algorithm: execution (limit = 10 seconds)



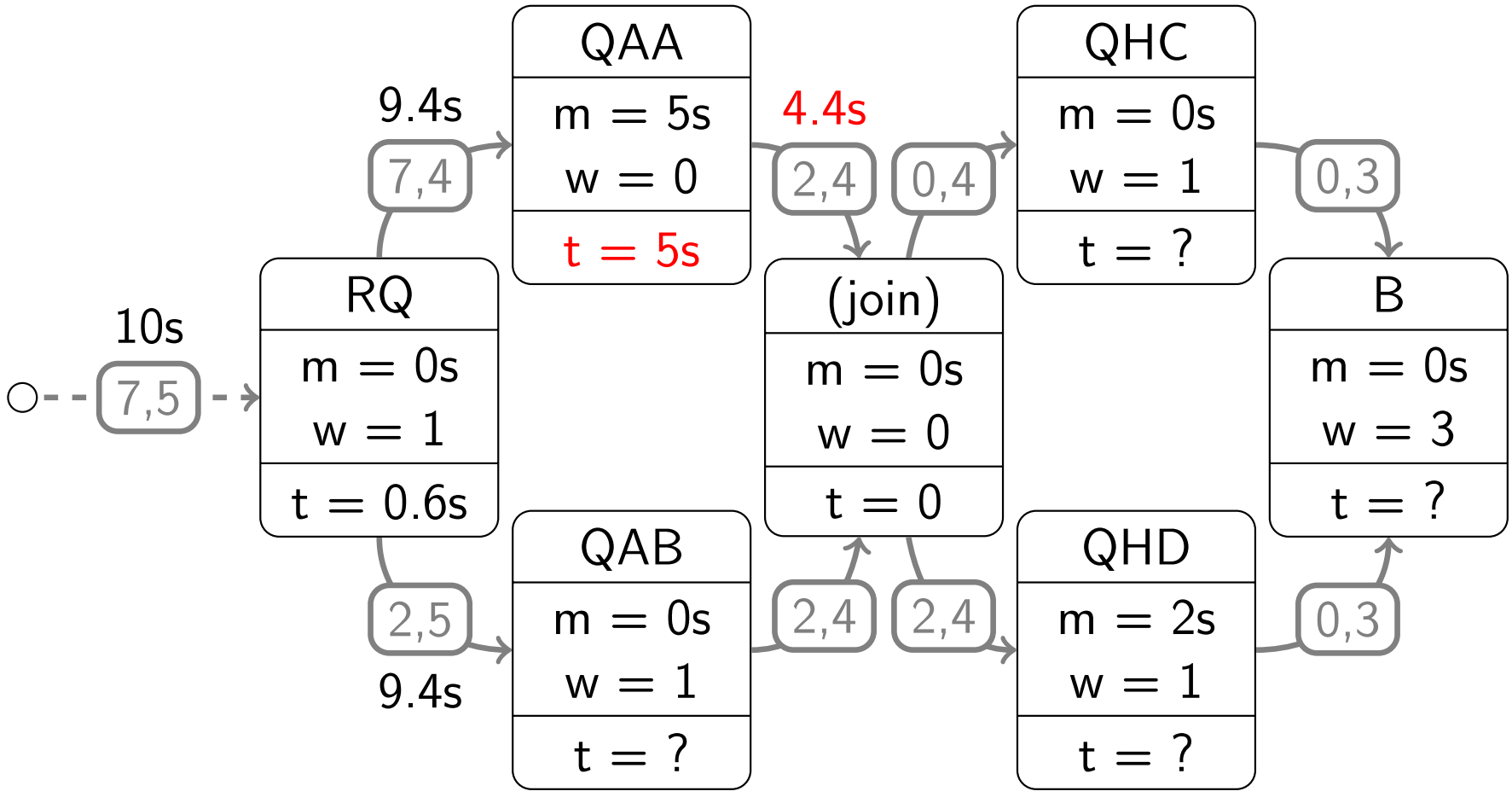

The strictest path through the whole graph (starting at the source) has total minimum time limit of $m = 7s$ and weight $w = 5$.

# Incremental algorithm: execution (limit = 10 seconds)



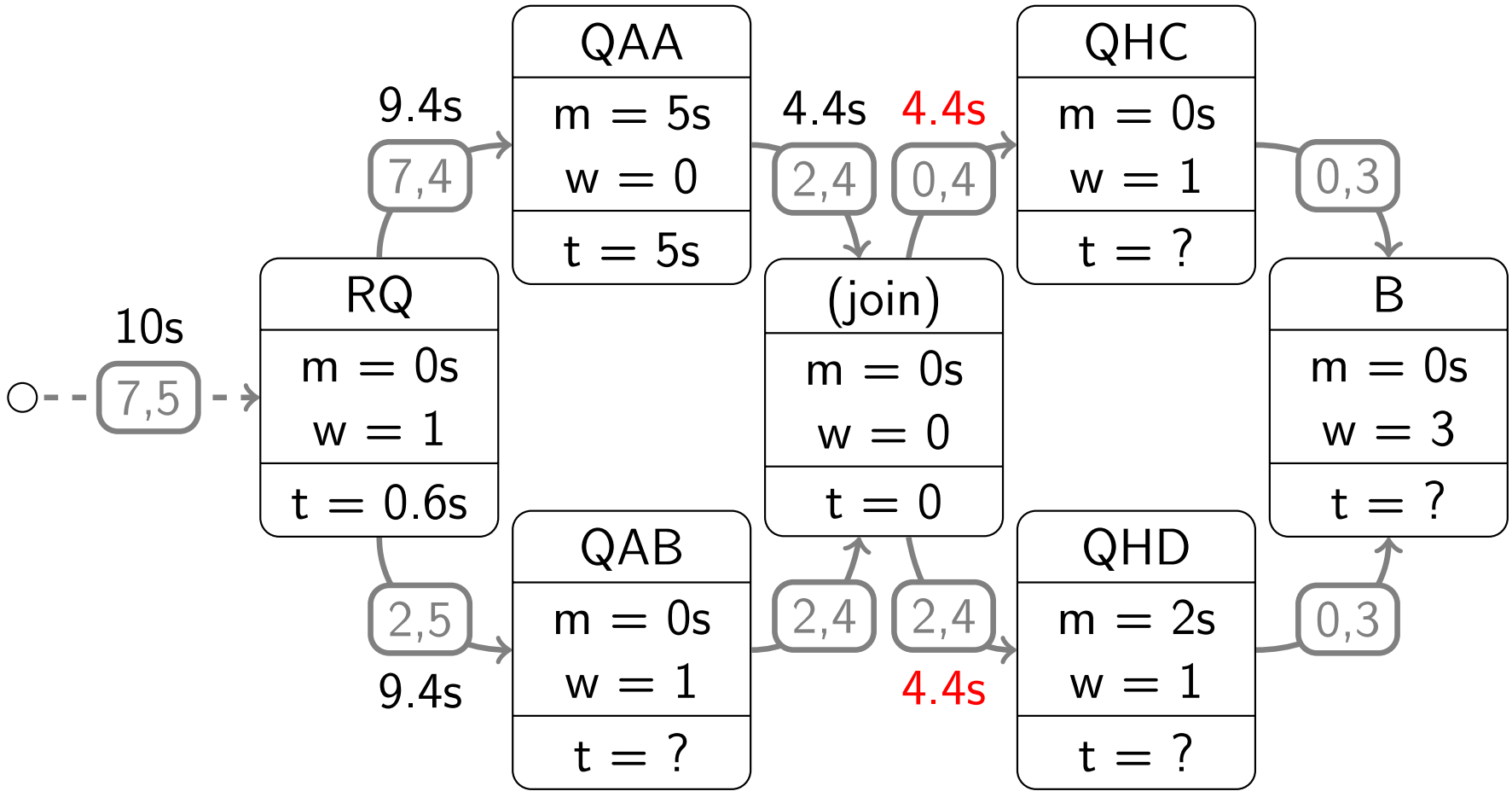

We now send 10 seconds (the global limit) into RQ. RQ takes $0 + (10-7)1/5 = 0.6s$ and sends the remaining 9.4s through its outgoing edges.

# Incremental algorithm: execution (limit = 10 seconds)



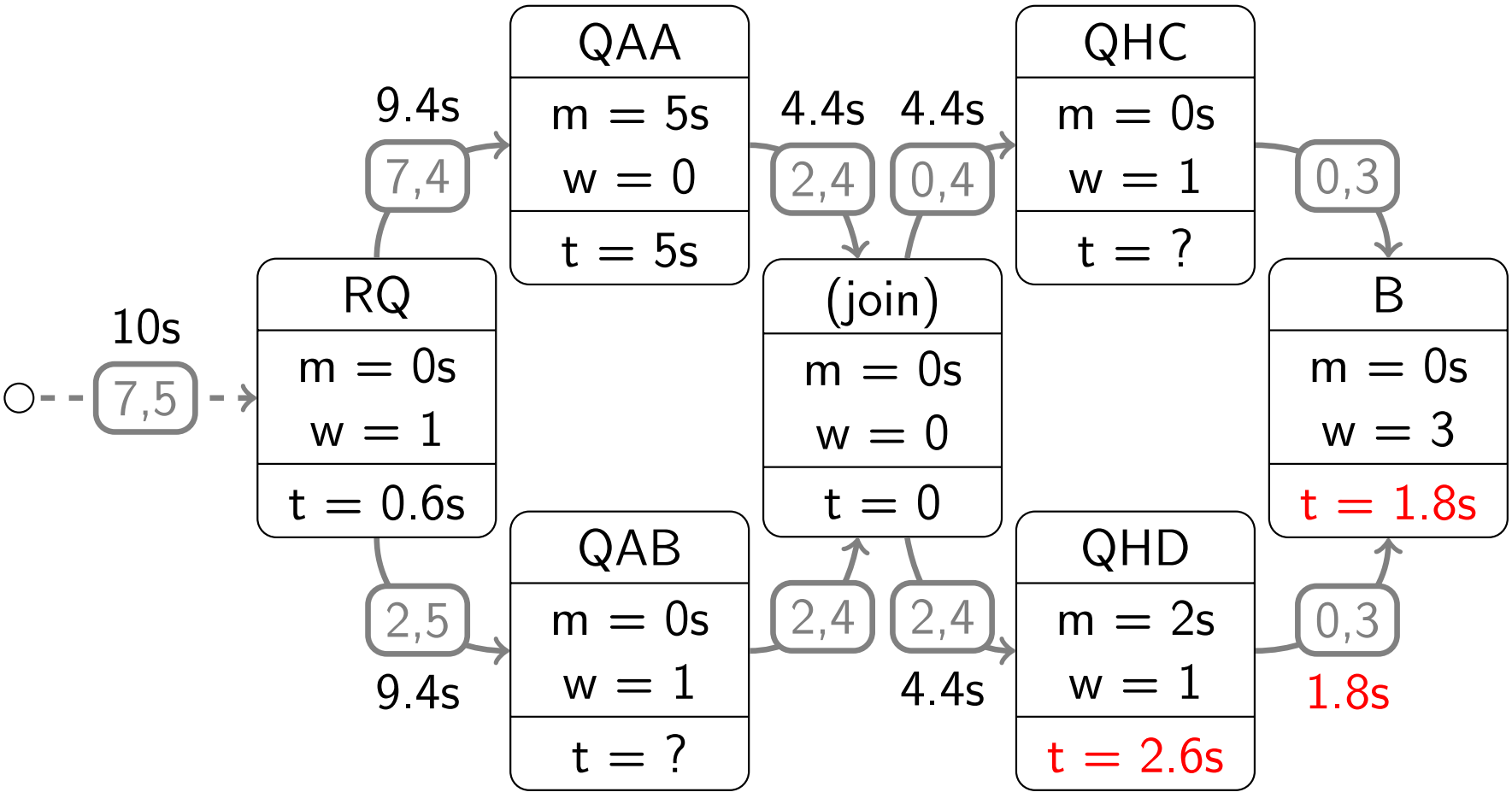

(7,4) is stricter than (2,5): we move to QAA. QAA takes $5 + (9.4 - 7)0/4 = 5s$ and sends the remaining 4.4s.

# Incremental algorithm: execution (limit = 10 seconds)



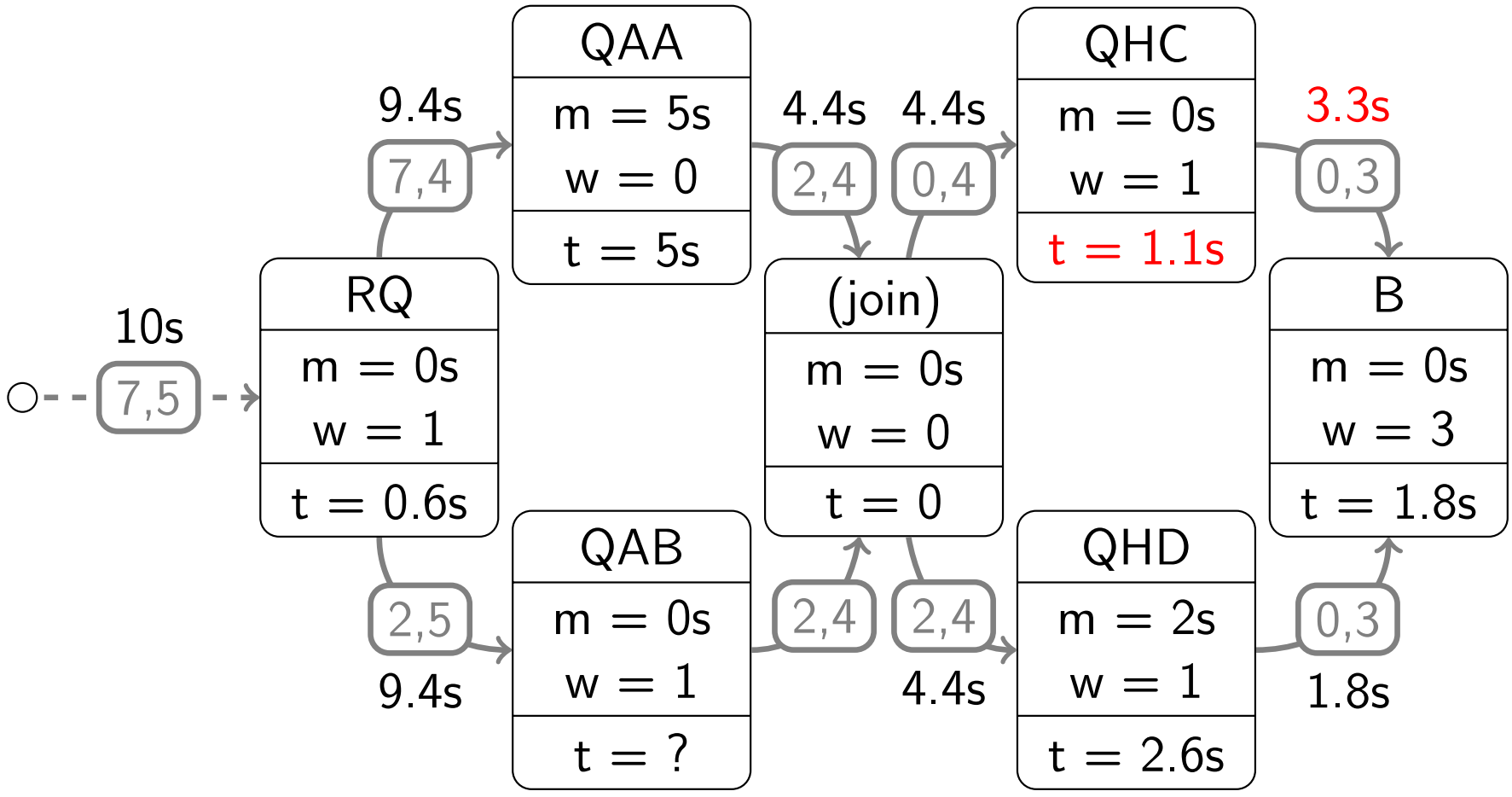

The join node simply sends the 4.4s to its outgoing edges.

# Incremental algorithm: execution (limit = 10 seconds)



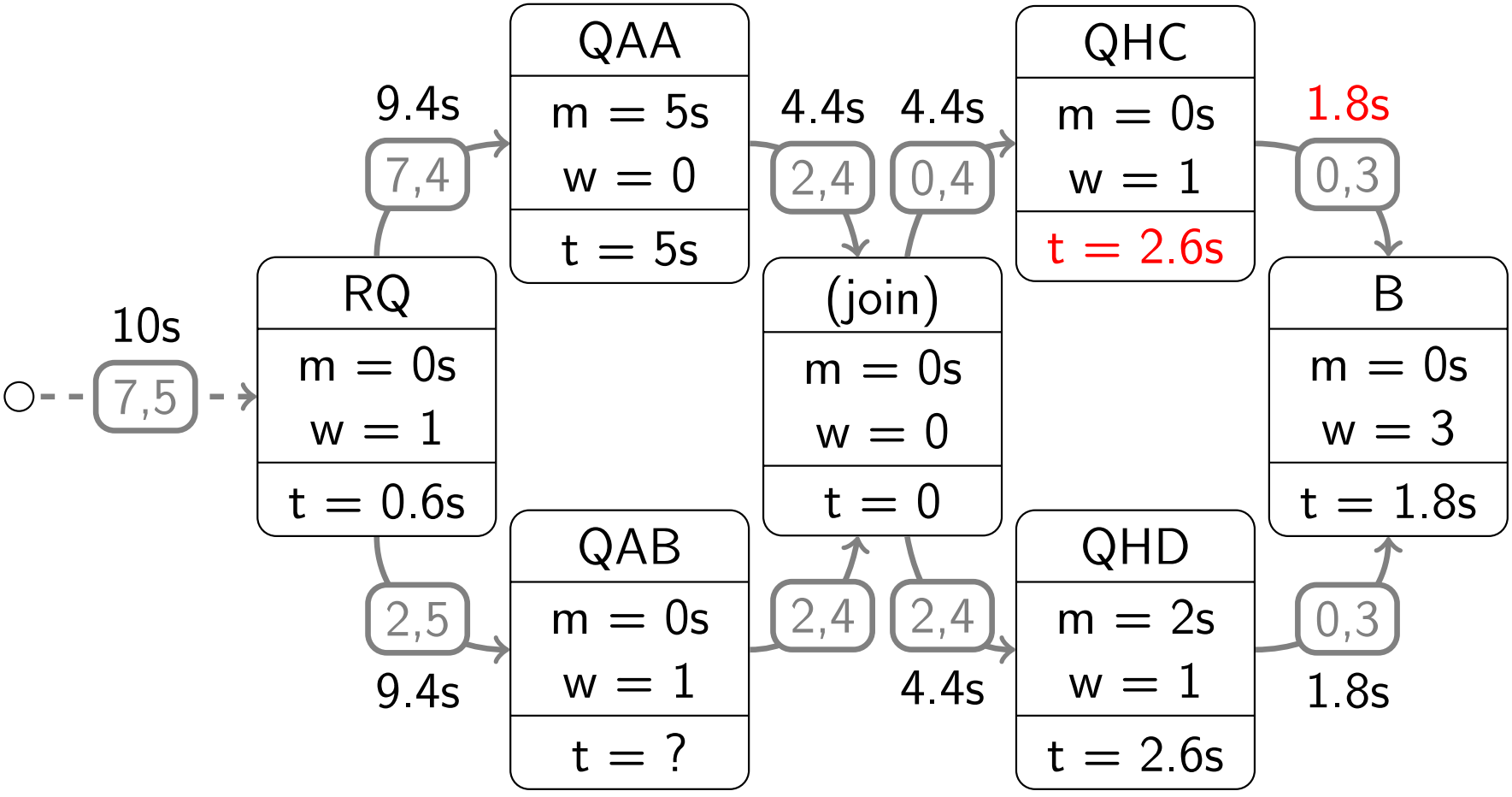

(2,4) is stricter than (0,4): we move to QHD. QHD takes $2 + (4.4 - 2)1/4 = 2.6s$ and sends the remaining 1.8s to B.

# Incremental algorithm: execution (limit = 10 seconds)



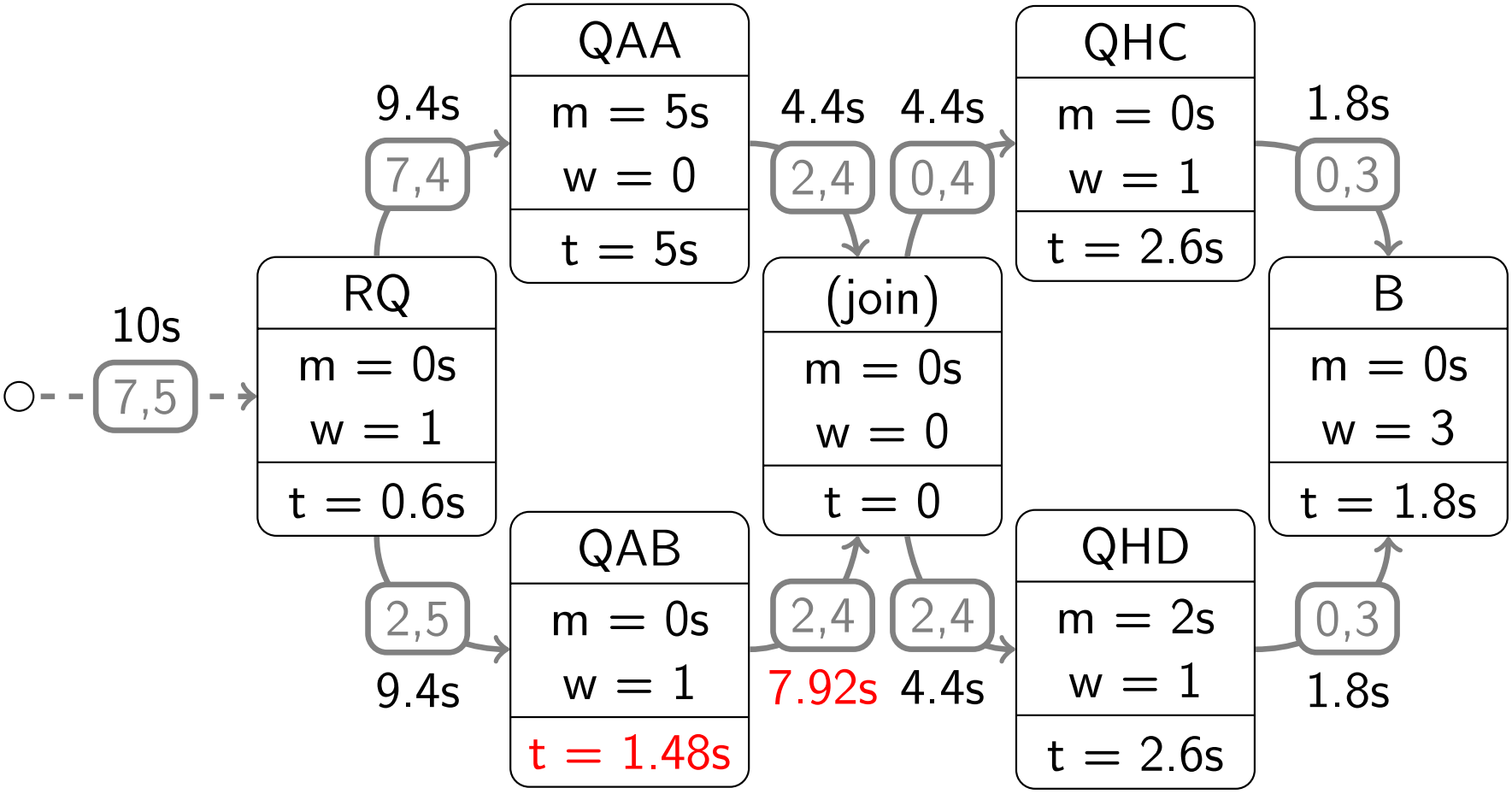

We go back to the join node and then to QHC. QHC initially takes $0 + (4.4 - 0)1/4 = 1.1s$ and tries to send 3.3s to B.

# Incremental algorithm: execution (limit = 10 seconds)



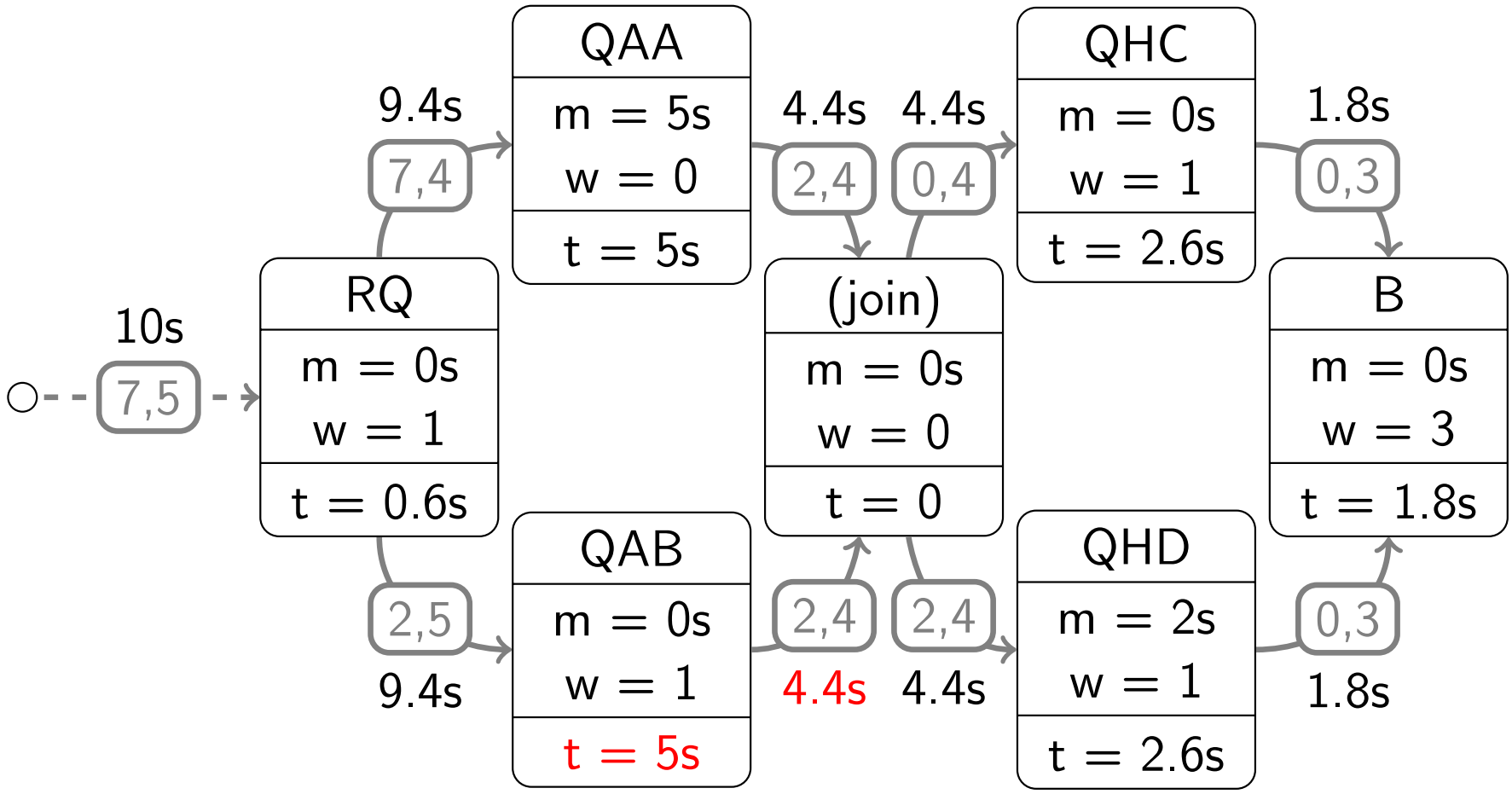

However, we previously sent 1.8s to B. We will use the surplus $3.3 - 1.8 = 1.5s$ on QHC, increasing its time limit to 2.6s.

# Incremental algorithm: execution (limit = 10 seconds)




We go back to RQ and then to QAB. QAB takes $0 + (9.4 - 2)1/5 = 1.48s$ and sends 7.92s to the join node.

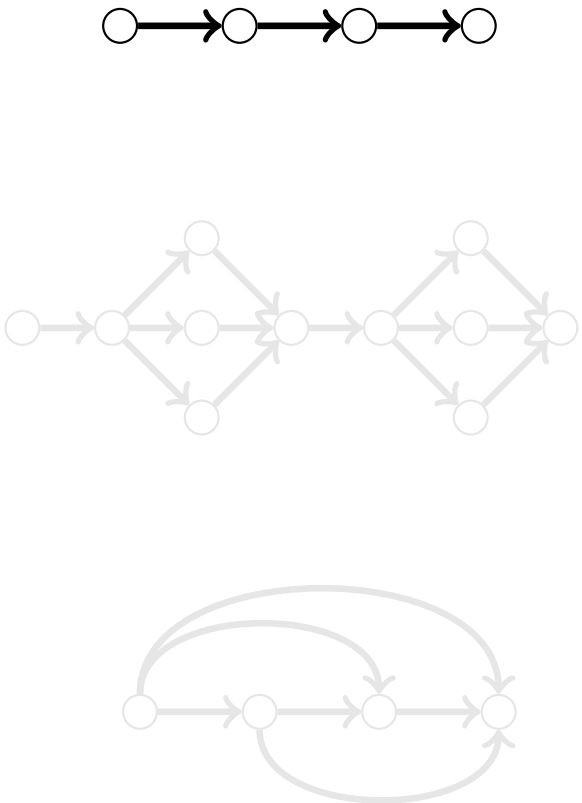
# Incremental algorithm: execution (limit = 10 seconds)

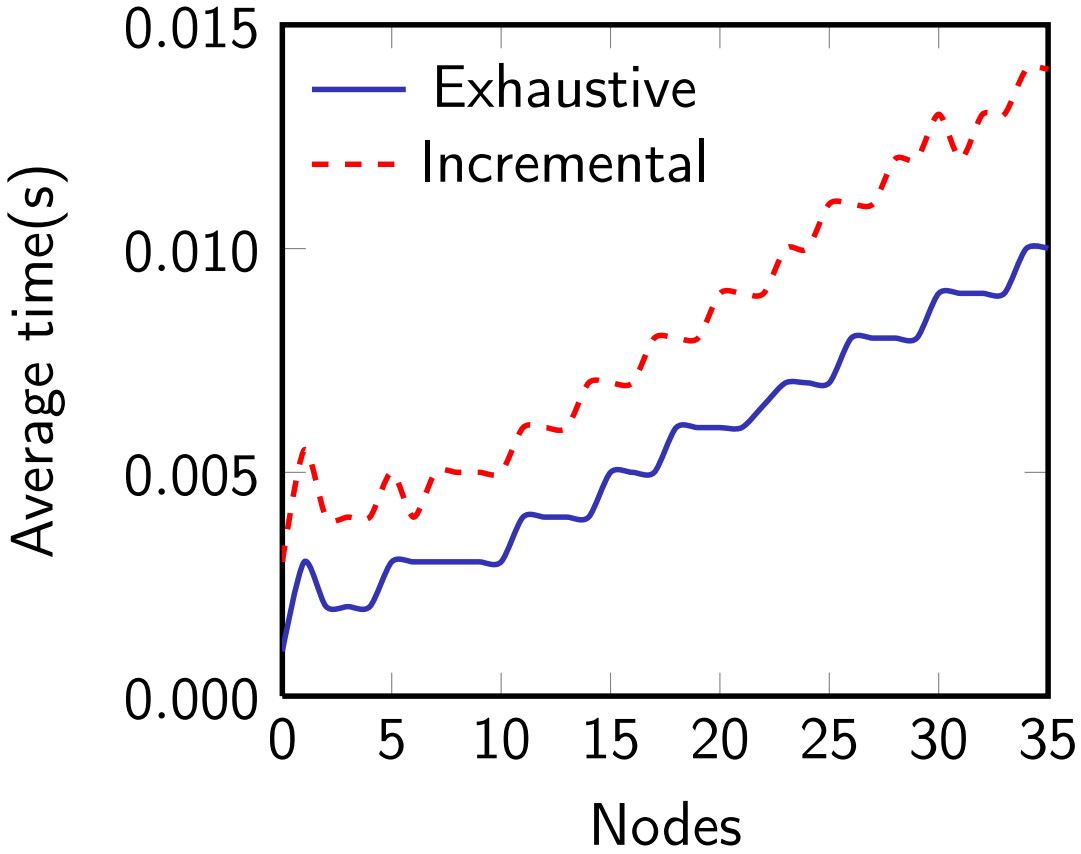



Since we previously sent $4.4s$ to the join node, we reuse the surplus $7.92 - 4.4 = 3.52s$ on QAB, increasing its time limit to 5s.

# Incremental algorithm: performance comparison

New algorithm offers better average-case performance, with some overhead

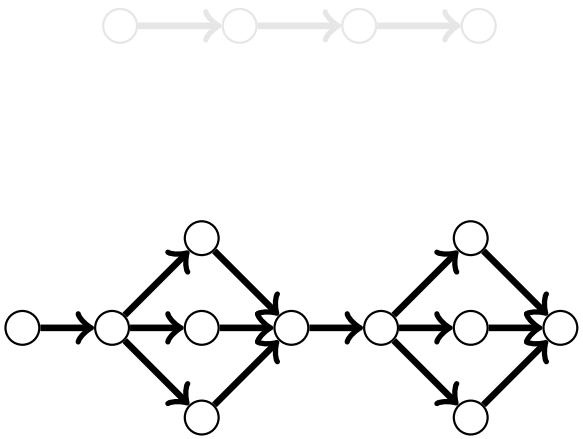




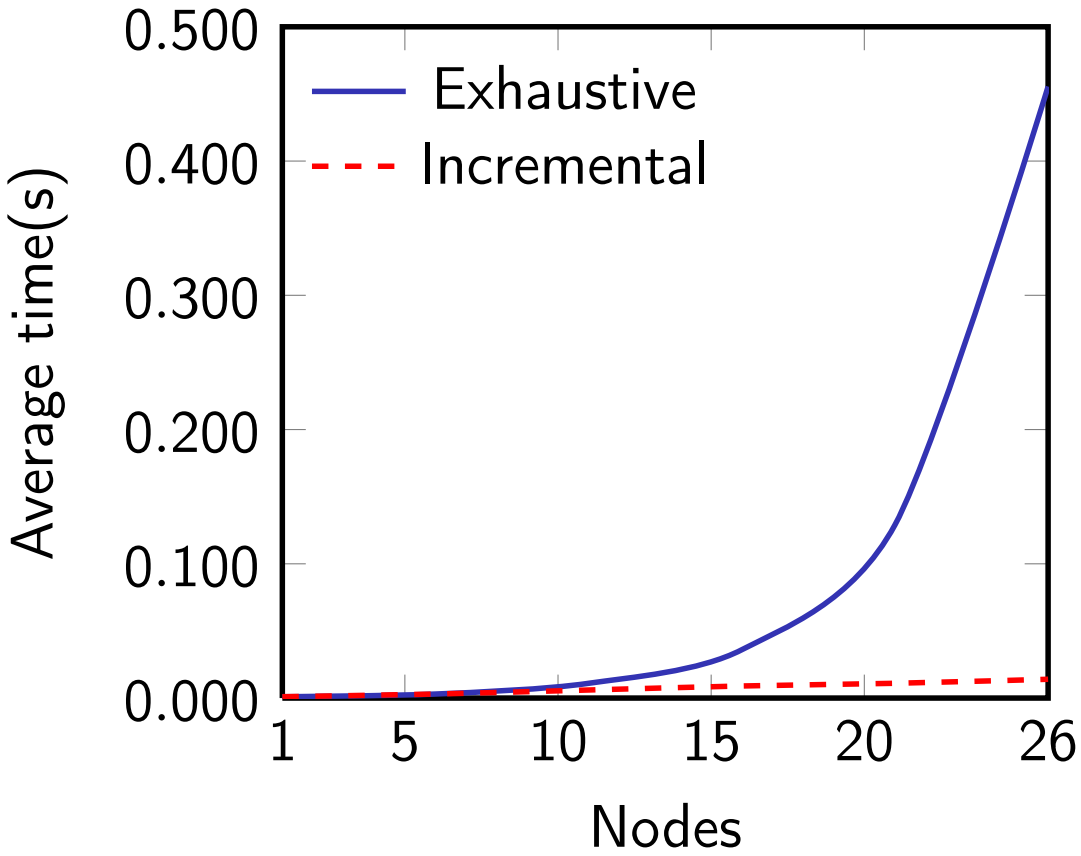

With 1 path, the incremental algorithm is a bit more expensive.

# Incremental algorithm: performance comparison

New algorithm offers better average-case performance, with some overhead





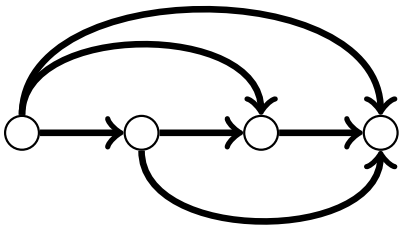


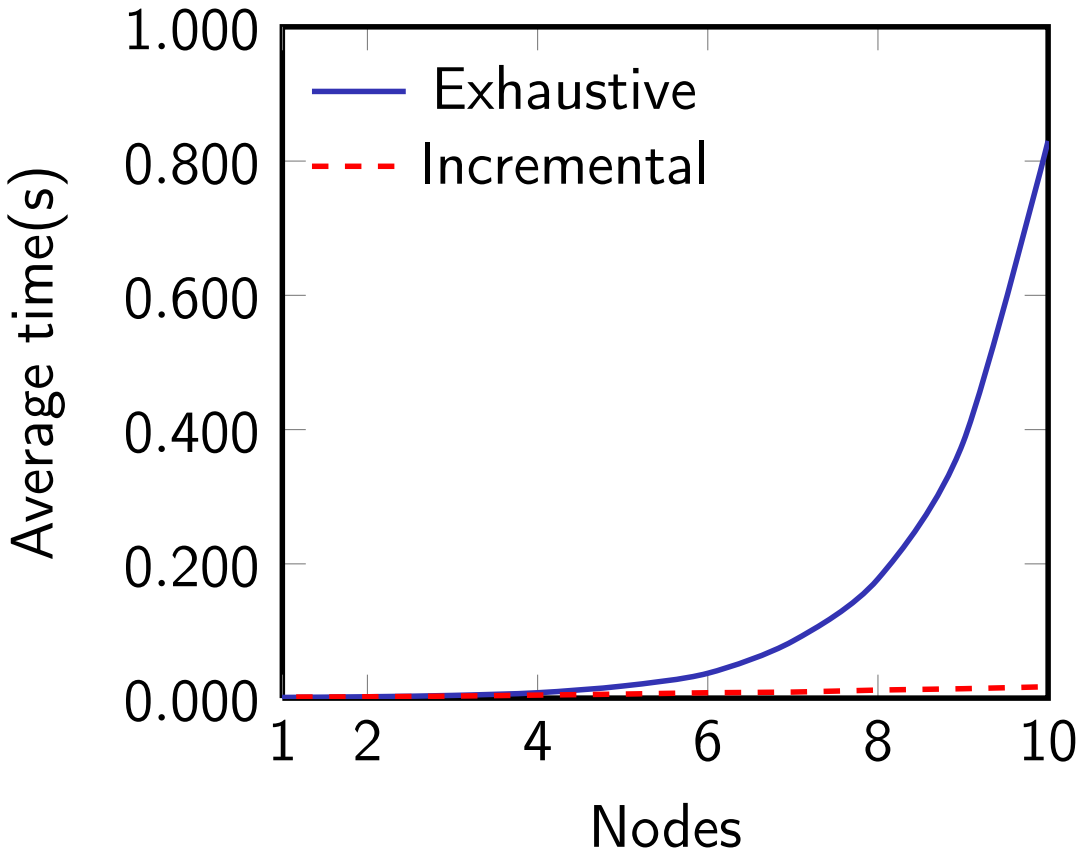

The incremental algorithm is much faster for "fork-join" graphs.

# Incremental algorithm: performance comparison

New algorithm offers better average-case performance, with some overhead





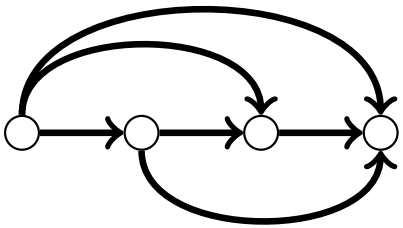


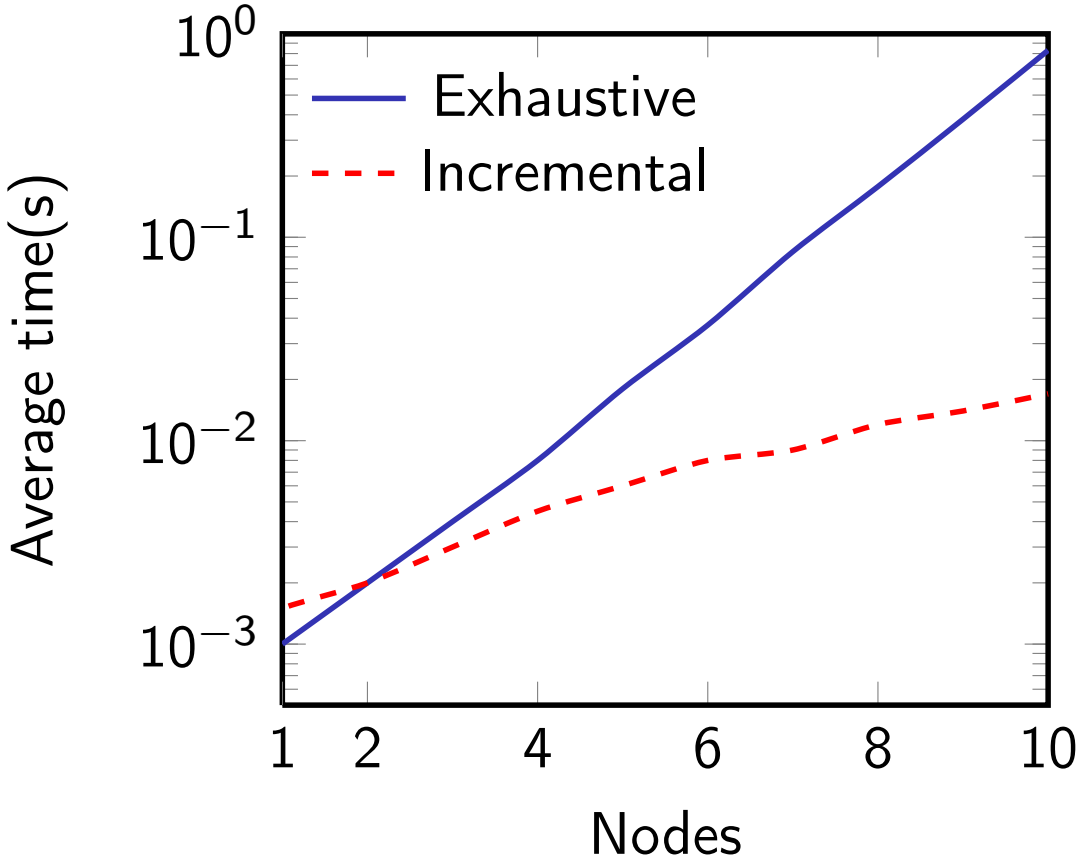

The incremental algorithm handles “dense” graphs much better.

# Incremental algorithm: performance comparison

New algorithm offers better average-case performance, with some overhead








This is more evident if we use a log scale in the $y$ axis.

# Conclusions and future work

## Results

- We presented two algorithms to infer time limits in workflows
- Both work on any DAG with 1 source and 1+ sinks
- Incremental algorithm: much better average-case performance
- Both algorithms share several limitations, e.g.:
  - Cannot handle cycles: repetitions are modeled with weights
  - Every instance of a task is seen as a different entity

## Future work

- ✓ Study worst case of the incremental algorithm (ICSOFT '11)
- ✓ Adapt notation to OMG MARTE (ICSOFT '11)
- ➠ Generate test cases (JUnitPerf, Apache JMeter)

# End of the presentation

## Thank you for your attention

Videos and source code:
`https://neptuno.uca.es/redmine/projects/sodmt`

E-mail:
antonio.garciadominguez@uca.es

Twitter:
`@antoniogado`