Distributed Garbage Collection for General Graphs

Steven R. Brandt, Hari Krishnan, Costas Busch, Gokarna Sharma

Basic Approaches to Garbage Collection

- Tracing Collectors
  - Collect Cycles
  - May not free objects quickly
- Reference Counting
  - Can’t collect cycles
  - Frees objects quickly
- Hybrids
  - Trace when decrementing and the ref count does not become zero

The Brownbridge Collector

- Two kinds of edge, strong and weak
- Two invariants
  - Strong edges connect all the nodes
  - Strong edges contain no cycles
- Trace (i.e. do work) when decrementing the **strong** ref count gives zero, but the **weak** doesn't.
- Original algorithm collected prematurely. Efforts to fix it failed in various ways.

SWP Collector

- Our ISMM 2014 Paper
- An additional kind of edge called "Phantom" was introduced. It represents an indeterminate and temporary state, neither strong nor weak.
- SWP stands for "Strong, Weak, Phantom."
- Nodes that lost their last strong edge...
  - Convert incoming weak edges to strong
  - Phantomize outgoing edges
  - Phantomization may spread if it causes other nodes to lose their last strong edge
- If a subgraph contains only phantom edges, it is garbage.

Distributed Garbage Collection, The Problem...

- High-end calculations spanning hundreds or thousands of nodes are commonplace
- New codes have adaptive meshes and multi-physics
- To meet this challenge, new frameworks use Asynchronous Multi-Tasking (AMT) - reliable but out of order messages, e.g.
    - HPX
    - Charm++
    - Uintah
- Almost all C++/C or Fortran with no garbage collection
- All present distributed garbage collectors require global barriers, stop-the-world, centralization, or sweeping all of memory on all computational nodes. Unscalable.
- A significant problem for developers

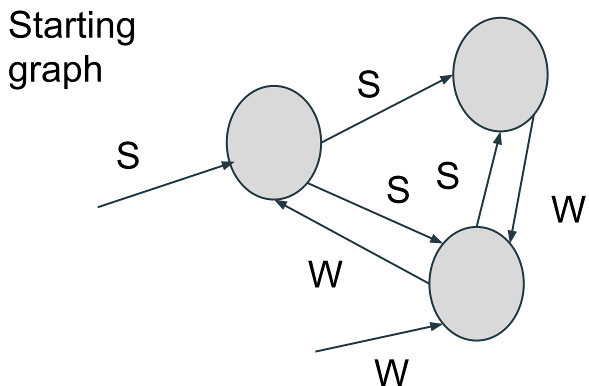
Distributed Garbage Collection: Prior Work

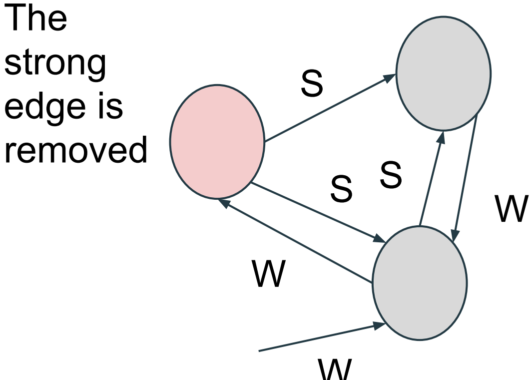
- Laden and Liskov 2015 used a centralized data store
- Tel 1993 and Blackburn 2001: There is a strong relation between distributed termination detection (DTD) and garbage collection, and a DTD algorithm can be derived from any distributed garbage collector. Examples are all, in some fashion, based on mark and sweep.
- Liskov 1995 used migrating objects to move the garbage cycle to a single machine.
- Bevan 1987 used distributed reference counting, but could not claim cycles (used by HPX currently)

Introducing the **SWPR** Collection Algorithm

- Decentralized, works on isolated areas of the graph independently
- No dedicated collector threads
- Linear time complexity, collection is O(N)
- No stop-the-world
- Built-in generational effect
- Minimal tracing
- Actor Model, only one object synchronized at a time.
- Message size is log(N) where N is the size of the graph.
- Overhead per node is log(N) where N is the size of the graph.

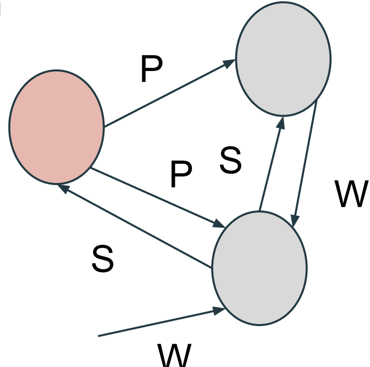
SWP Collector, an Example



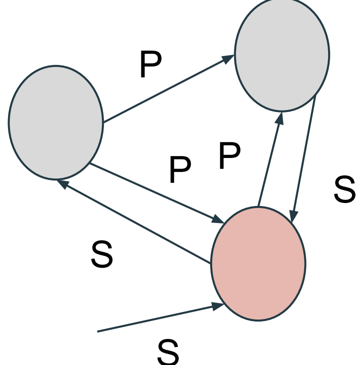


SWP Collector, an Example

Phantomization occurs

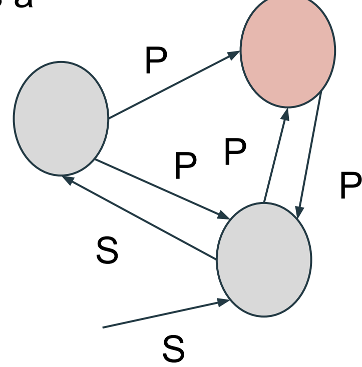


Phantomization spreads

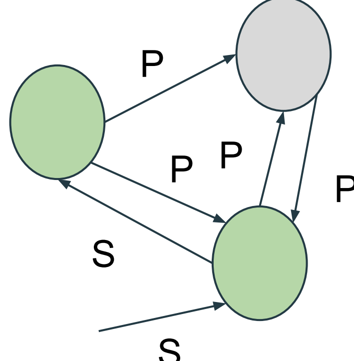


SWP Collector, an Example

… and spreads a final time

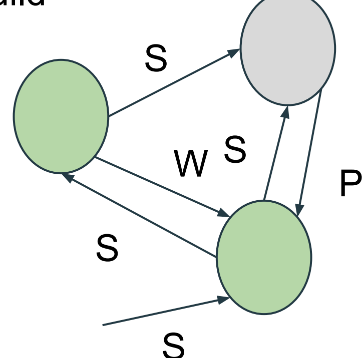


These nodes have strong edges at the end



SWP Collector, an Example

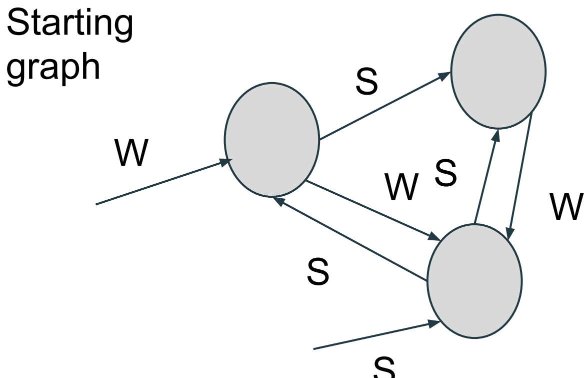
And so we rebuild their outgoing edges

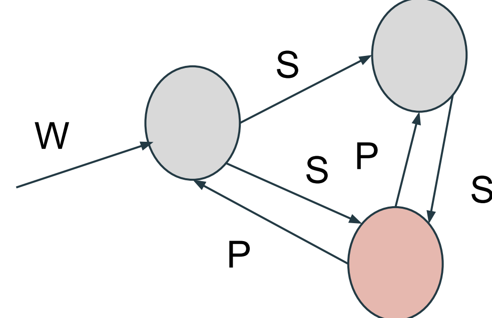


Rebuilding propagates, too

And we're done!

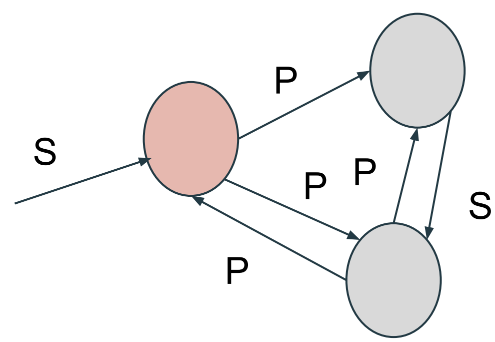
SWP Collector: A Similar Example



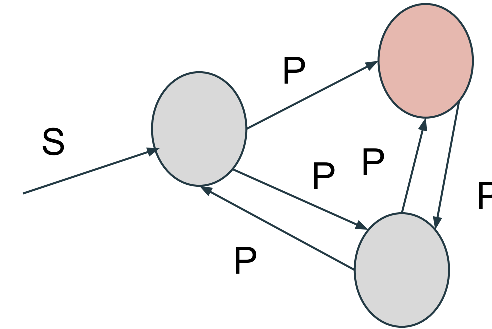


As before, when we pull a strong edge away, the node toggles and phantomizes.

SWP Collector: A Similar Example

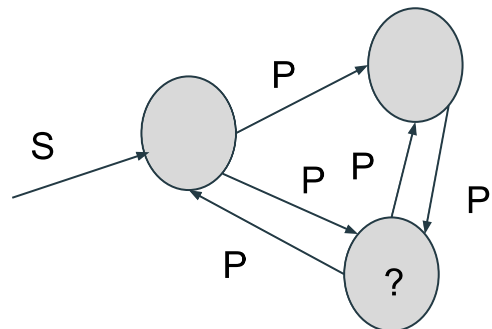


Phantomization spreads, causing another node to toggle.

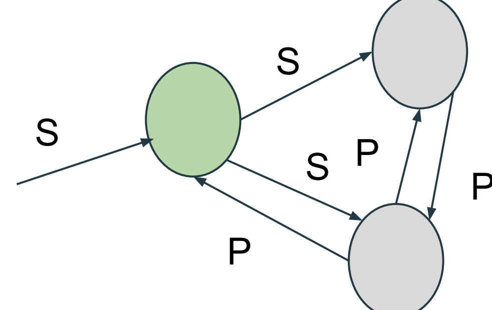


Phantomization spreads to the last node. All nodes phantomized. The initial node has all incoming phantom edges.

SWP Collector: A Similar Example

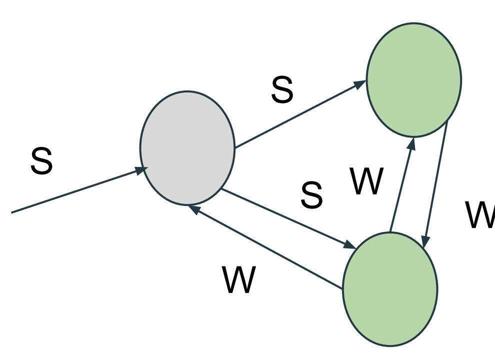


We cannot rebuild the graph from the initial node.



We can, however, “recover” from one of the other nodes in the graph because it has a strong edge.

SWP Collector: A Similar Example

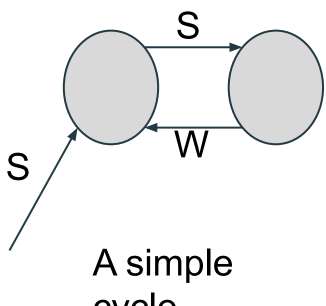


The recovery spreads, and all the edges of the graph are rebuilt.

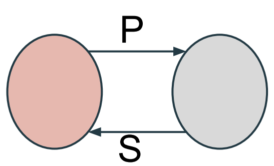
The Phases of Collection:

- Phantomization
- Build - If the initiator has a strong edge
- Recover - If the initiator does not have a strong edge. Recover can lead to building.
- Delete - If Recover fails.

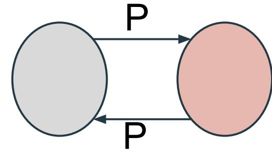
SWP Collector, Collecting a Simple Cycle



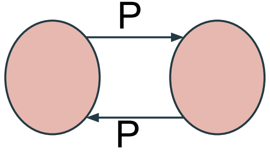
A simple cycle



Remove the incoming edge, Convert incoming edge and phantomize.



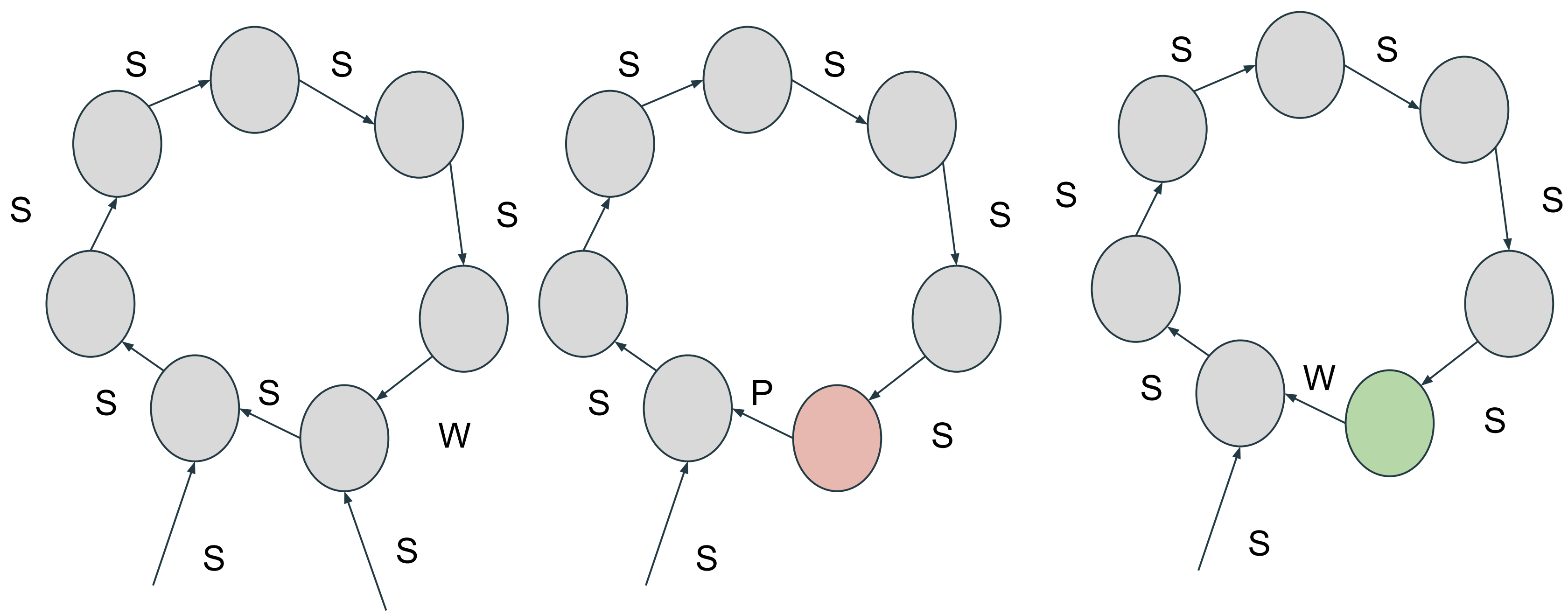
Phantomization spreads



Everything is phantomized. Recovery fails. The cycle is garbage.

SWP Collector: One Last Example

This kind of graph stabilizes quickly!



The Multi-Collection Algorithm

- More complex
  - An additional counter, the R
  - Rollback of recovery operation is possible
- Collection ID tuple
- Handles deletion of edges during collection
- Handles collections starting in different places and meeting

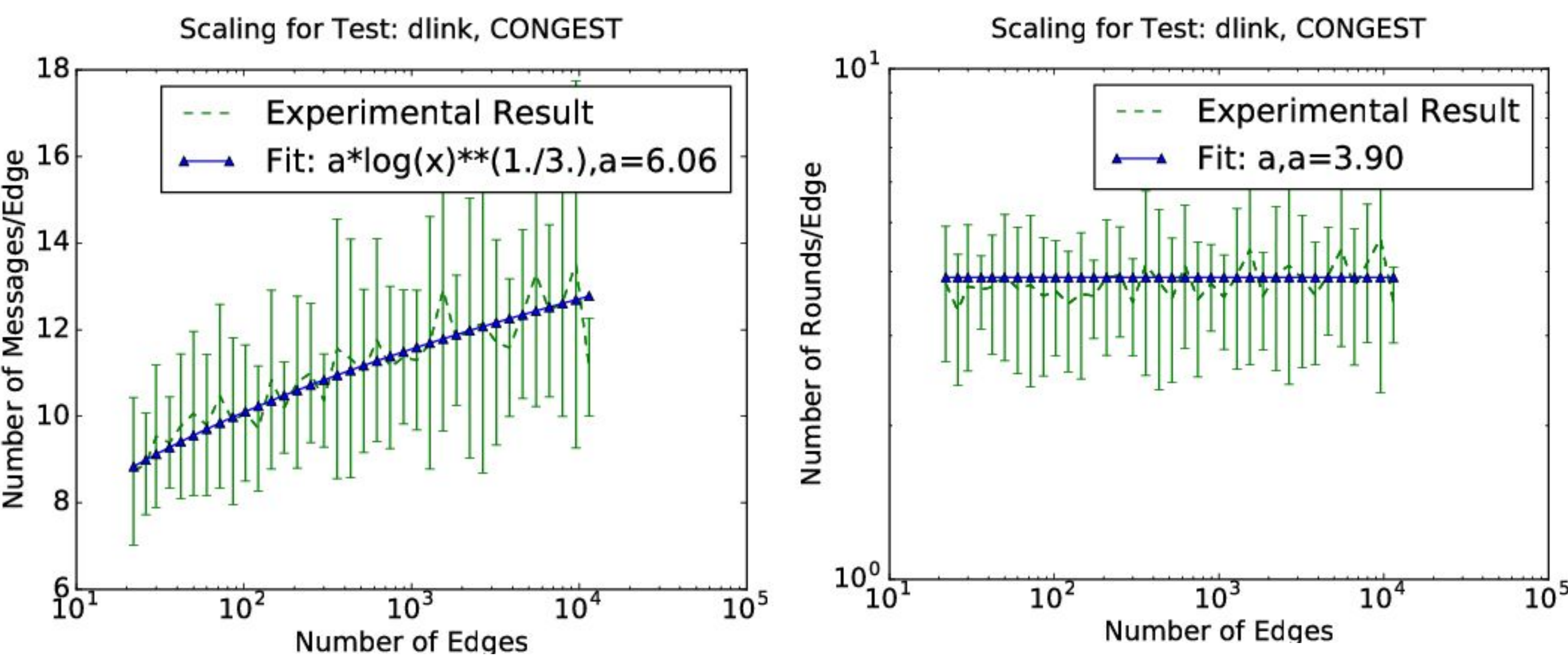
Does it work?

- "Proof" arguments in ISMM18 paper
- Hundreds of thousands of tests
    - Doubly linked lists
    - Cycles
    - Random graphs
    - Grids of nodes
    - Cliques
- Millions of nodes / collections
- Try the Java simulator: https://github.com/stevenrbrandt/DistributedGarbageCollectorSimulator

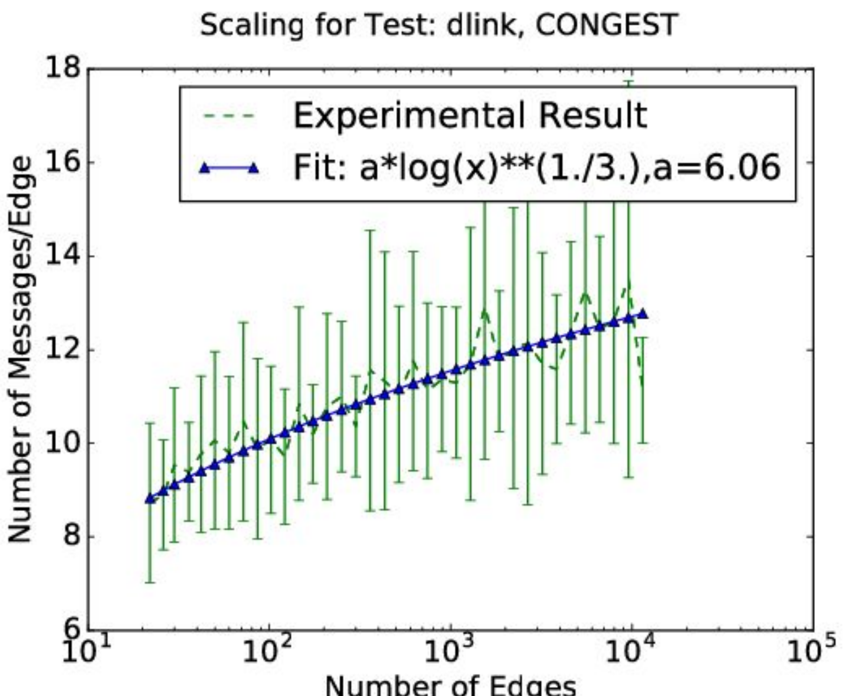
Details

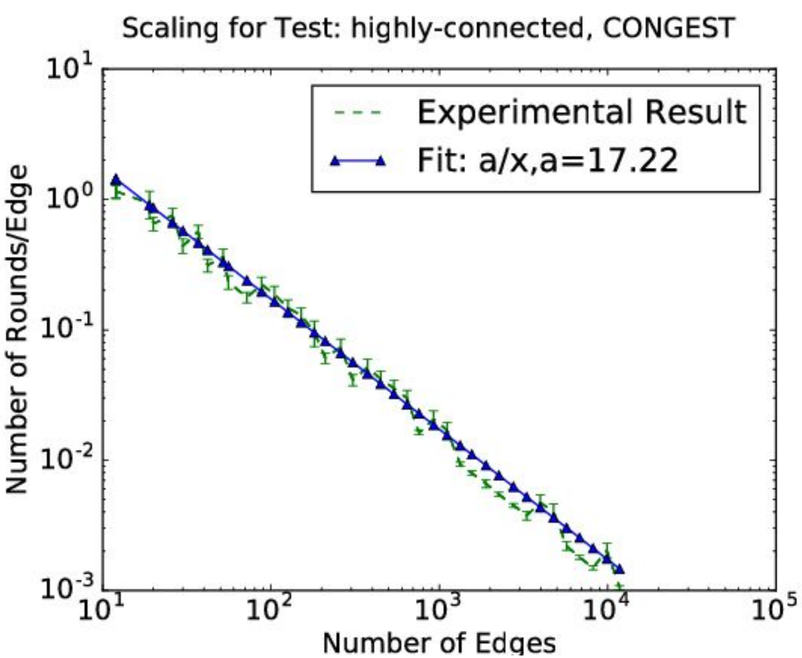
- SWPR uses “weights” to identify strong edges. Edges that go from lower to higher weight are strong, all others are weak.
- Each node tracks a weight (w) and a “max weight” (mw), i.e. the largest weight from any incoming node.
- Roots have a weight of 0.
- “Toggling” now means increasing the weight: $w \rightarrow mw + 1$.
- As before, toggling phantomizes outgoing edges. In SWPR, however, the state of the node is phantomized, not the edges. When the node is phantomized, all the edges are, too.
- Based on Actor Model: Since no nodes have any shared state and only communicate by messages, we can simulate with a sequential code that processes messages in random order.

Performance

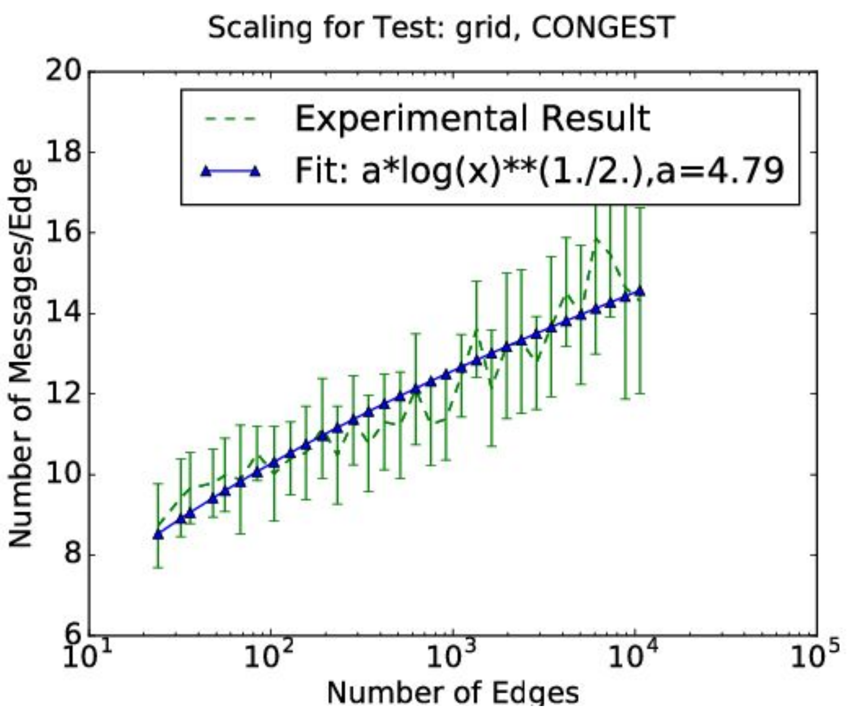


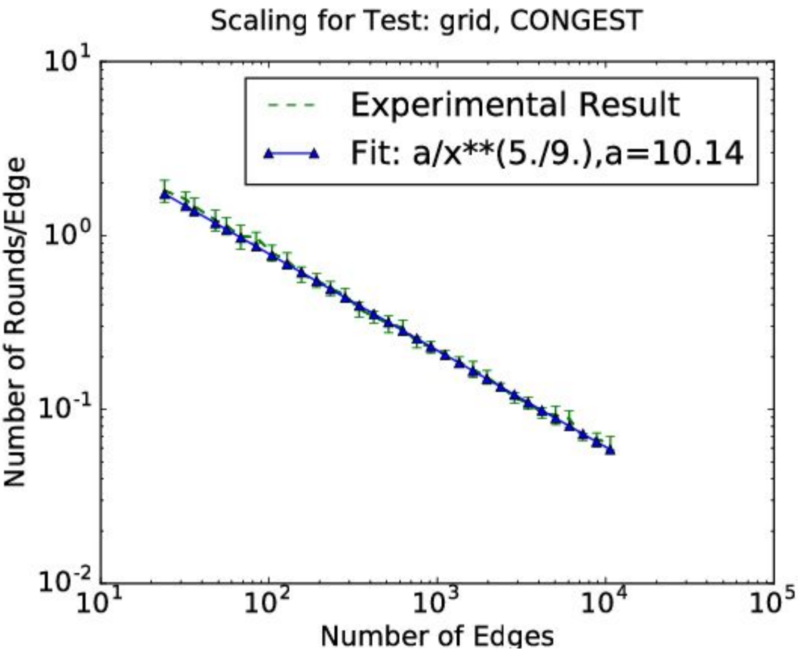
Performance





Performance





Thanks!

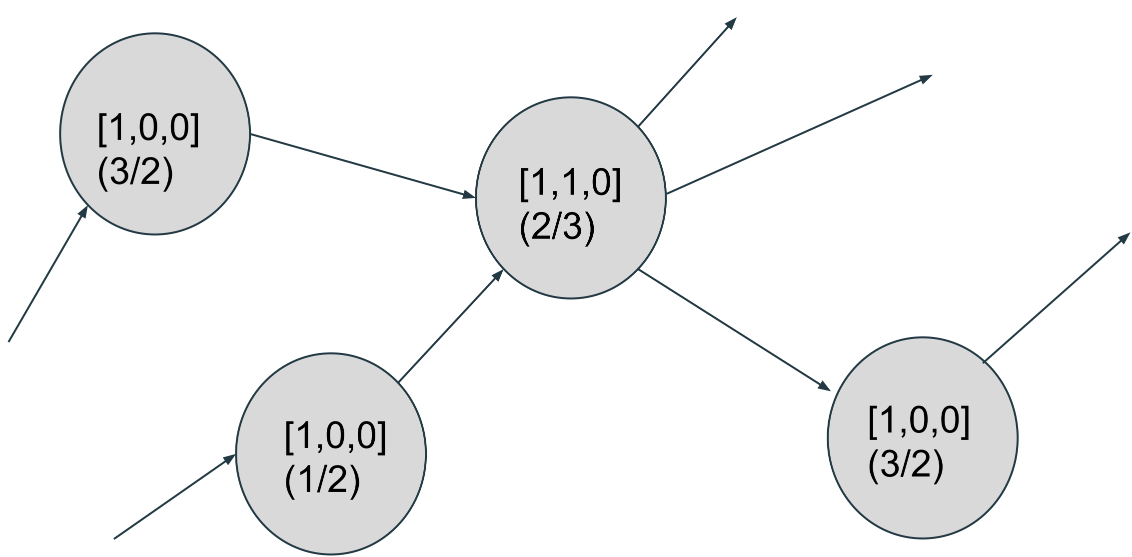
Appendix

The remaining slides describe details in implementing the parallel collector. This is too much for a ½ hour talk, but are provided for reference purpose.

Take a look if there’s time.

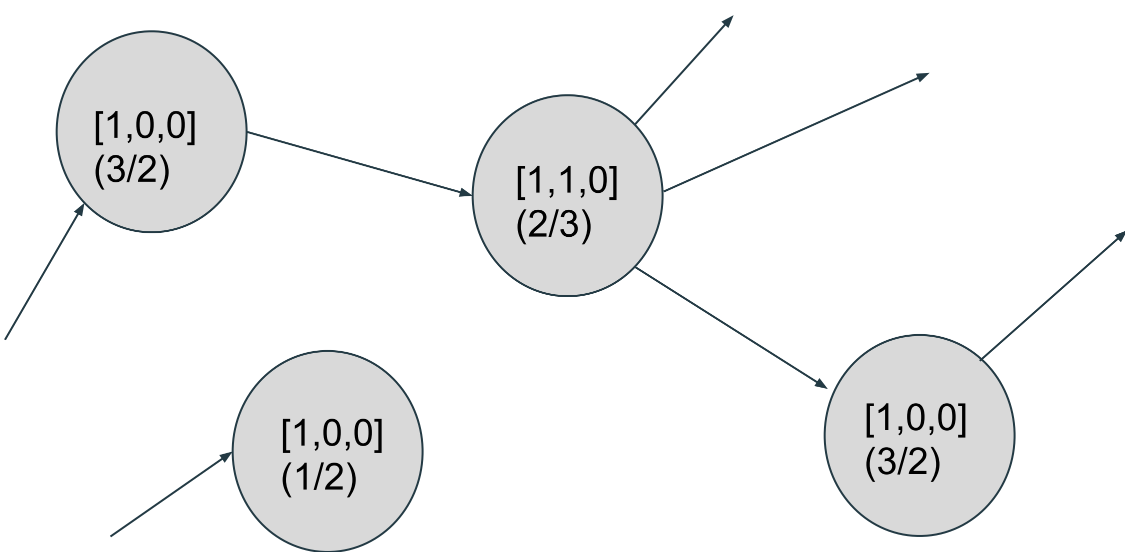
Collecting With SWPR

Reference counts are written like this: [2,1,0], it means strong count=2, weak count=1, phantom count=0. Weights are written like this (2/3), it means weight=2, max weight=3.



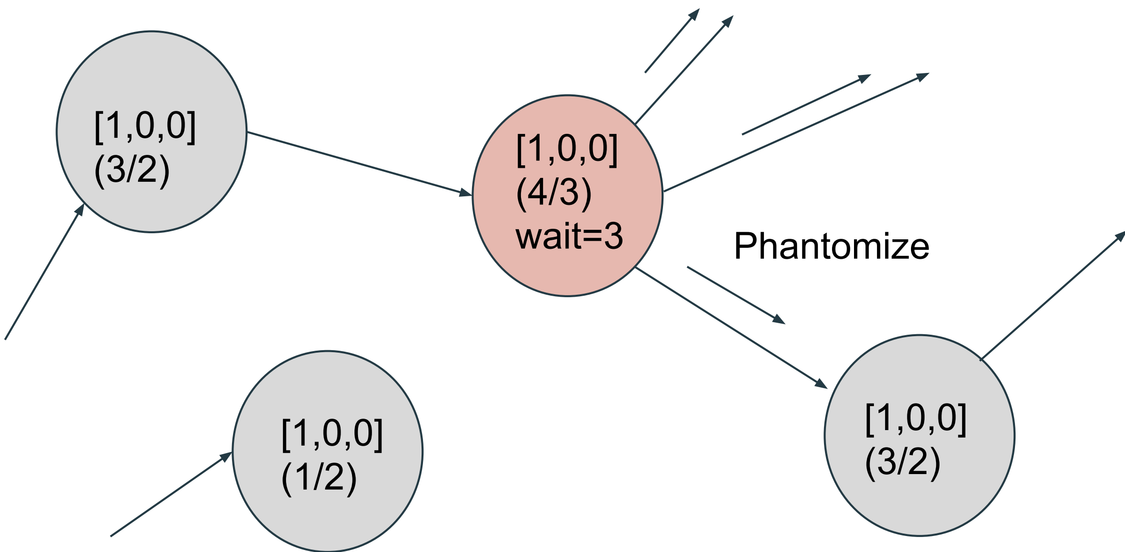
Collecting With SWPR

We now delete one of the edges. The central node now has strong count=0, and weak count=1. This violates our requirement that all nodes have strong support.



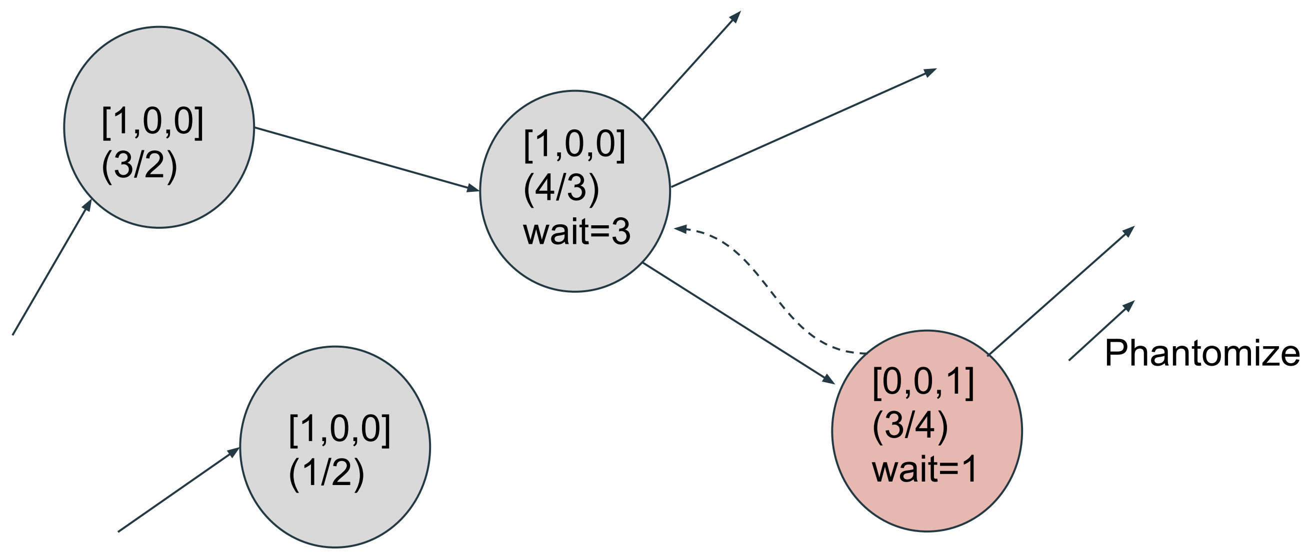
Collecting With SWPR

The central node toggles, increasing its weight so that its weak edges become strong. It now phantomizes, sending phantomize messages along its outgoing edges. It sets its wait count to 3. The node will take no further action until wait=0. This central node is called the “initiator” and it coordintes the collection.



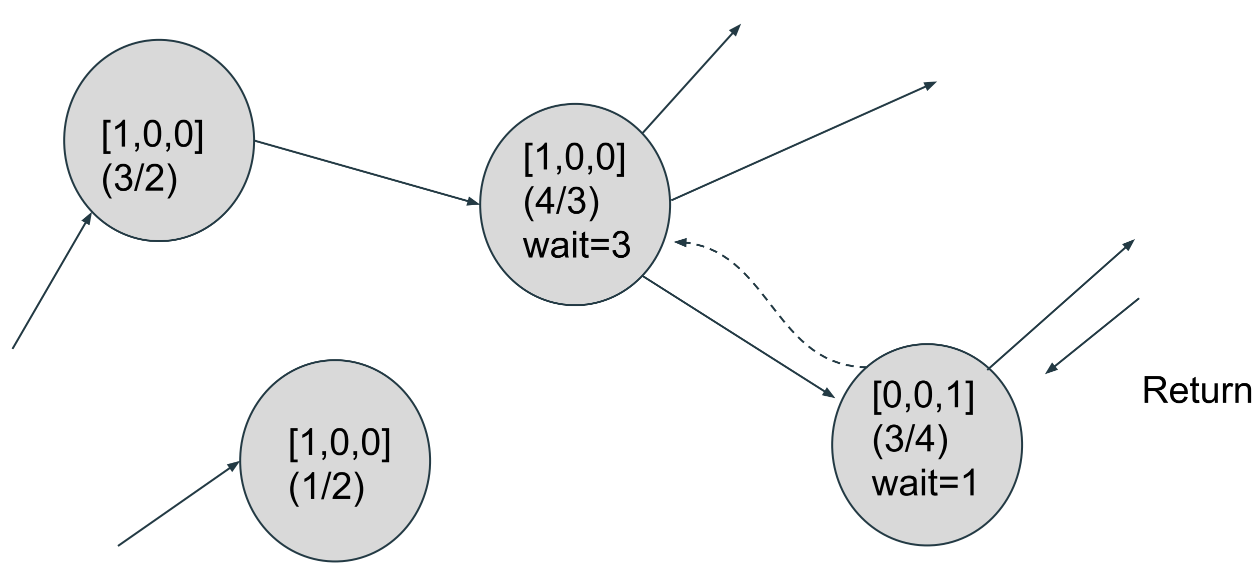
Collecting With SWPR

When phantomize reaches the node on the right, its strong edge is converted to phantom. Because it loses strong support, it, too, phantomizes. Note that it remembers the node it must send Return to after phantomization with a parent edge (dashed arrow). All nodes have storage for a single parent edge.



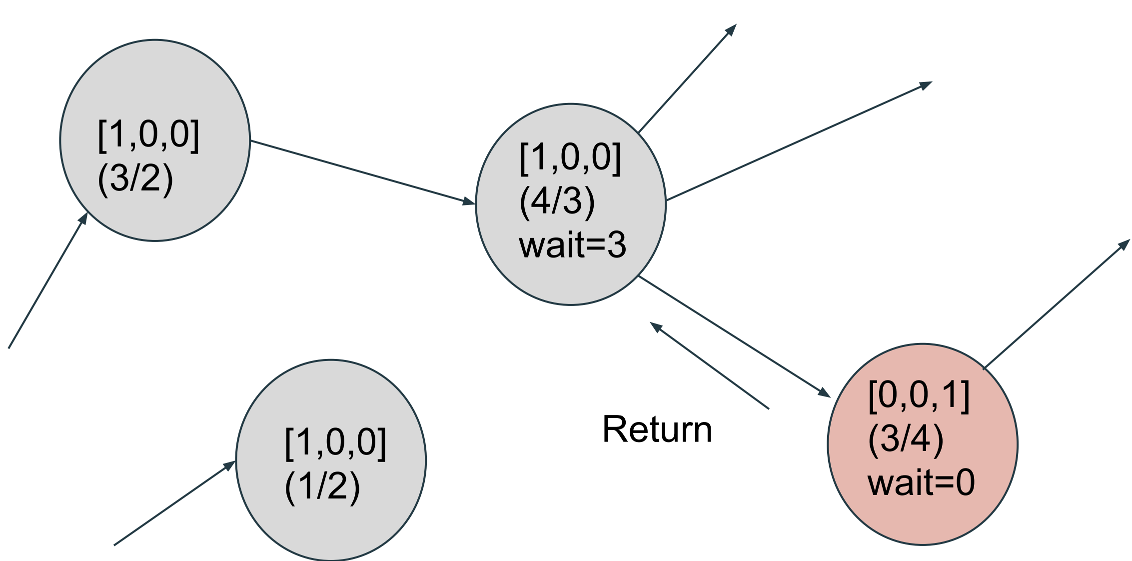
Collecting With SWPR

Eventually, a return message comes back to the node on the right.



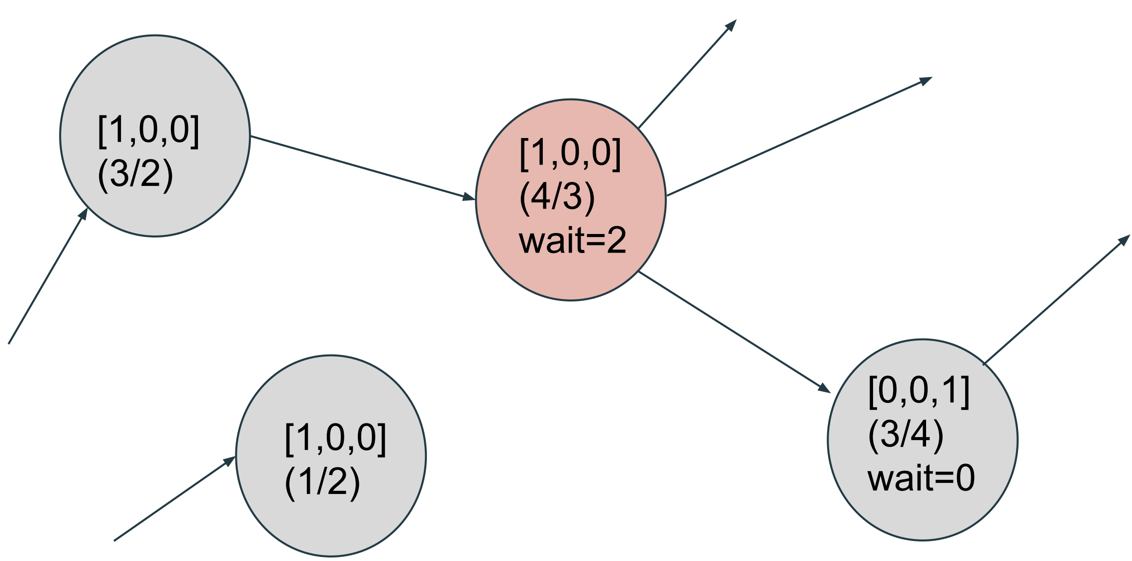
Collecting With SWPR

It's wait count is now zero, so it sends return to its parent and unsets the parent edge.



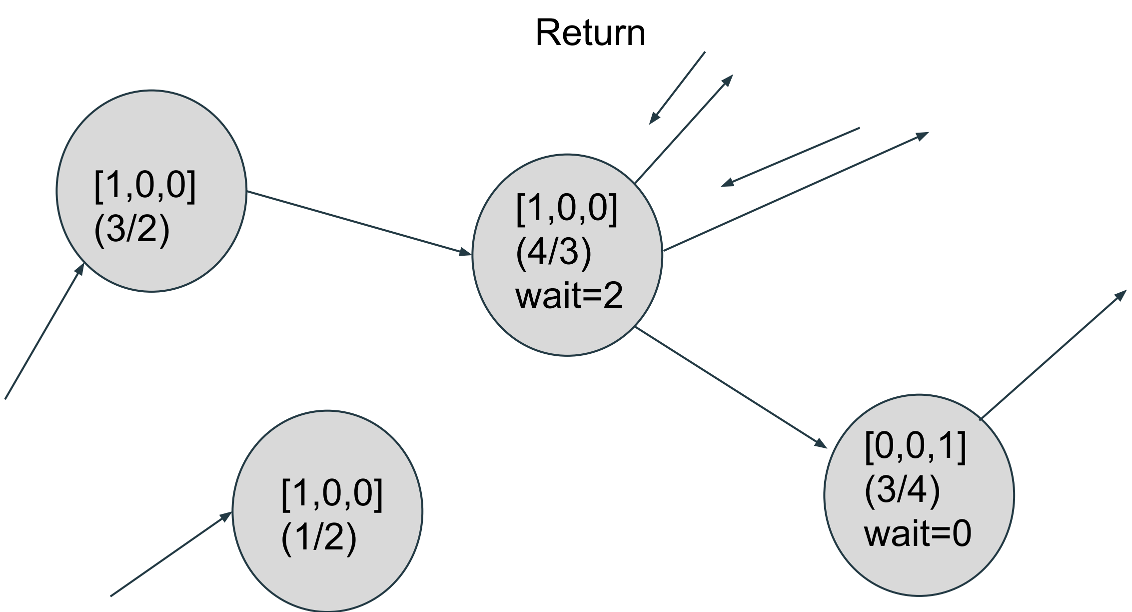
Collecting With SWPR

The wait count on the central node drops to 2. That’s not zero, so it doesn’t do anything.



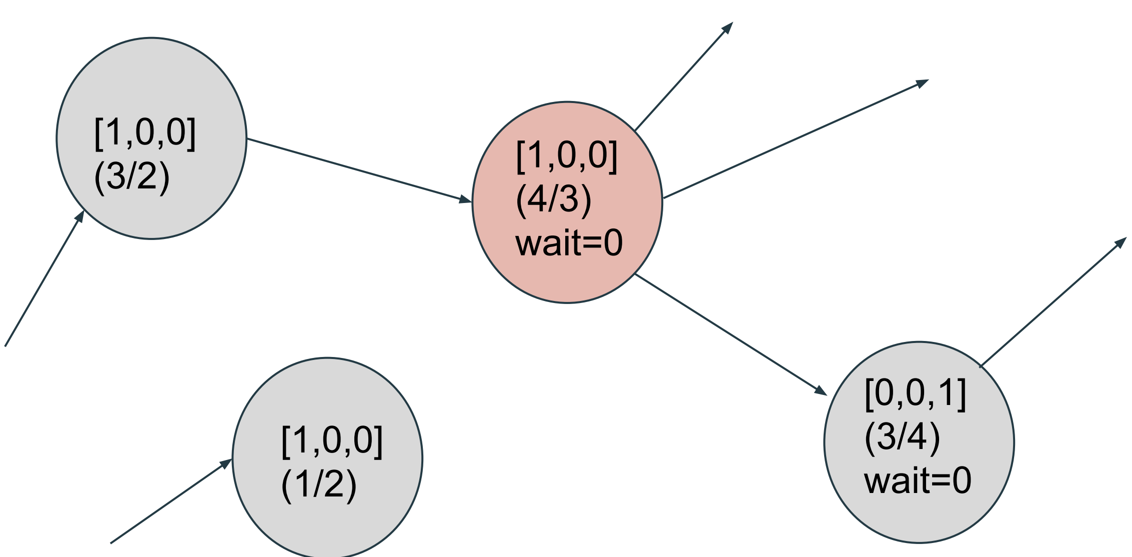
Collecting With SWPR

Finally, the other return messages come back.



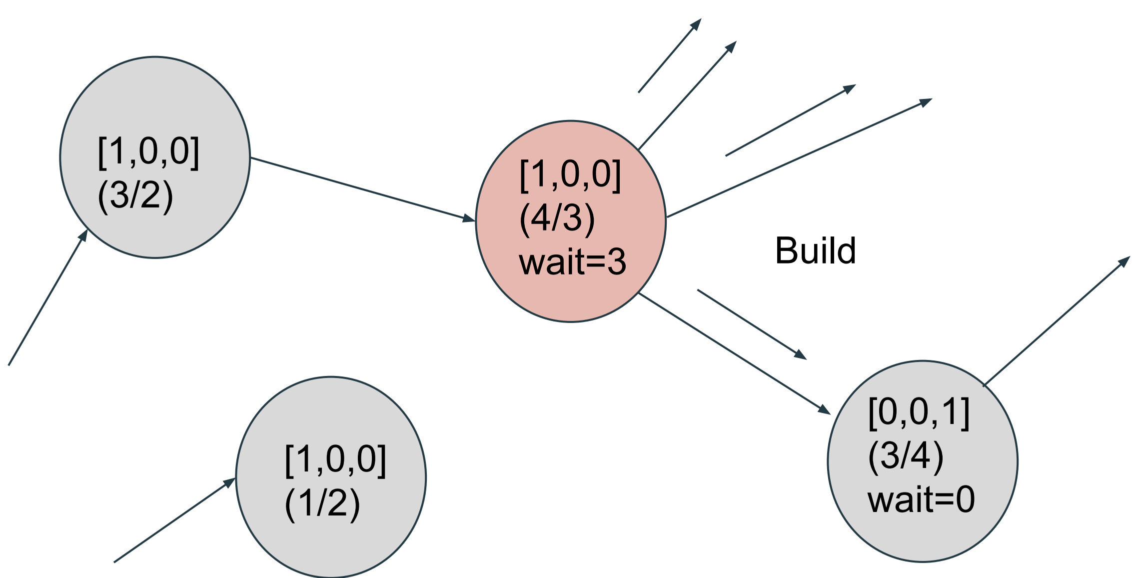
Collecting With SWPR

The wait count is now zero, so an action can be taken...



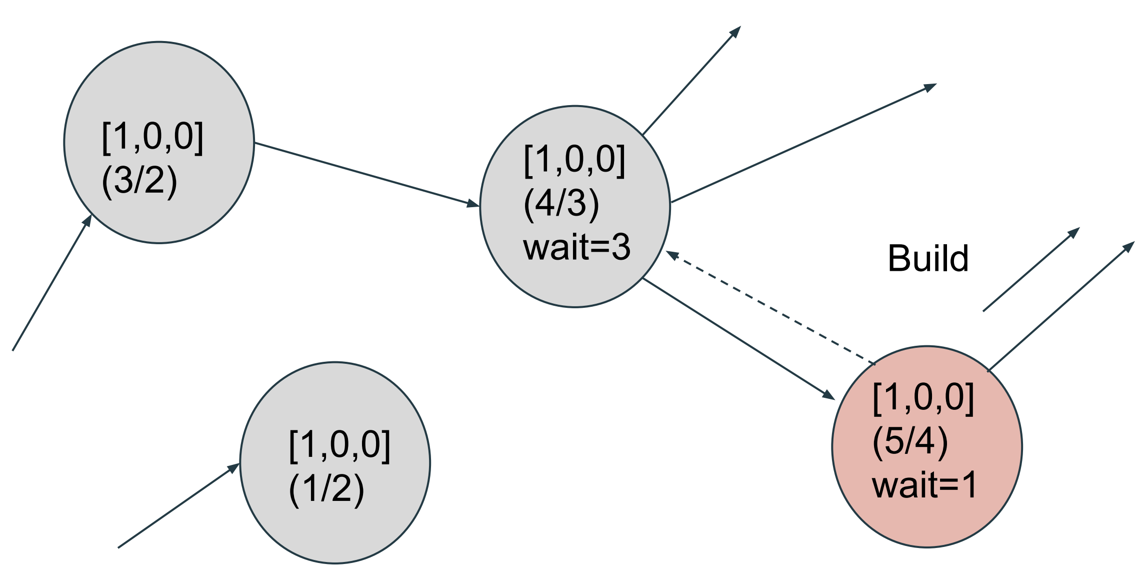
Collecting With SWPR

Because our strong count is positive, the next action is to build, i.e. to clear the phantomized state. Note that this increases the wait count back to 3.



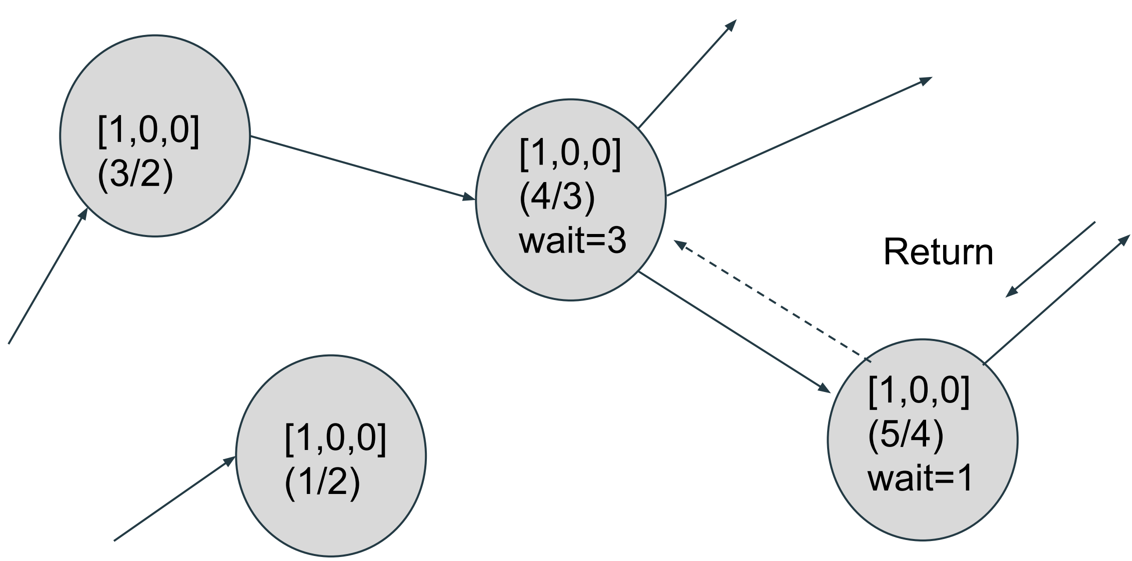
Collecting With SWPR

The node on the right now has a weight of 5 and a max weight of 4. Accordingly, it sets its strong count to 1, its phantom count to zero, and propagates the build message.



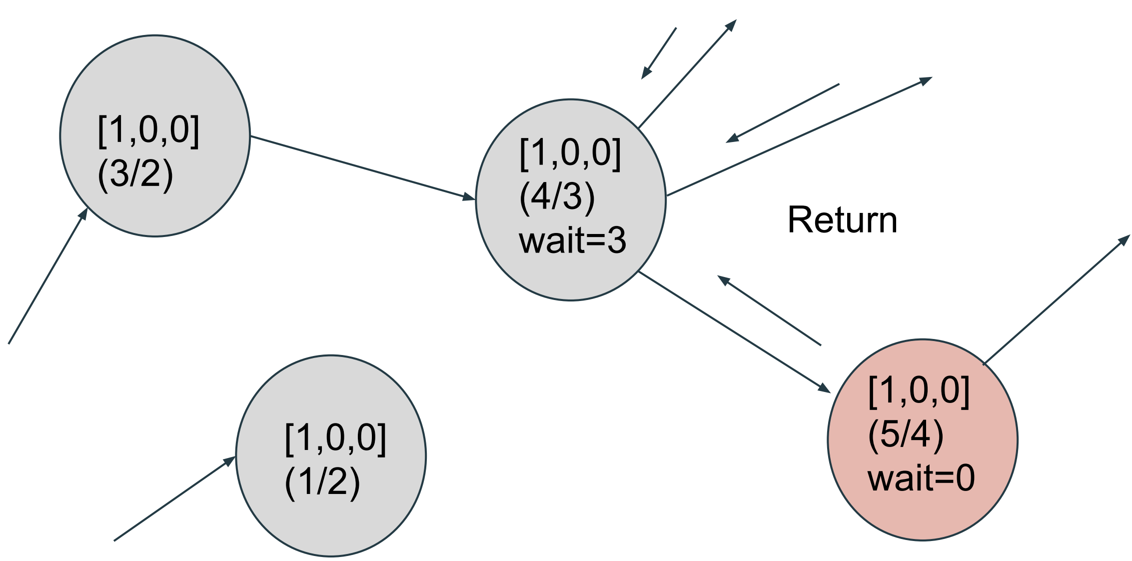
Collecting With SWPR

The build message returns.



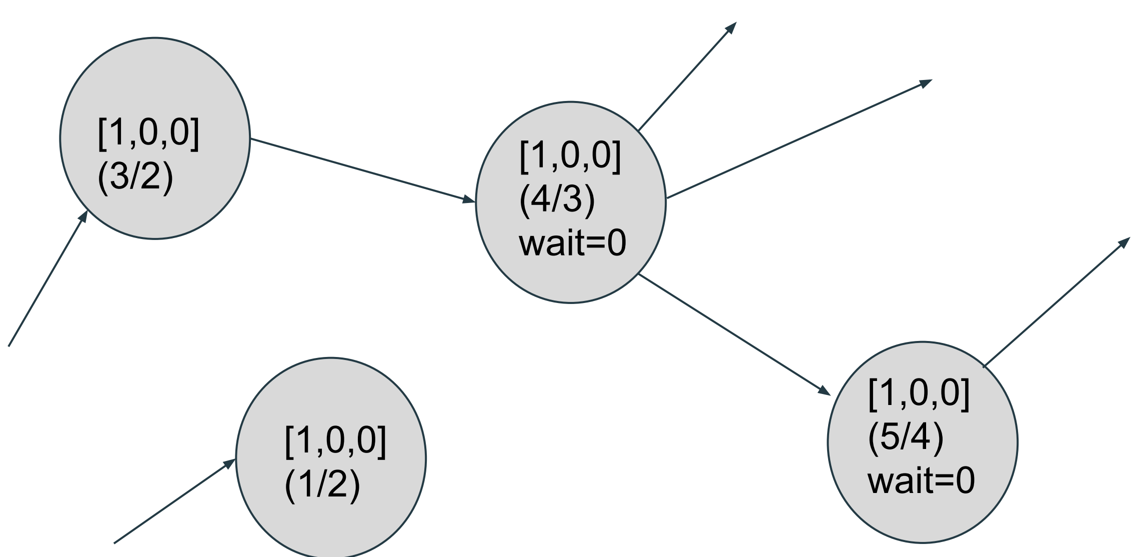
Collecting With SWPR

The node on the right now has a wait of 0, so it sends return. When the central node receives it, its wait will be zero, and the collector will be done.

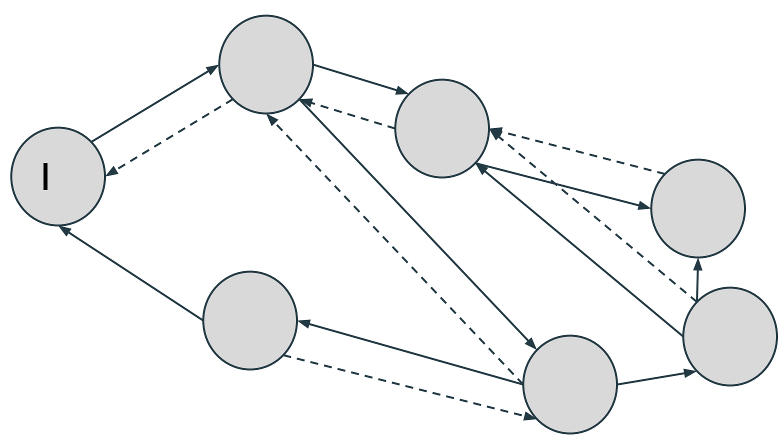


Collecting With SWPR

A quiet state is achieved.



The Phantomization Process

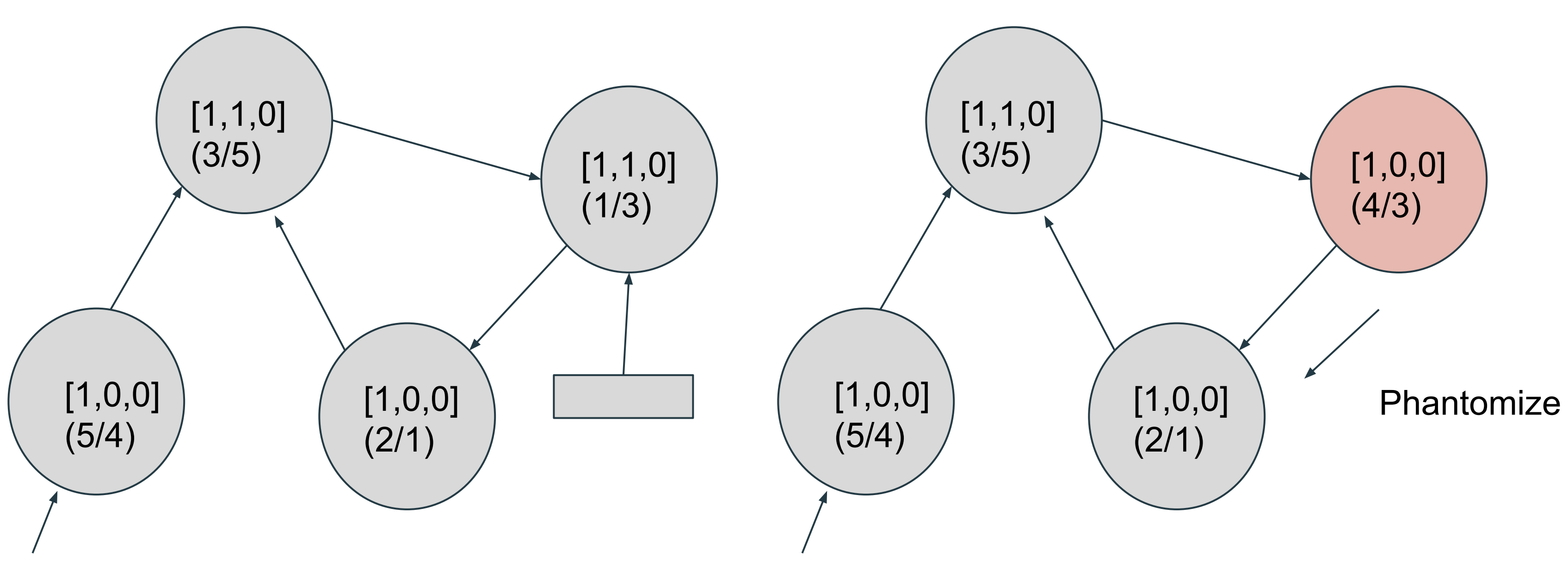


- A possible set of parent edges for a Phantomizing graph is pictured at left.
- The Initiator node (Node I) does not have a parent edge
- There is only one outgoing parent edge per node
- Connects all nodes in the Phantomizing graph with edges back to the initiator
- Parent edge cannot form a cycle

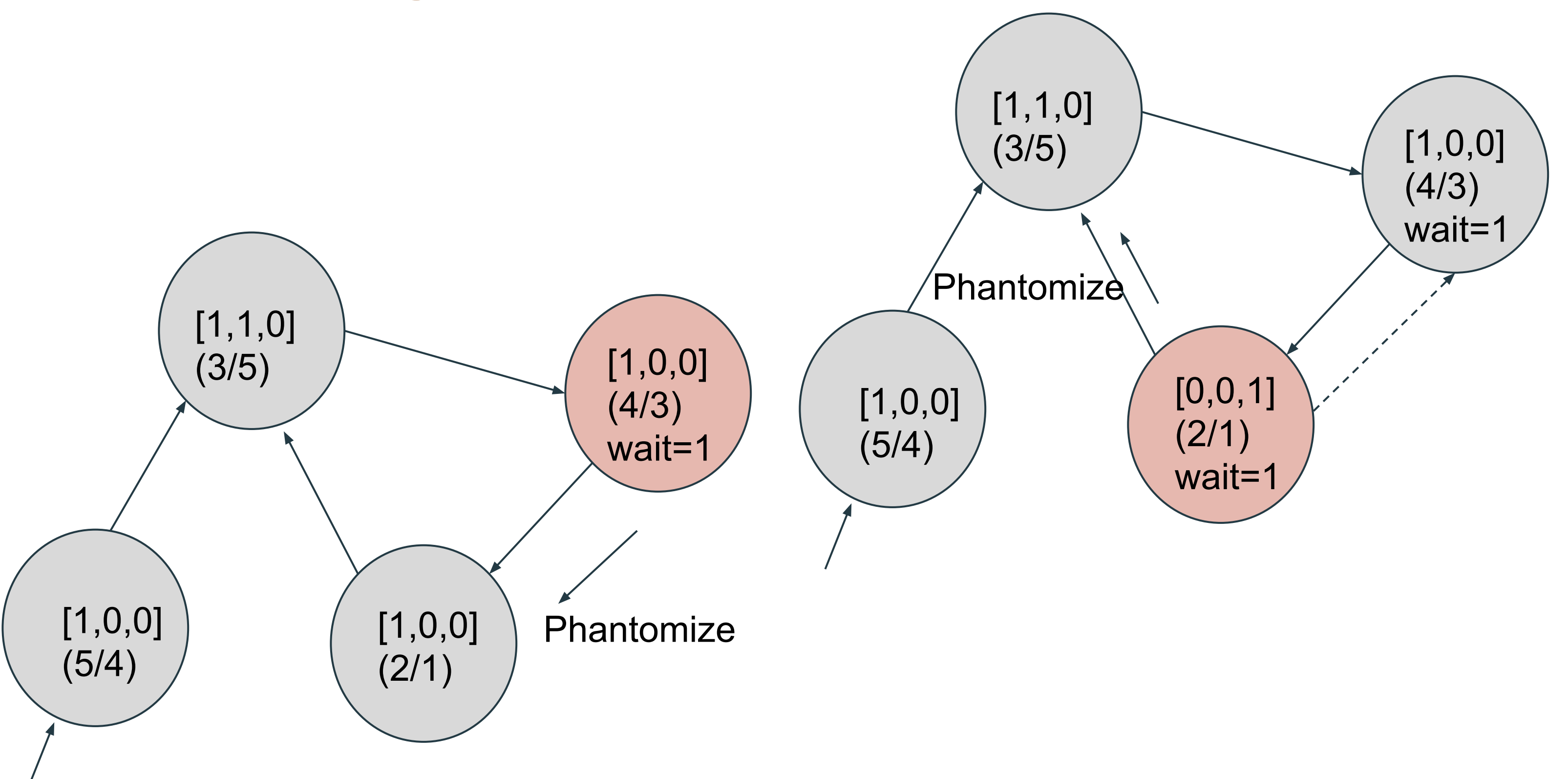
Another Example...

Collecting With SWPR

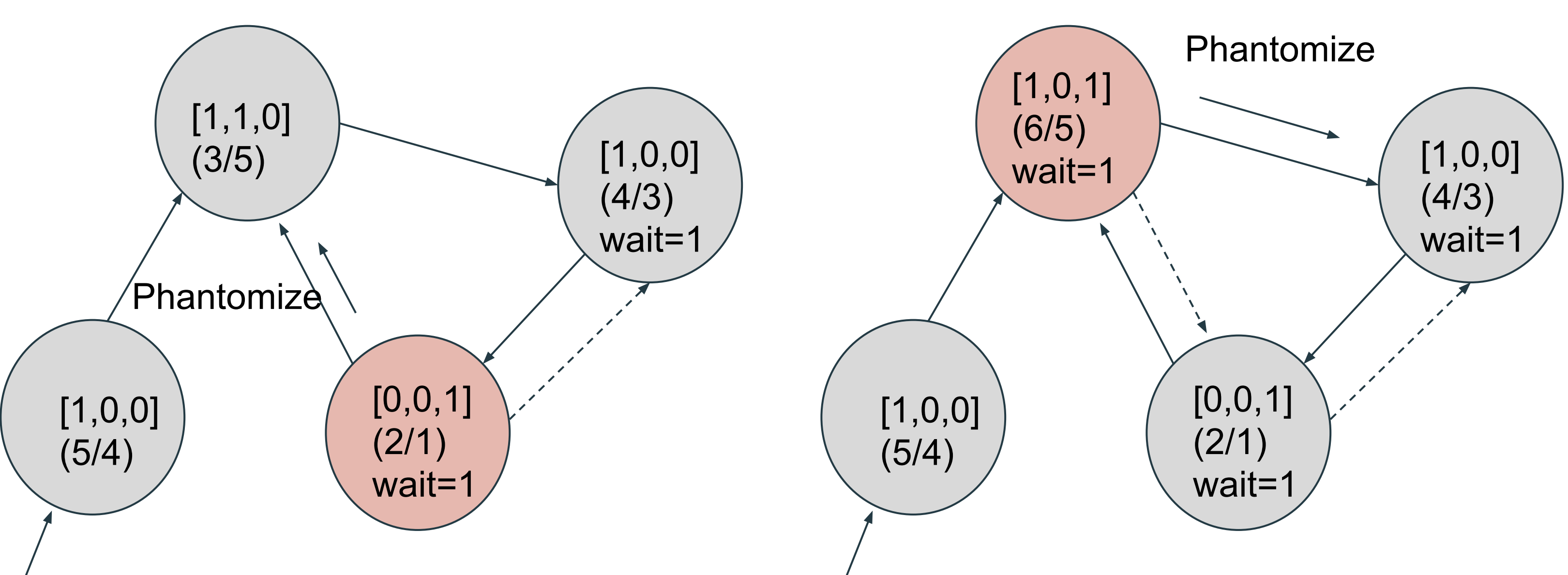
Reference counts are written like this: [2,1,0], it means strong count=2, weak count=1, phantom count=0. Weights are written like this (2/3), it means weight=2, max weight=3.



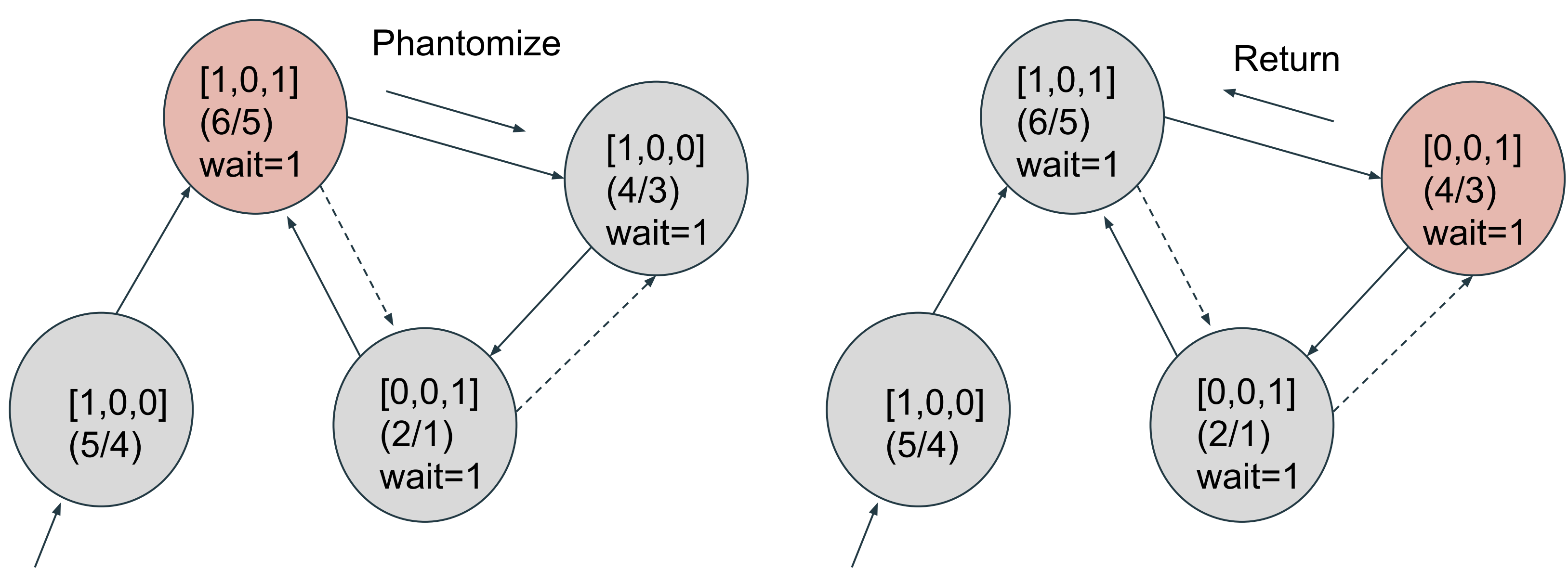
Collecting With SWPR



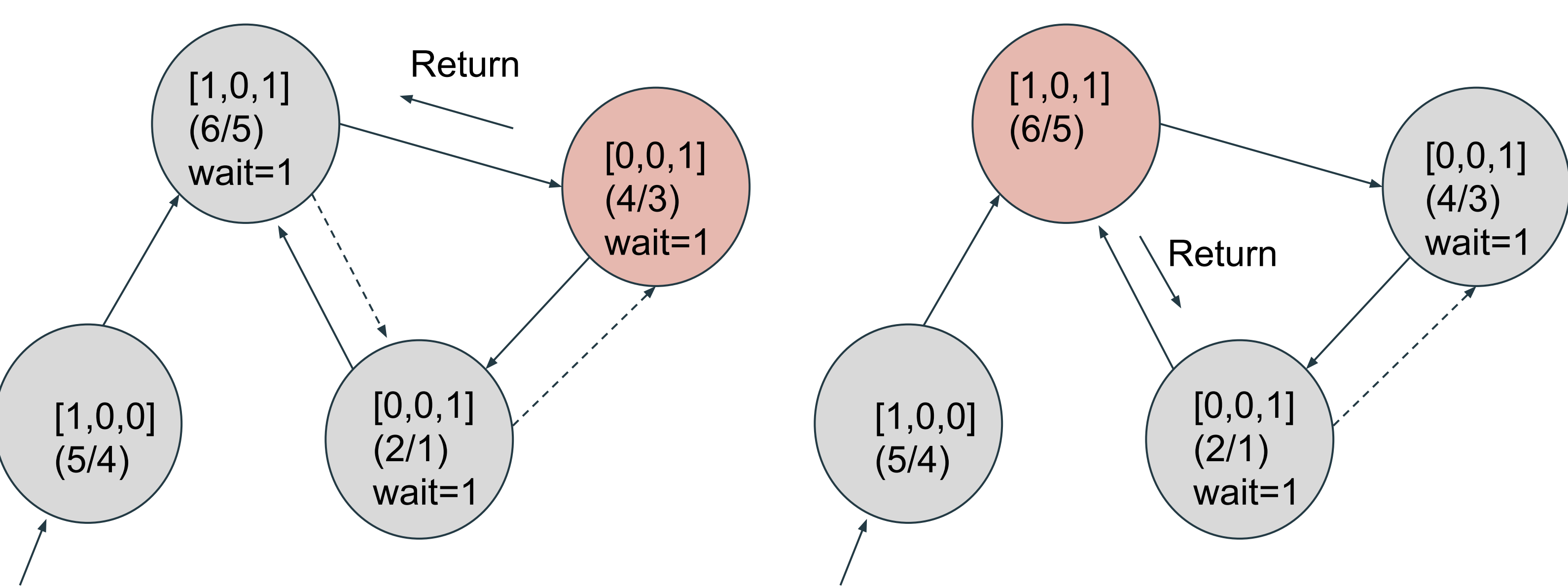
Collecting With SWPR



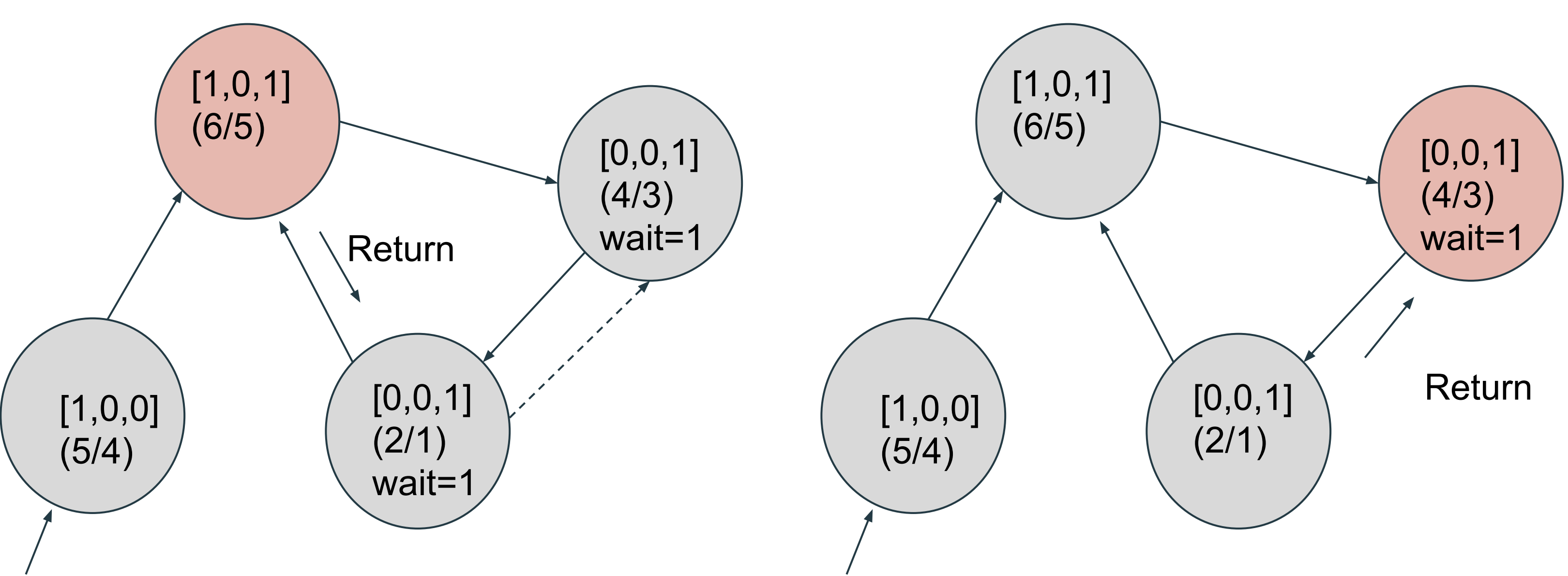
Collecting With SWPR



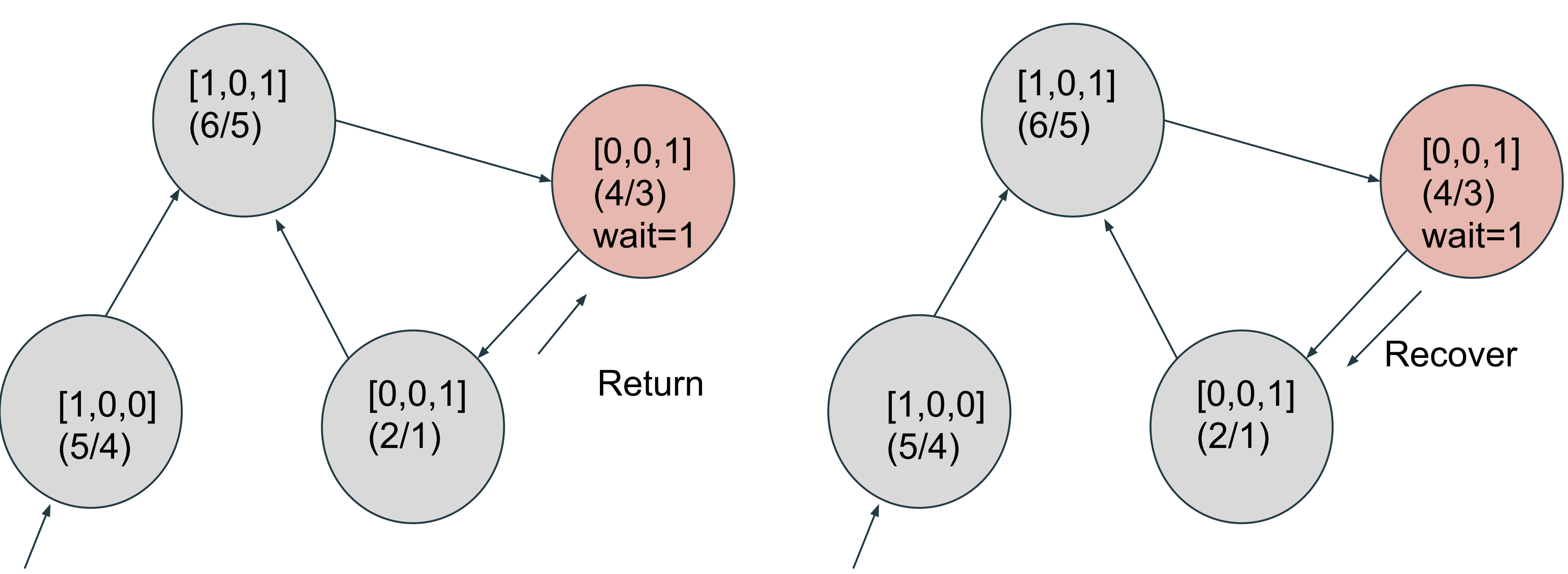
Collecting With SWPR



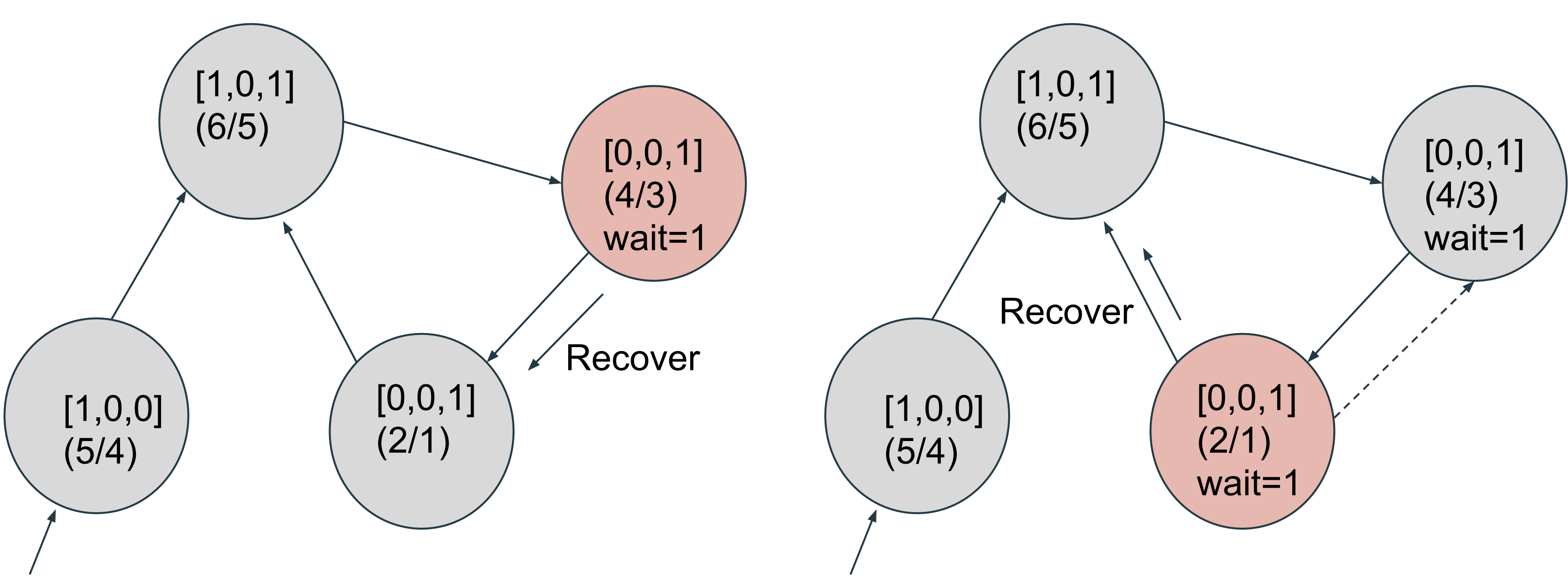
Collecting With SWPR



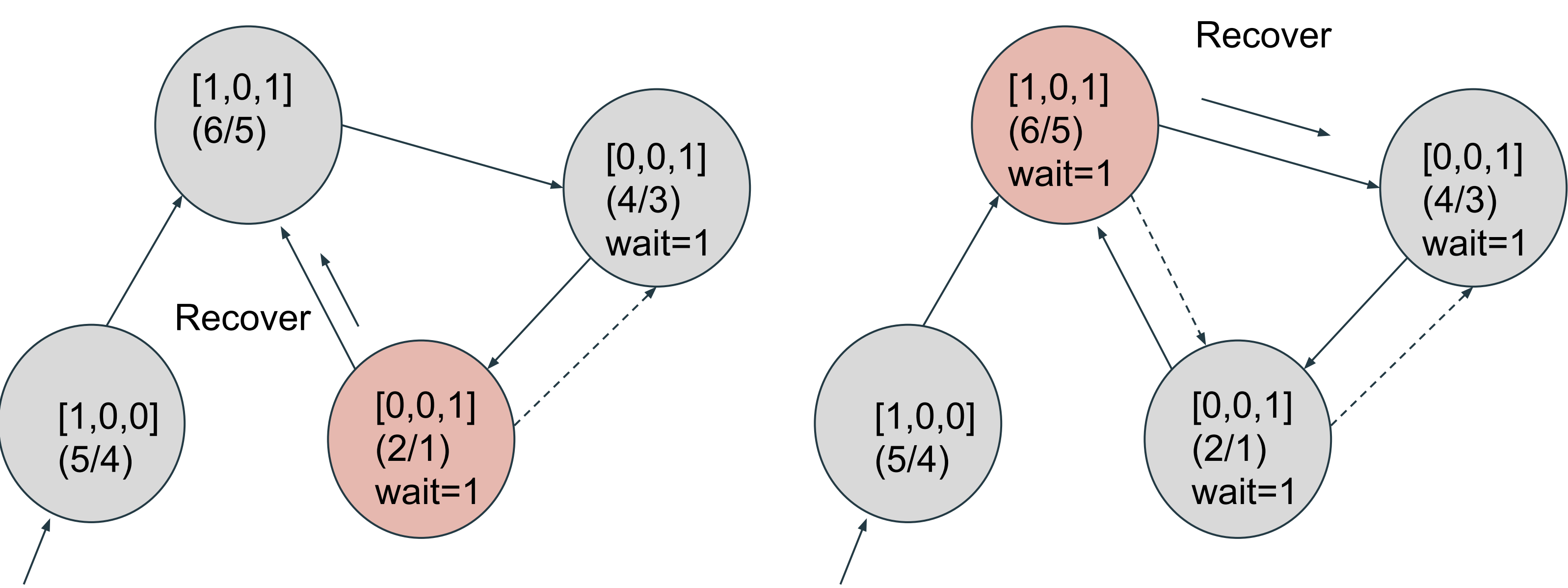
Collecting With SWPR



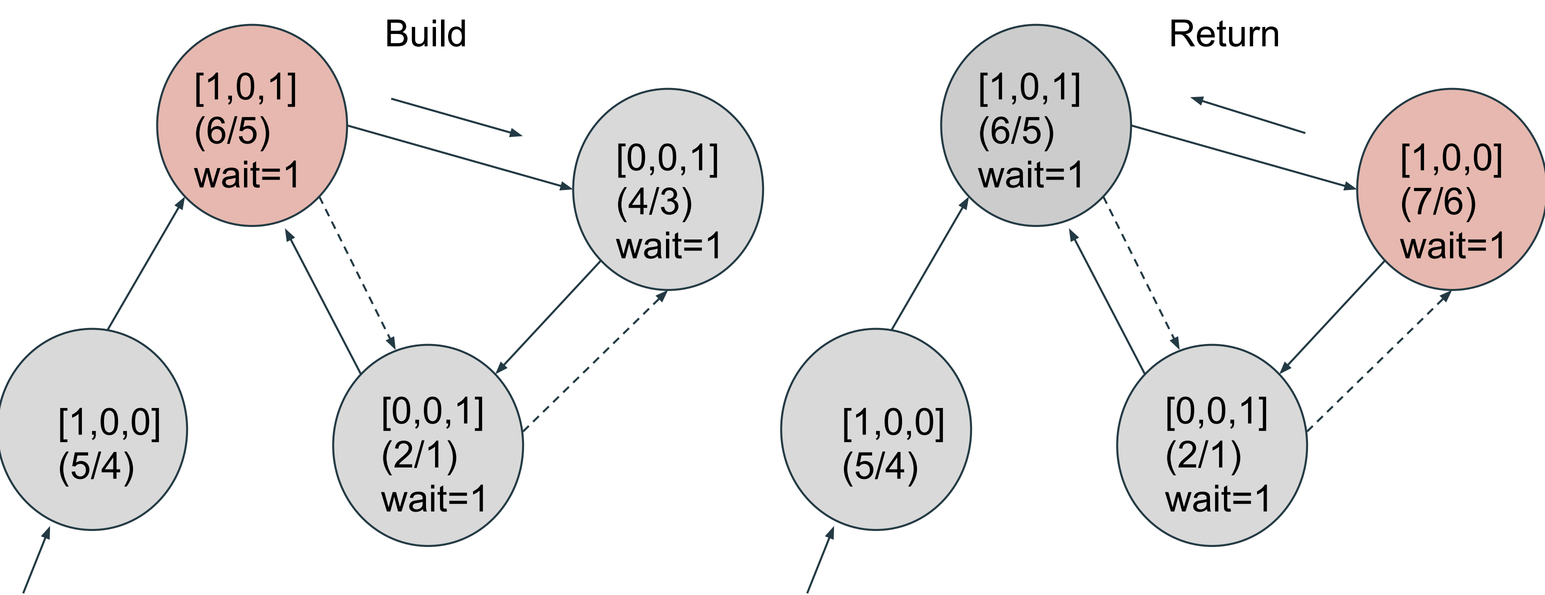
Collecting With SWPR



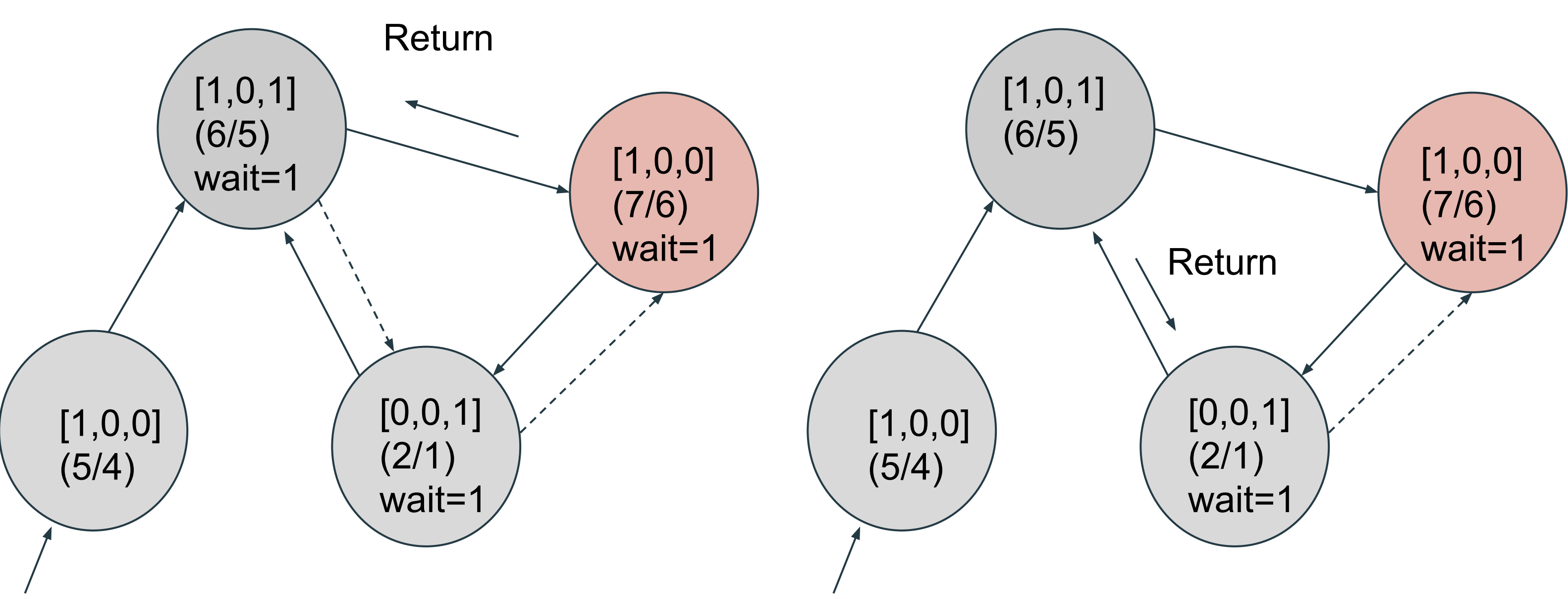
Collecting With SWPR



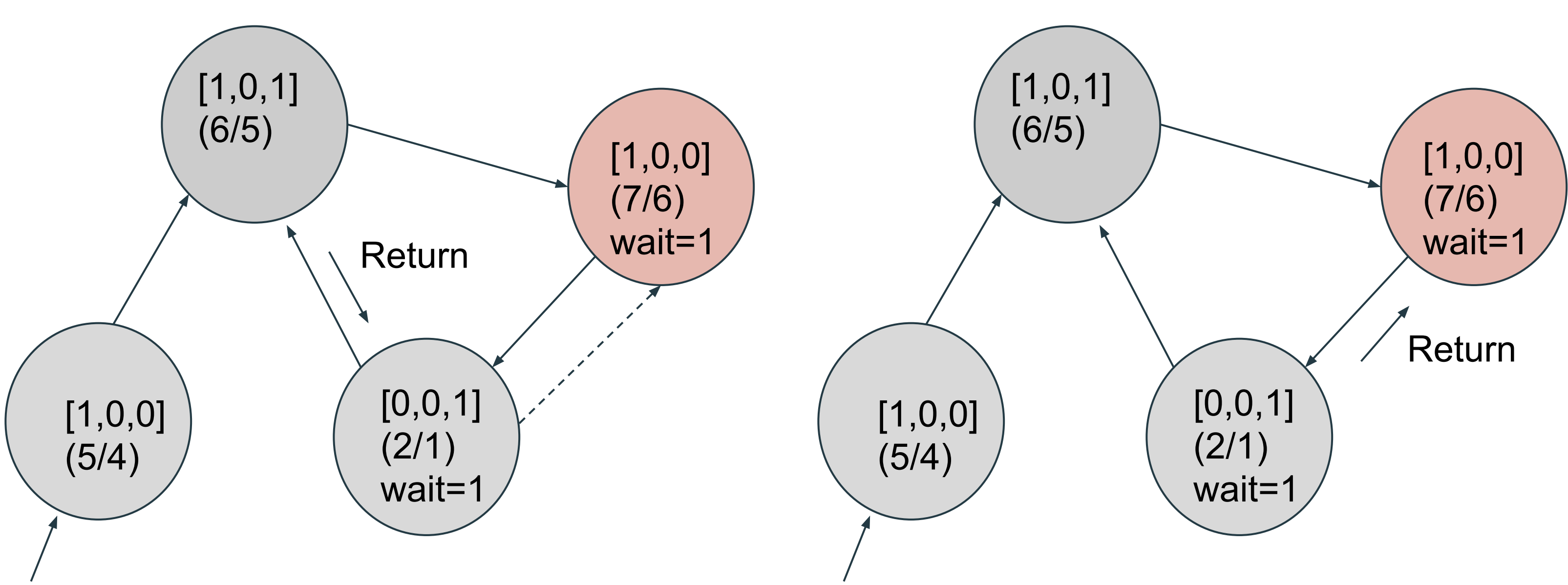
Collecting With SWPR



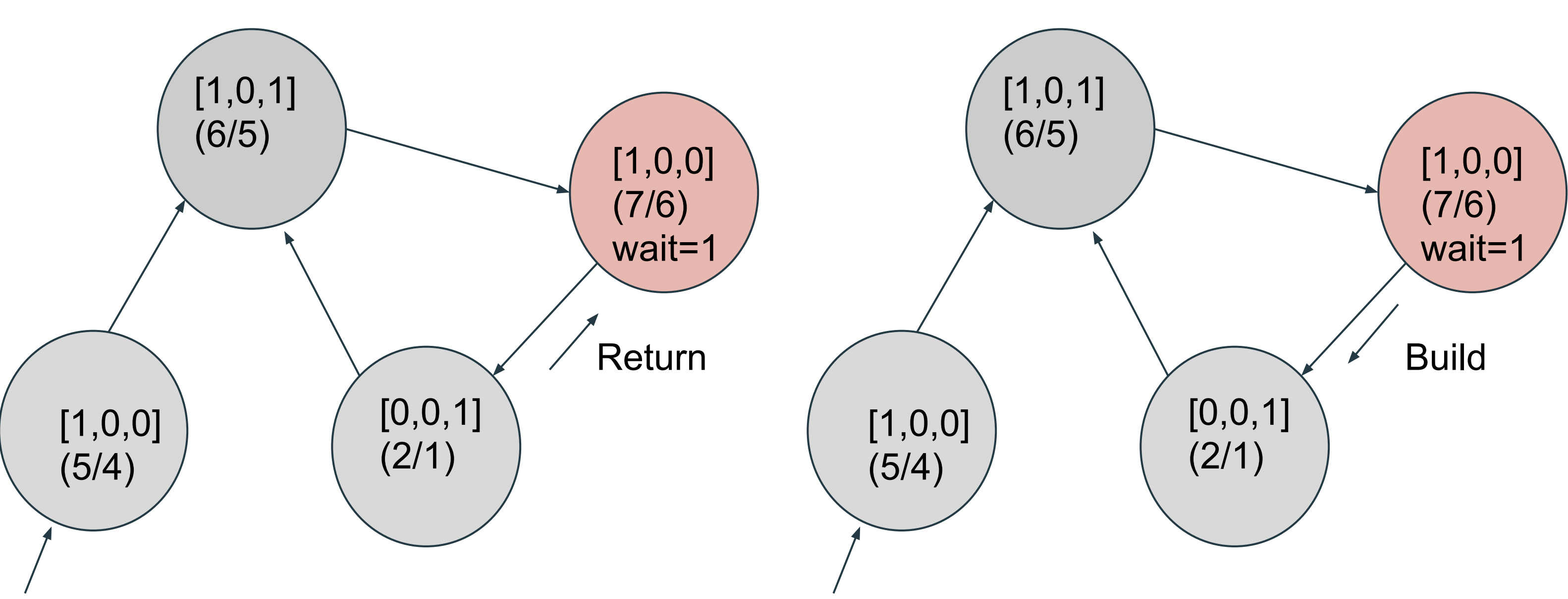
Collecting With SWPR



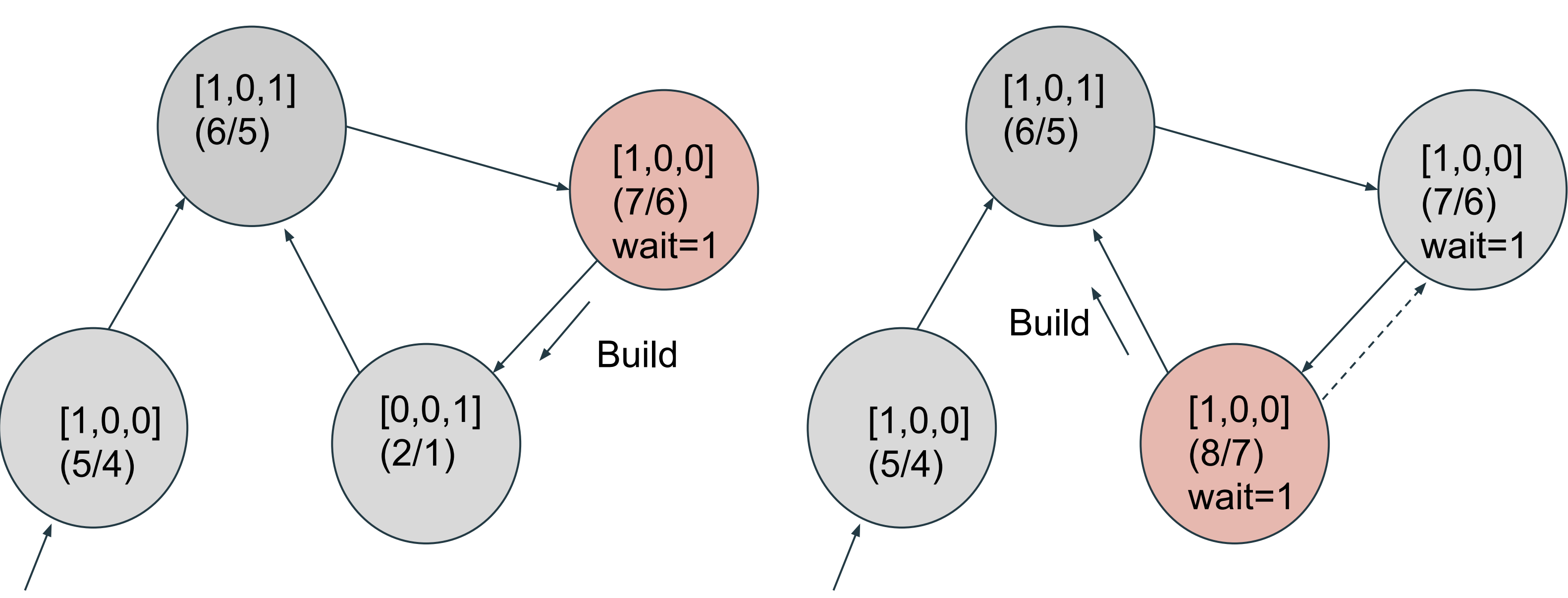
Collecting With SWPR



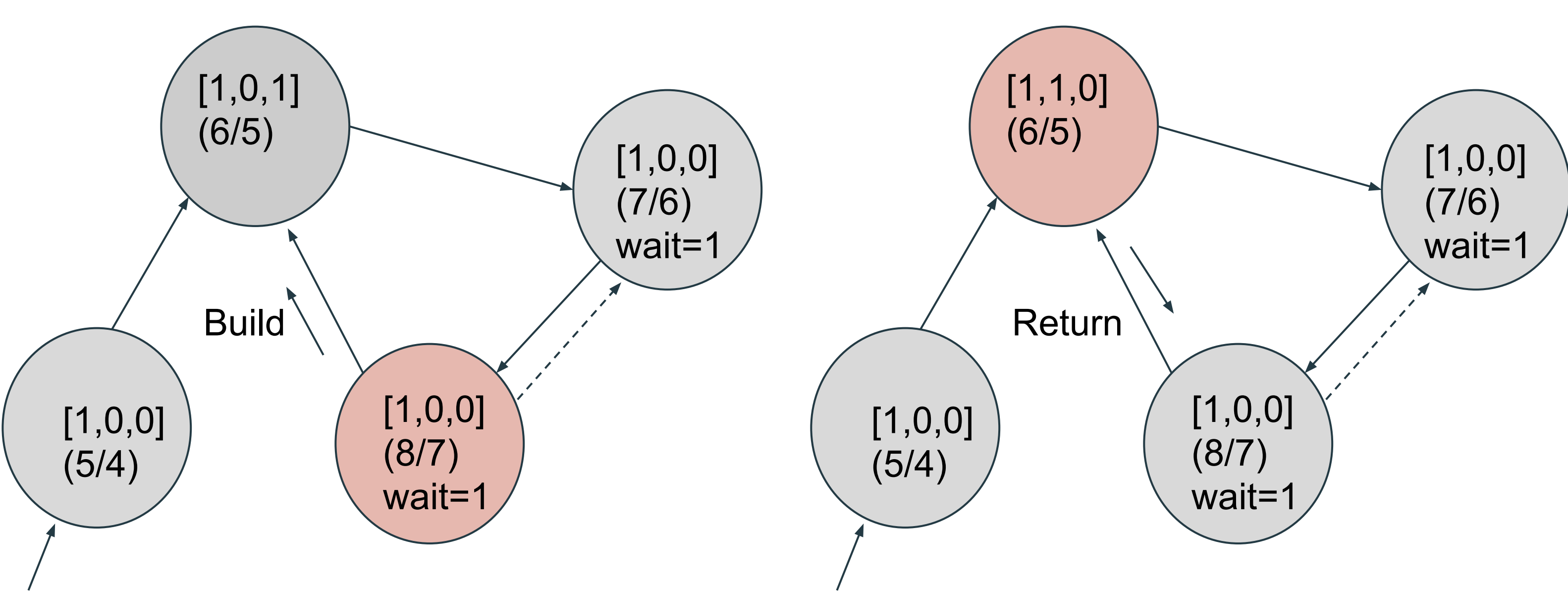
Collecting With SWPR



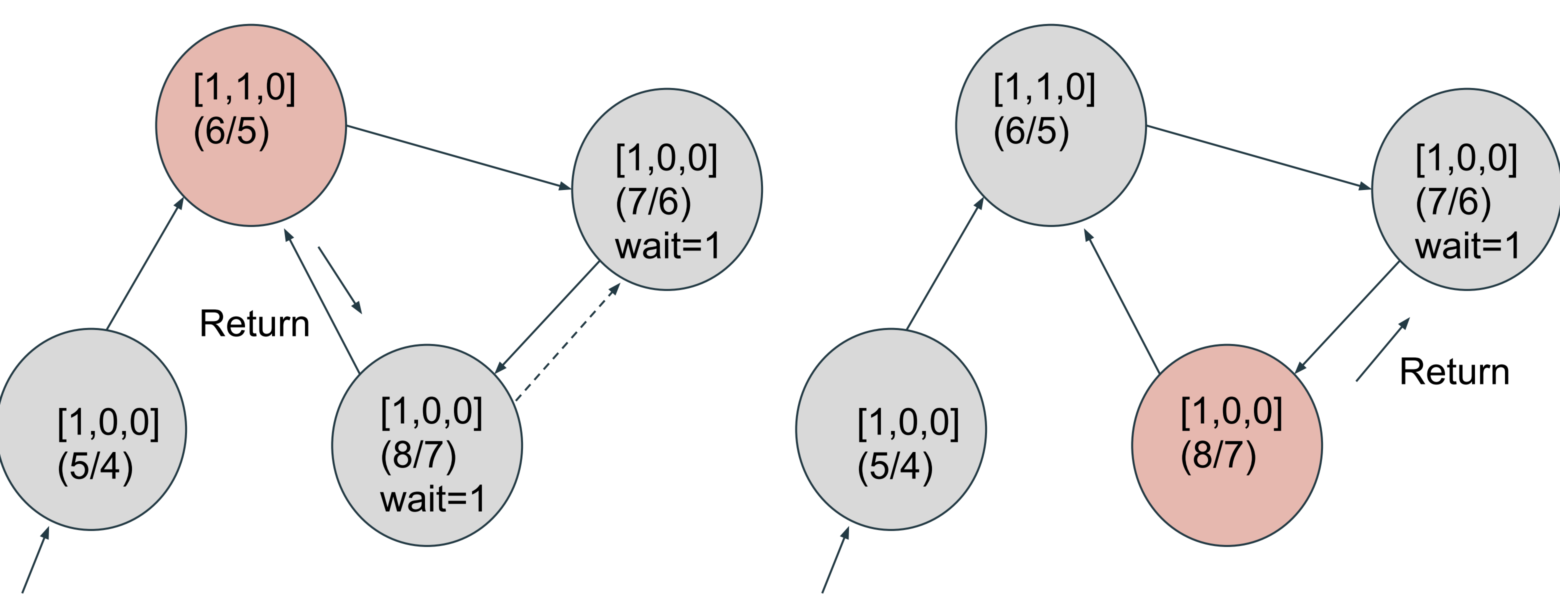
Collecting With SWPR



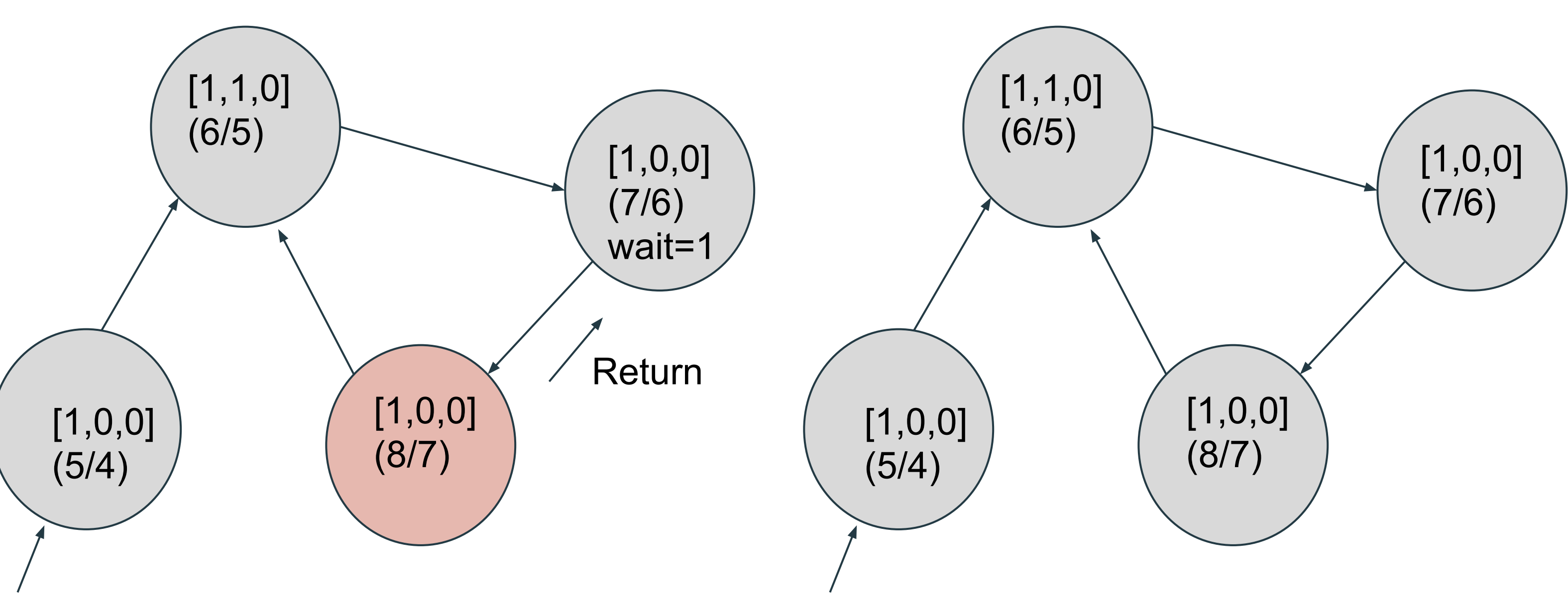
Collecting With SWPR



Collecting With SWPR



Collecting With SWPR

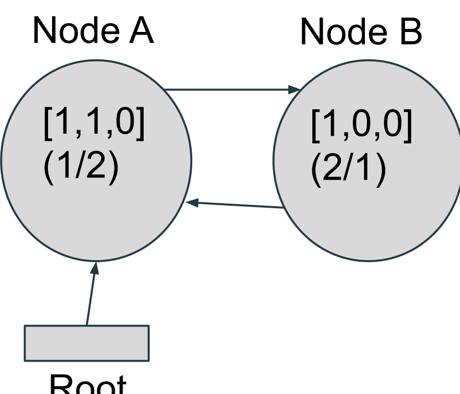


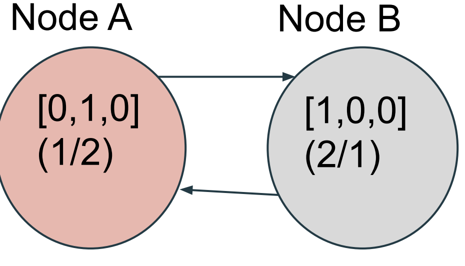
Collecting a Two Node Cycle…

Another Example with SWPR

Reference counts are written like this: [2,1,0], it means strong count=2, weak count=1, phantom count=0.

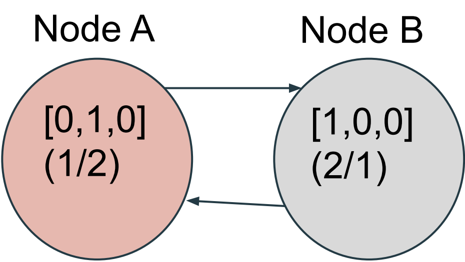
Weights are written like this (2/3), it means weight=2, max weight=3.

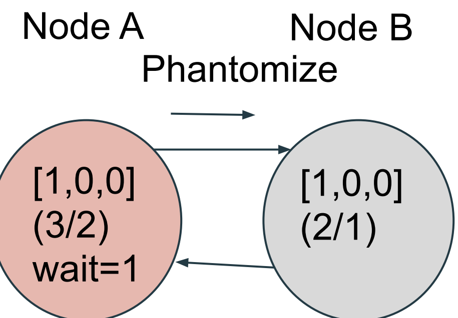




Remove root from the cycle.

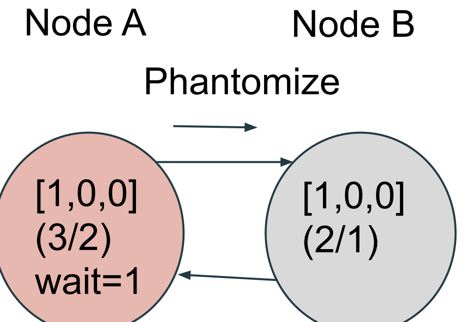
Another Example with SWPR

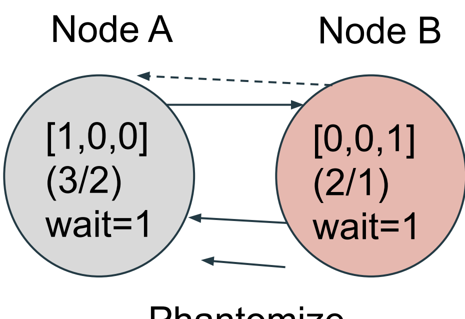




Node A phantomizes.

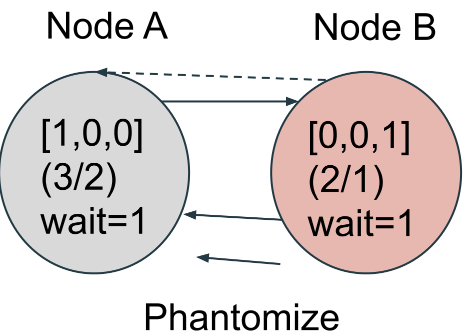
Another Example with SWPR

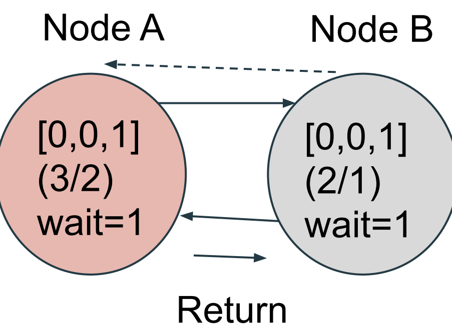




Node B propagates the phantomization.

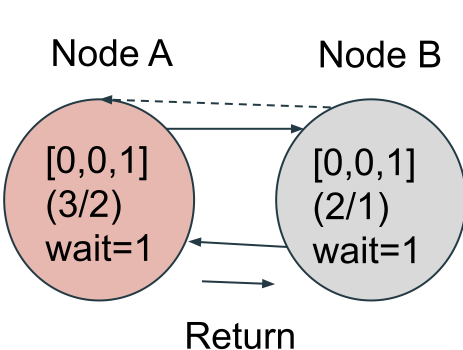
Another Example with SWPR

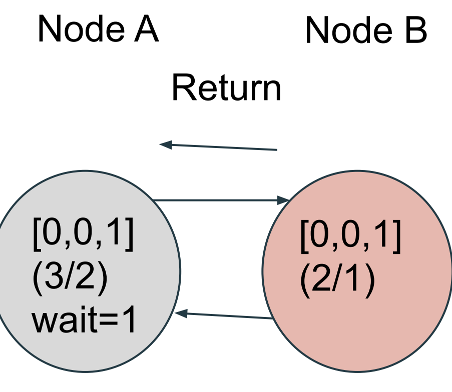




Node A is already phantomized, so it sends Return back.

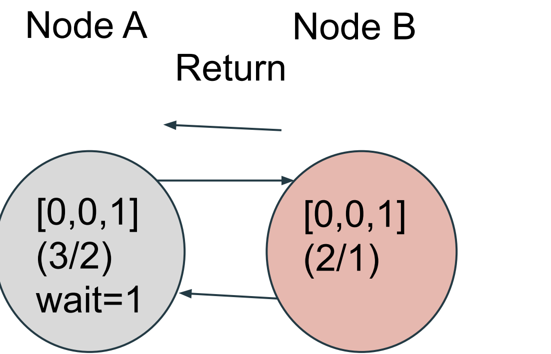
Another Example with SWPR

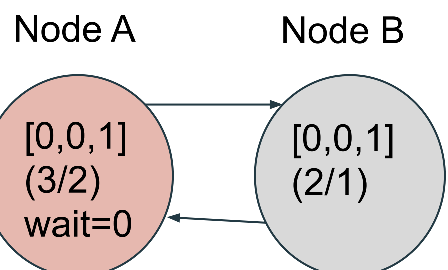




Node B’s wait count is now zero, so it sends Return to its parent, A, and clears its parent edge.

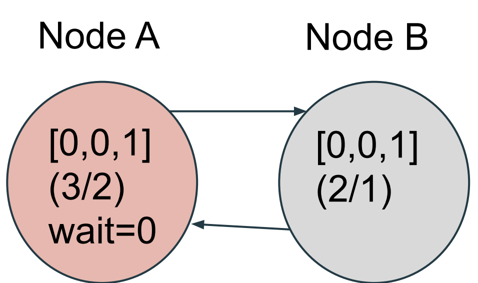
A Simple Example With SWPR

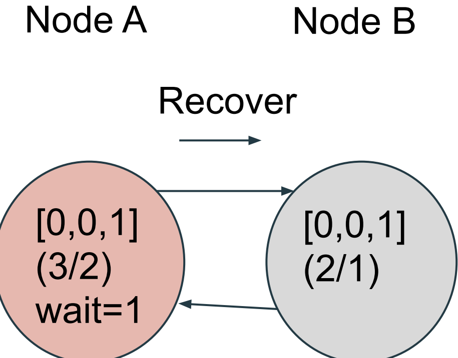




Node A's wait counter drops to zero. Phantomization is complete.

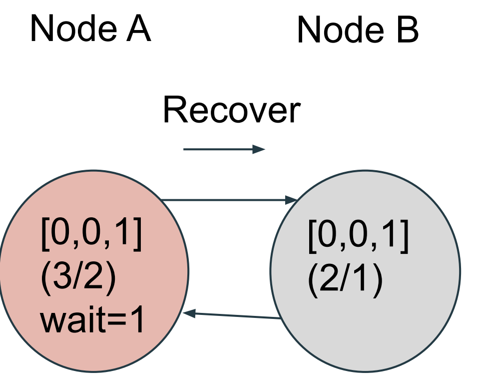
A Simple Example With SWPR

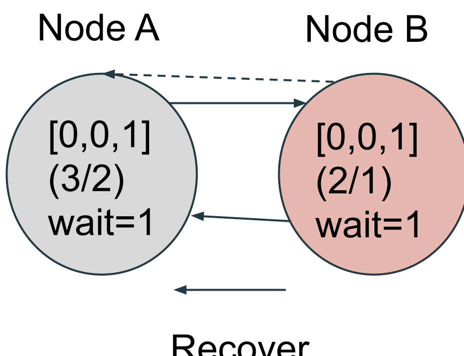




Node A has no strong count, so it might be garbage. It attemps to Recover, i.e. to look for phantomized nodes which have strong incoming edges from which it can rebuild the subgraph.

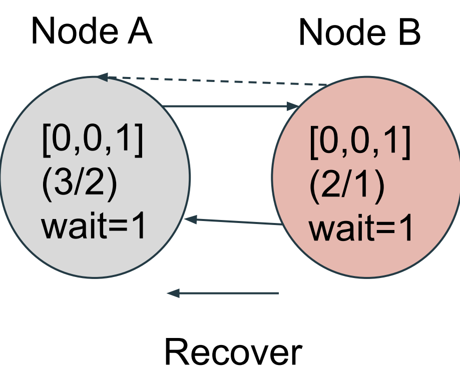
A Simple Example With SWPR

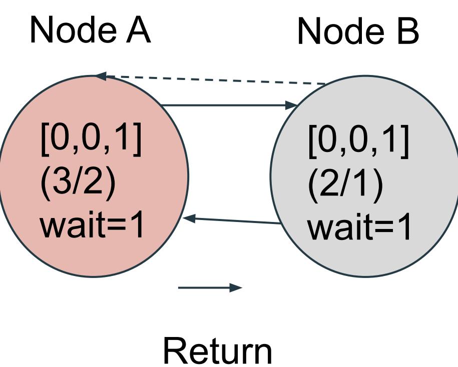




Node B receives recover. It has no strong edges, so it propagates the Recover.

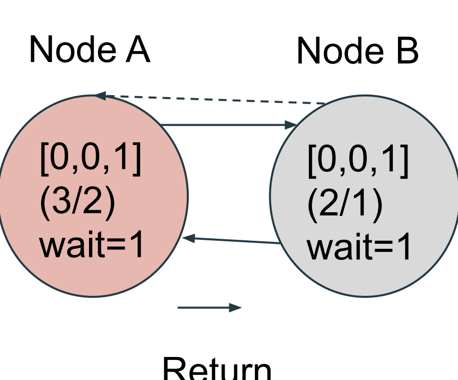
A Simple Example With SWPR

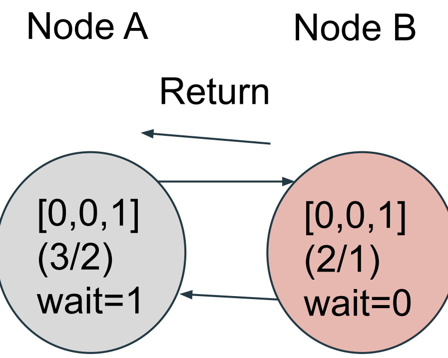




Node A is already recovering, so it sends Return.

A Simple Example With SWPR



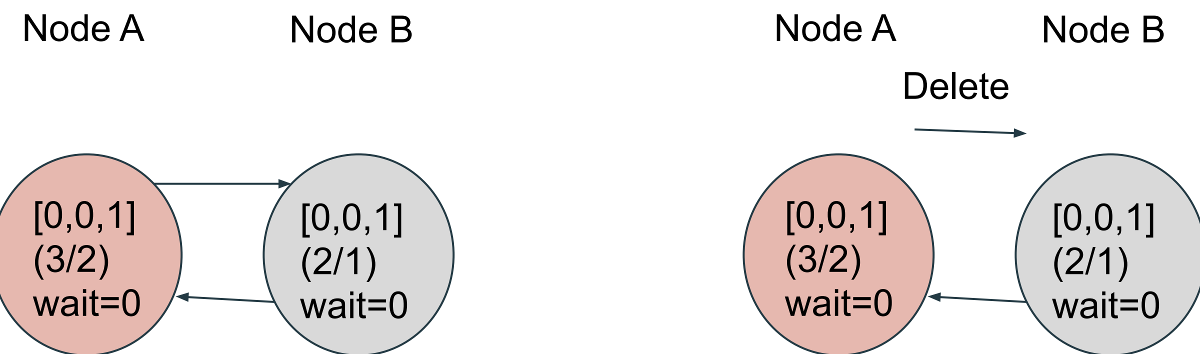


Node B’s wait count is zero, so it returns and clears its parent edge.

A Simple Example With SWPR

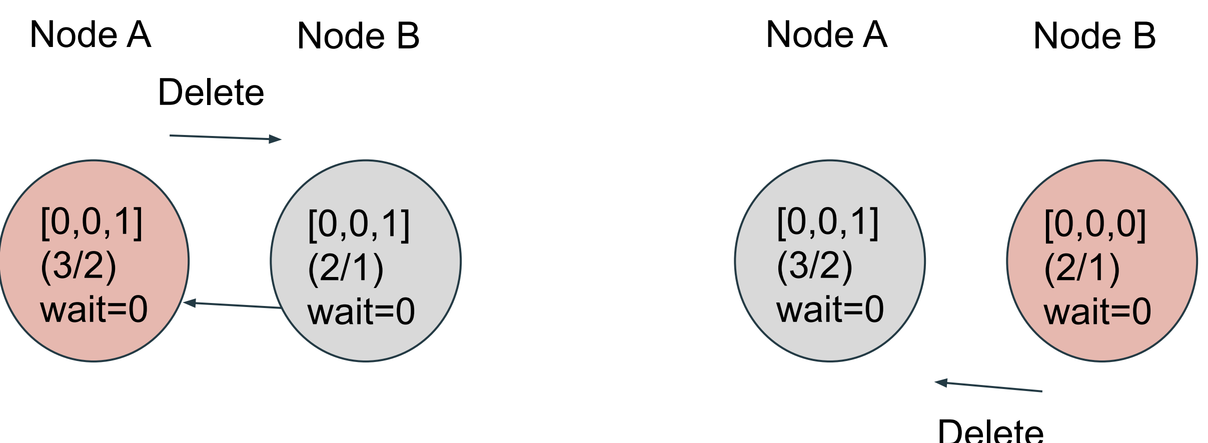
At the end of recovery, Node A still has no strong nodes...

A Simple Example With SWPR



Node A knows it's garbage. It sends Delete along all its outgoing edges, then deletes those same edges. It does not set its wait count.

A Simple Example With SWPR

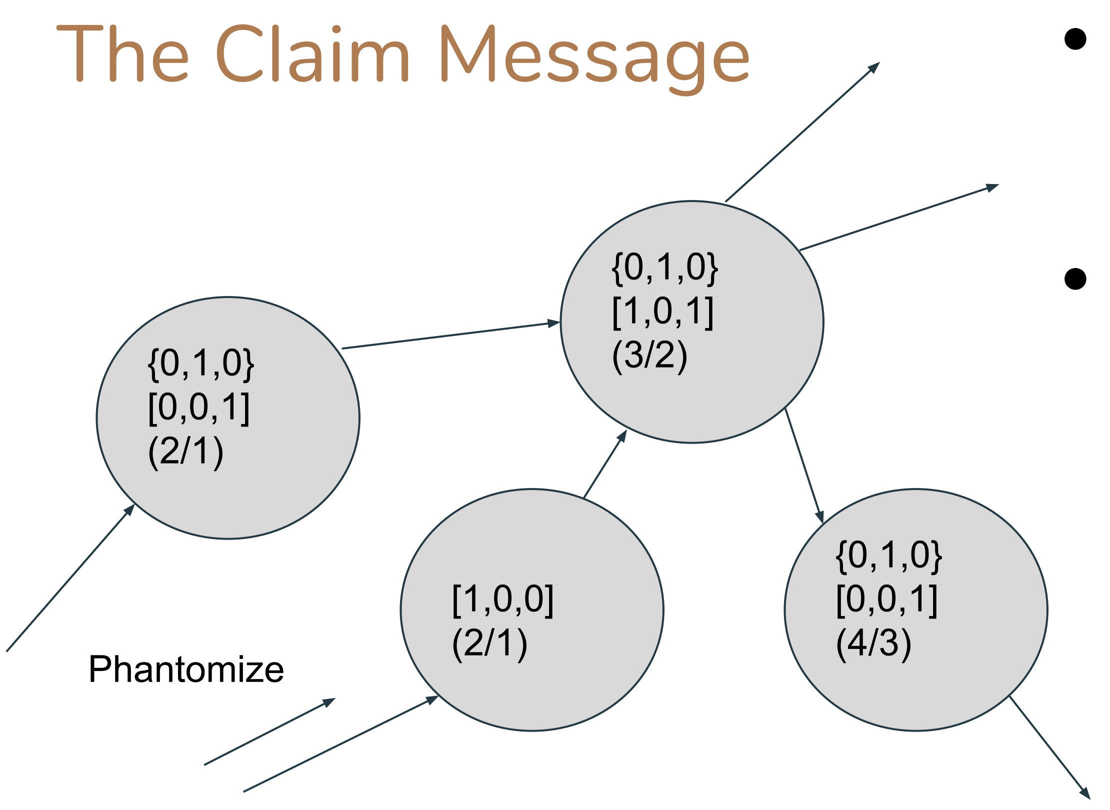


After Node B receives Delete, it propagates the message and deletes itself. Node A cannot be reclaimed until it receives Delete from B. Once it does, it deletes itself.

The Multicollector

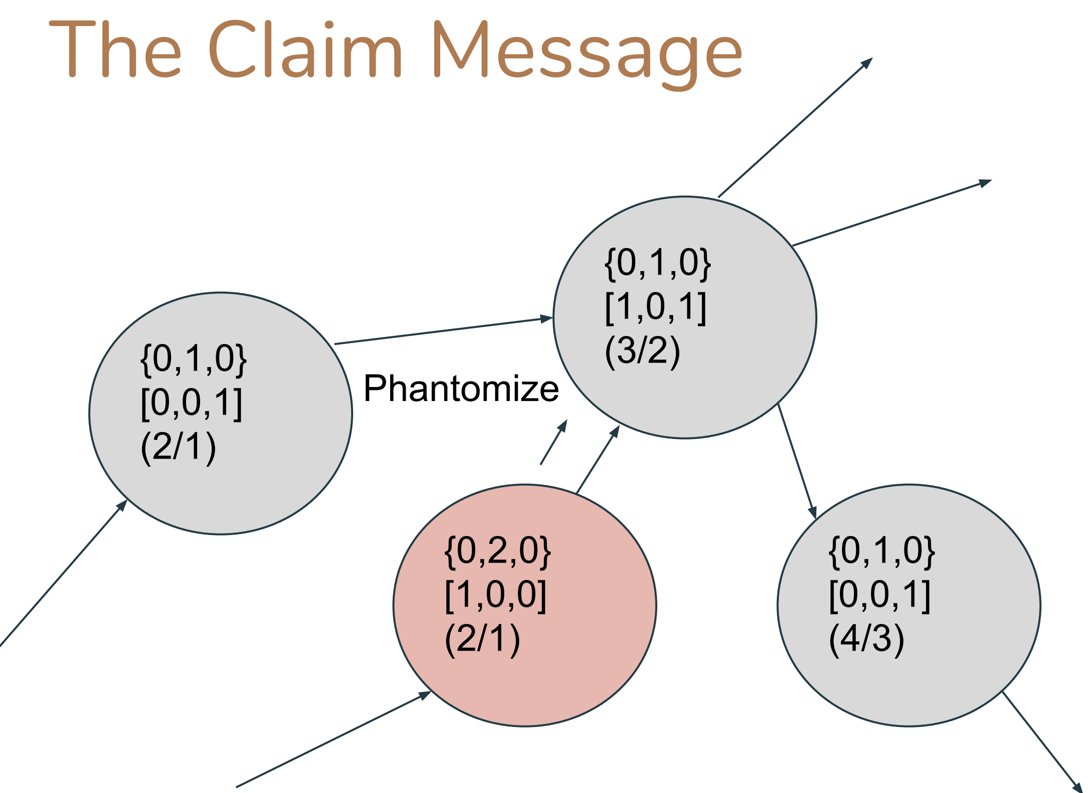
- You now know the basic method
- Works if there’s no interference
  - No other collector
  - No mutating process
- Multicollector Mechansims - Enforce global ordering among collections with no cycles
  - A identifier tuple [major id, id, minor id]
  - A “Claim” message
  - Start Over flag
  - Recovery Count (the “R” in SWPR)

The Claim Message



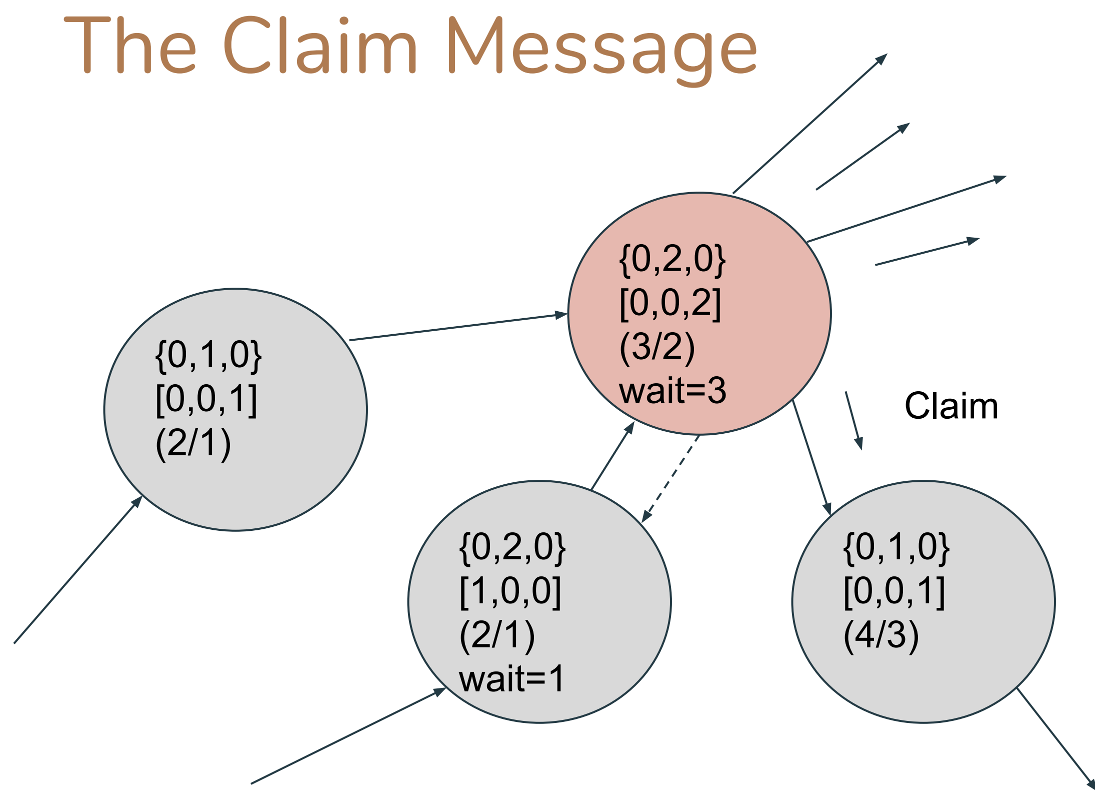
- The tuples {0,1,0} and {0,2,0} are collection id's. For now, the first and last tuple element are zero.
- The middle value is the main id, and must be unique. It can be the node id.

The Claim Message



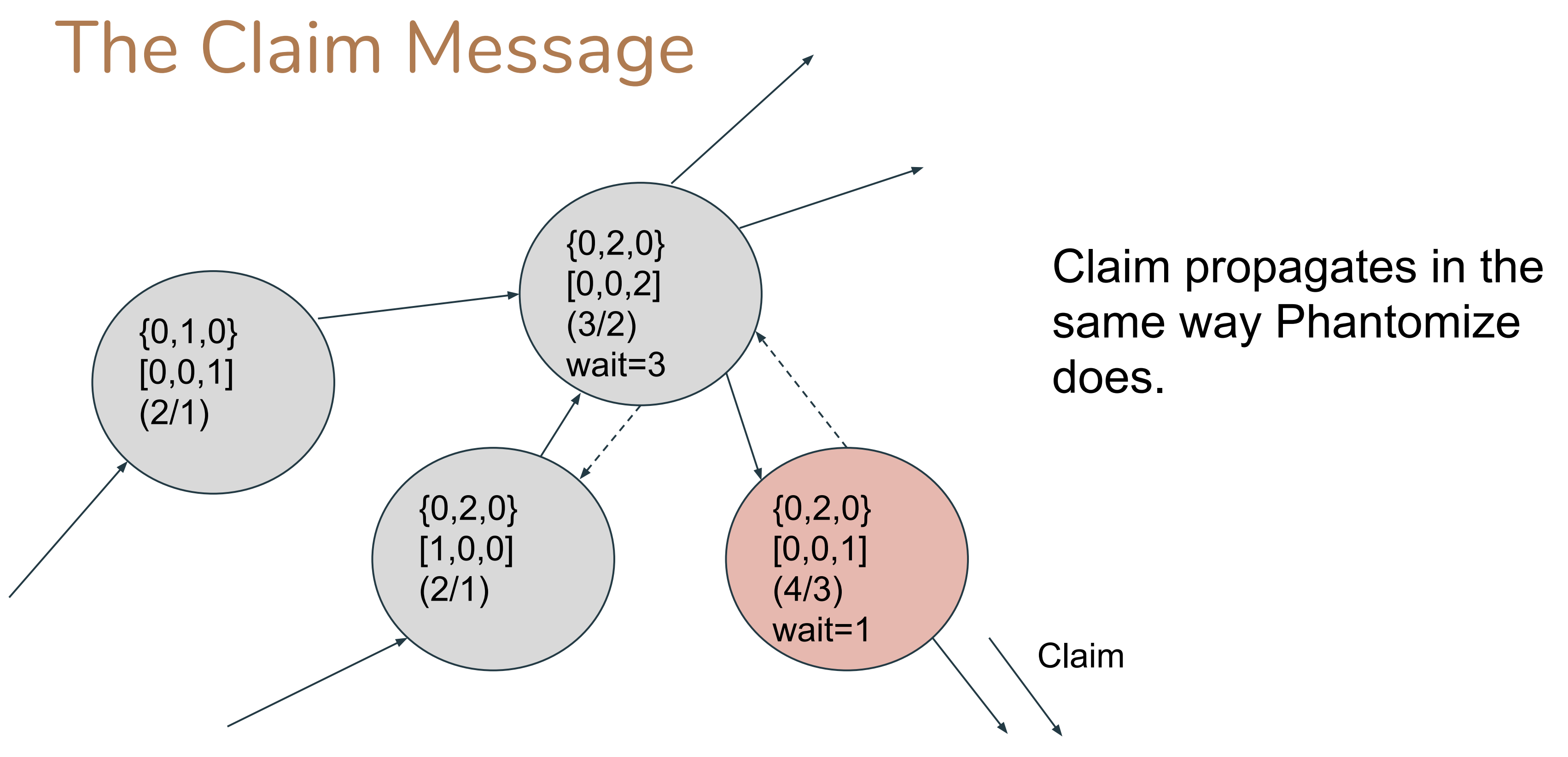
The Phantomize message is traveling toward an already phantomized node. Normally this would immediately return. If the sending tuple were smaller or equal to the receiving, it still would return.

The Claim Message



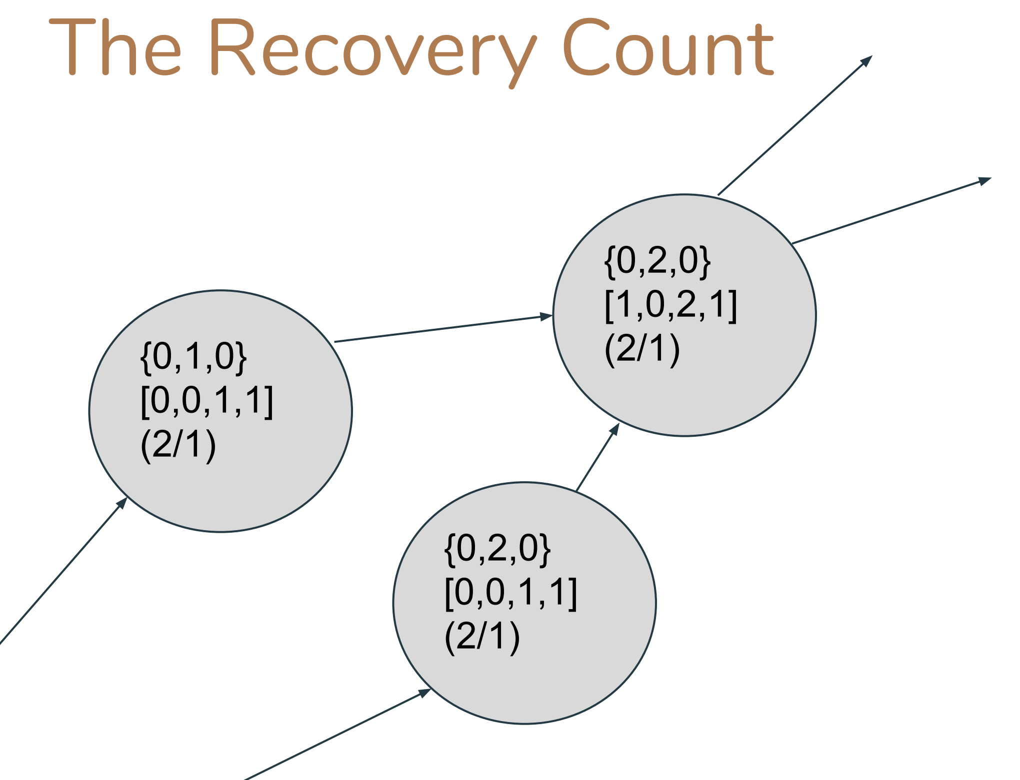
However, in this case, the Phantomize comes from a higher collection id, namely {0,2,0}. So instead of returning immediately, it marks the receiving node with its id and sends Claim along its outward edges..

The Claim Message



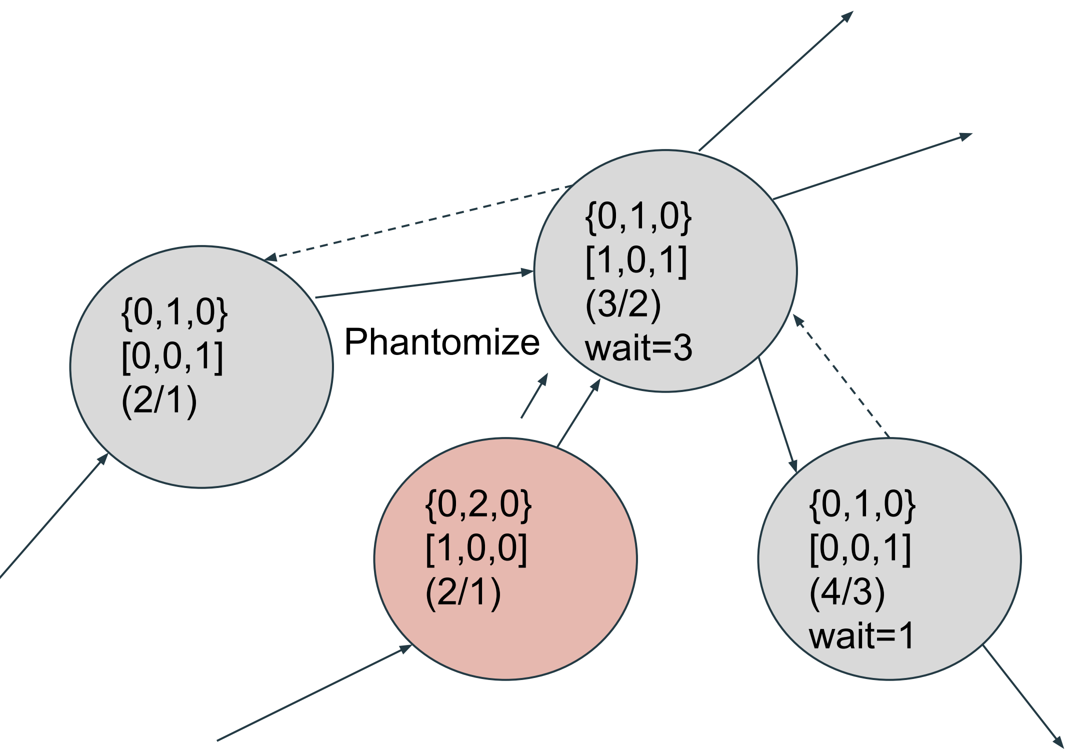
Claim propagates in the same way Phantomize does.

The Recovery Count



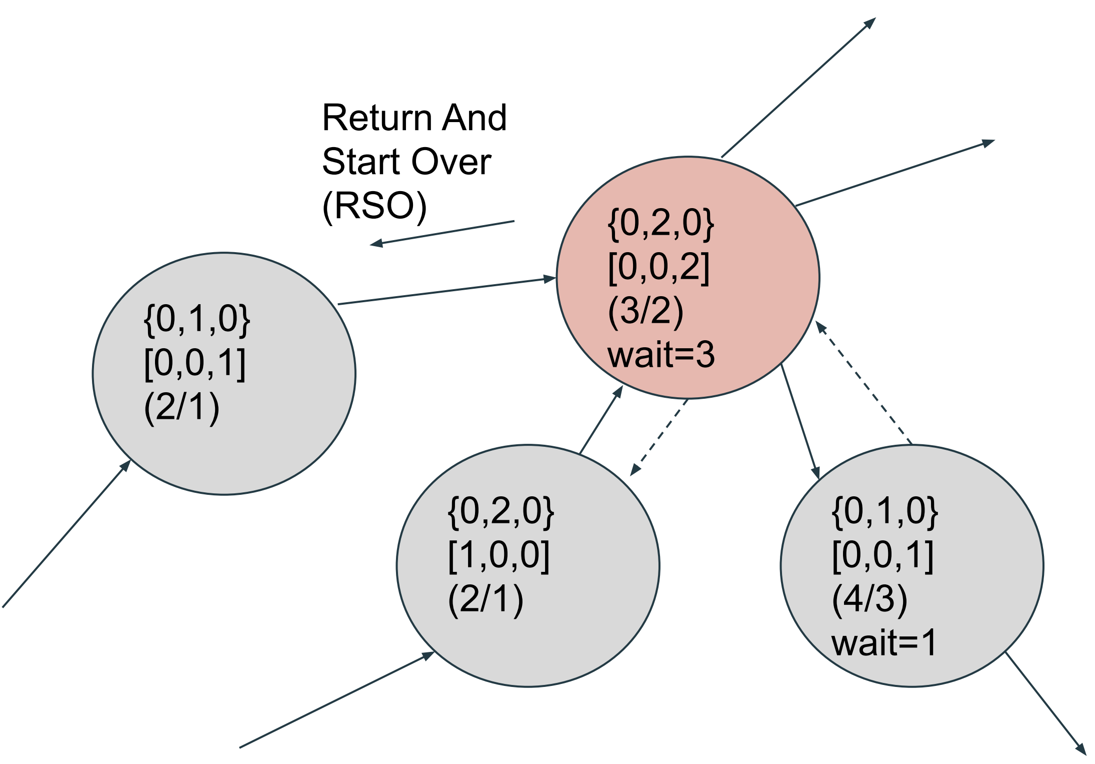
- When a node receives a recovery from a node with the same collection id, the recovery count is incremented.
- Unless the phantom and recovery count are equal, nodes cannot send return messages or decide to start deleting.

The Return and Start Over Mechanism



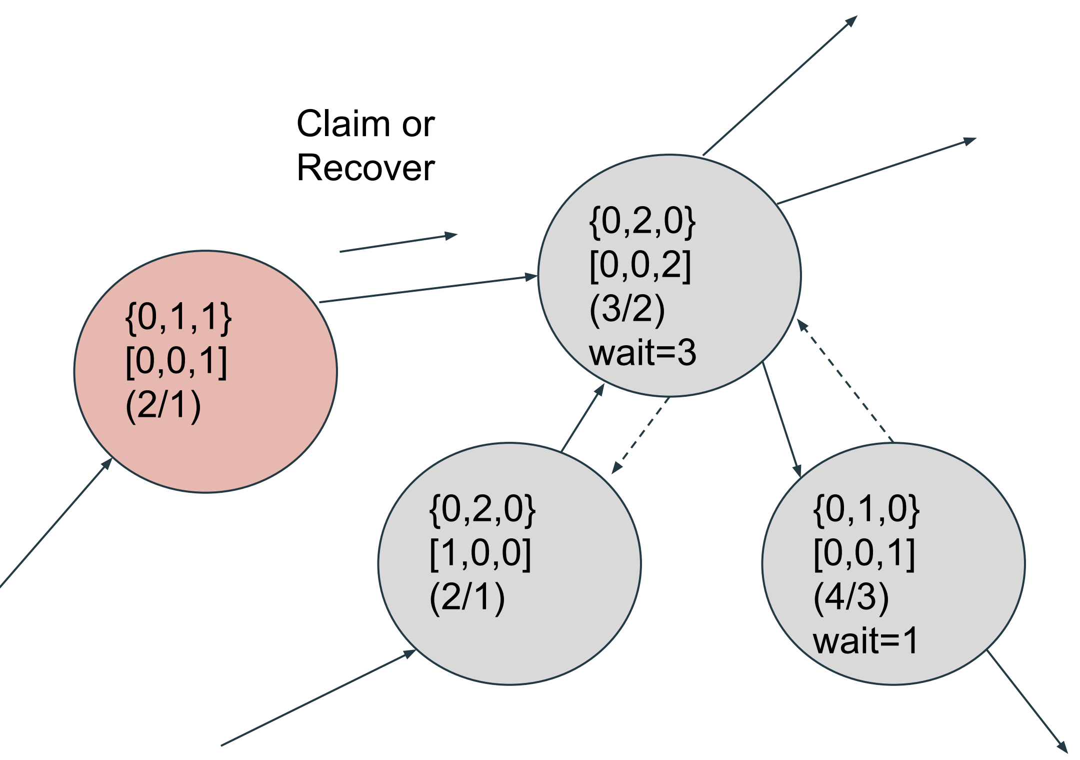
Consider this scenario...

The Return and Start Over Mechanism



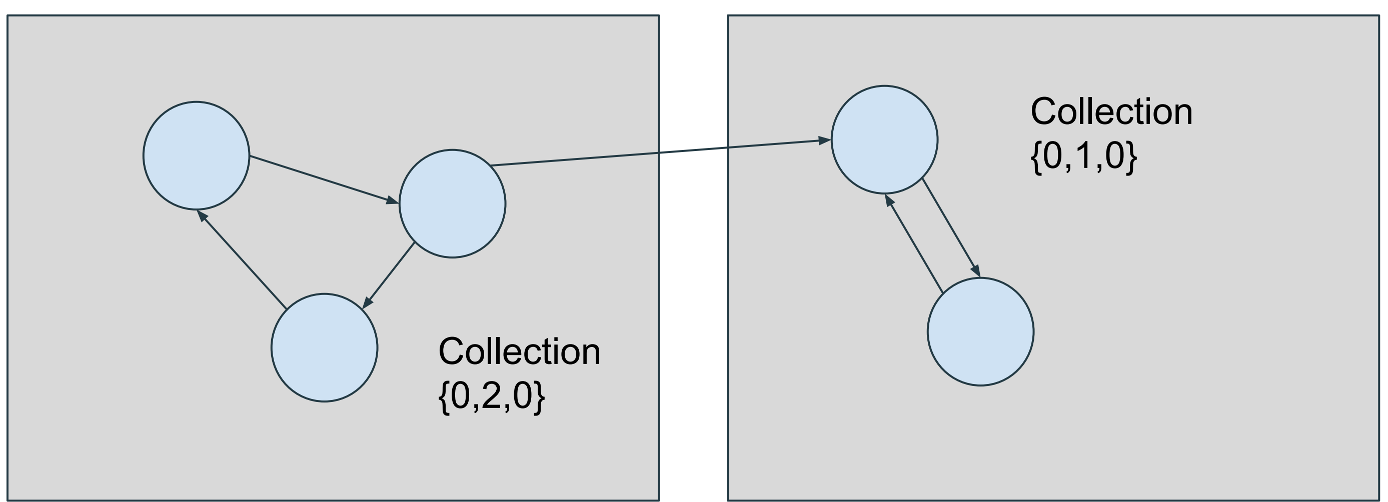
The new collection has to take over, but we can only have a single parent edge, so we send Return And Start Over to the current parent and set a new parent.

The Return and Start Over Mechanism



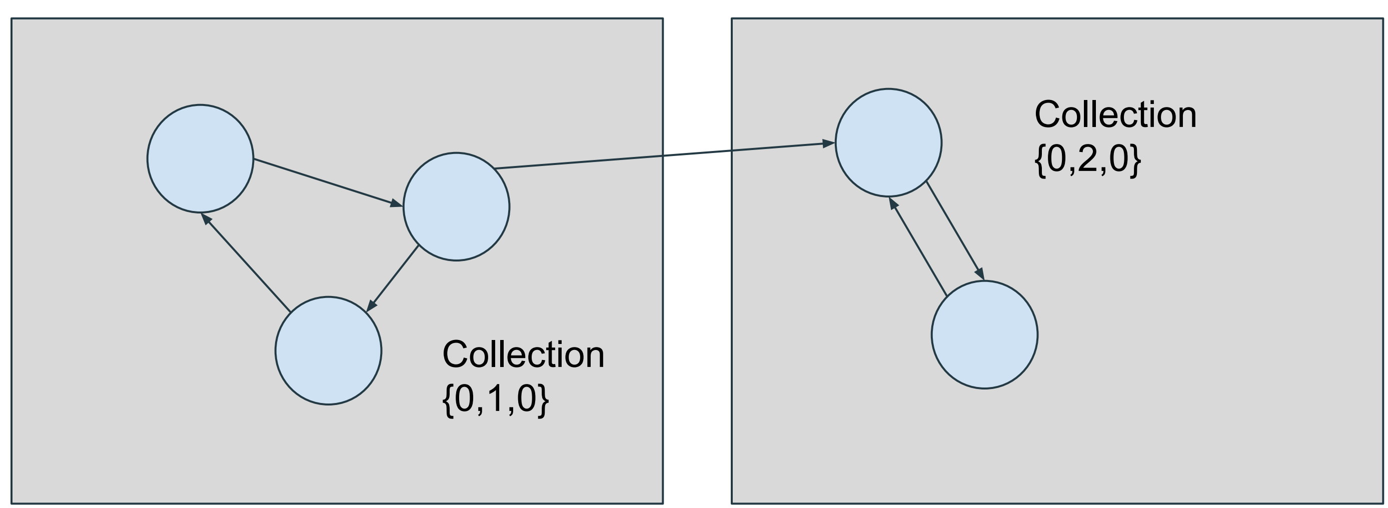
Pink node starts over with a slightly higher collection id (incremented minor id)

How These Mechanisms Work Together



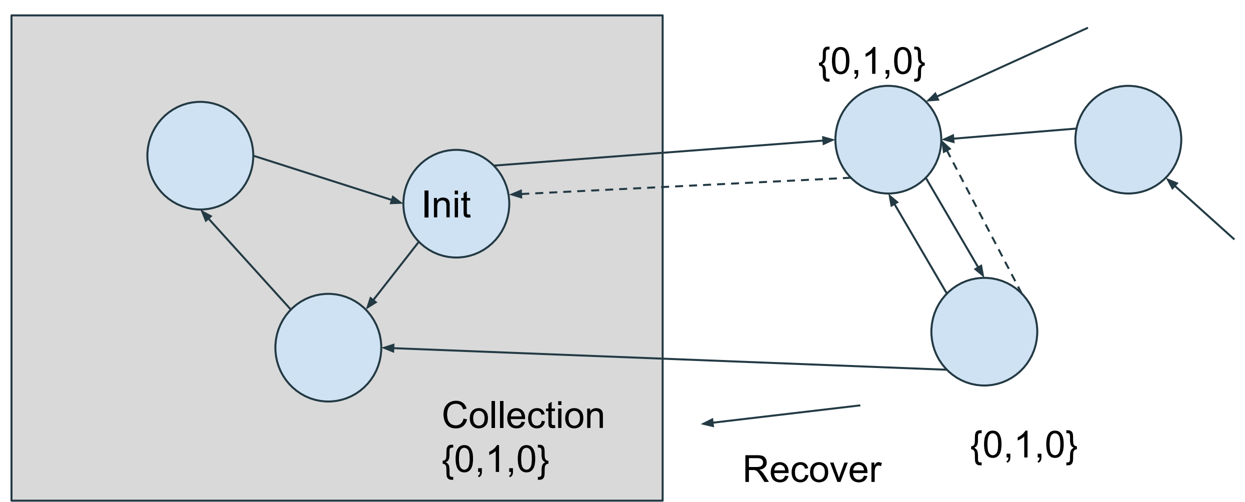
Collection {0,2,0} takes over the collection of the two nodes at right.

How These Mechanisms Work Together, Take 2



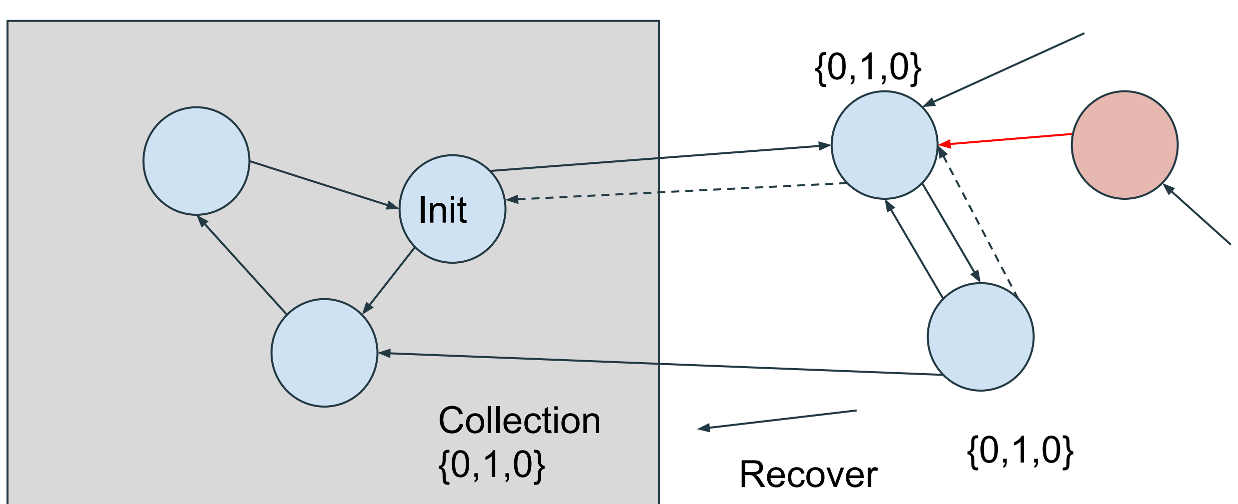
Collection {0,2,0} waits for {0,1,0} to finish, then collects its two nodes.

How These Mechanisms Work Together, Take 3



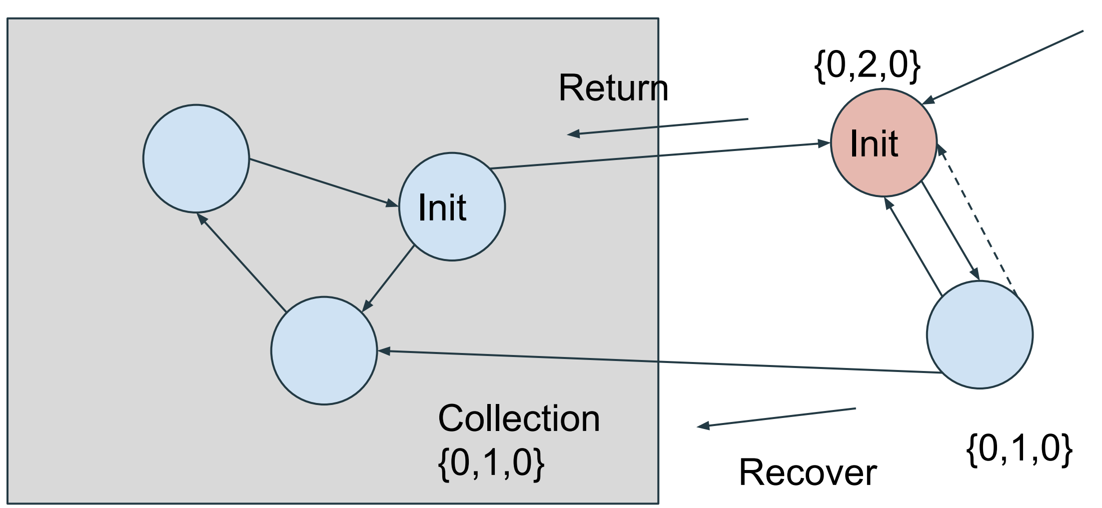
Collection {0,1,0} starts...

How These Mechanisms Work Together, Take 3



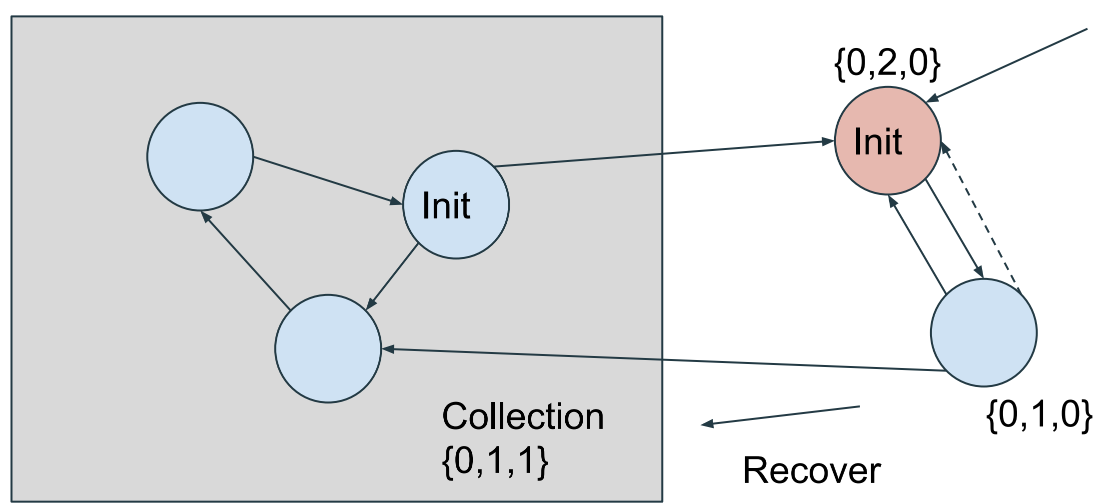
The red edge is removed

How These Mechanisms Work Together, Take 3



Danger: If we didn't increment the minor id, the blue "Init" could finish and clean up, but it should be kept alive by red "Init."

How These Mechanisms Work Together, Take 3



Collection {0,1,1} stalls and waits to be taken over by {0,2,0}

Working with a Mutator

- Easier than you might expect…
- A new edge from a phantomized node is a phantom edge
- If we delete a phantomized edge, create a new collection with an incremented major id.