A Reference-Counting Garbage Collection Algorithm for Cyclical Functional Programming

Baltasar Trancón y Widemann

Universität Bayreuth, Germany

ISMM '08
Tucson, Arizona
2008–6–7/8

Reference Counting
&
Cycles

Reference Counting

In a Nutshell

- simple basic algorithm
  - count number of live references to a cell
  - reference count drops to zero ⟹ garbage
- often considered outdated, but not quite dead yet!

Famous Problem

- what about cycles?
  - reference count drops to zero ⟸ garbage

History of Cyclic Reference Counting

The Algorithm of Brownbridge (1985–88)

Principle

- partition references into strong and weak subset
  - no cycle entirely strong
  - weak edges irrelevant for reachability (maintain!)
- collection based on strong count only

Advantages

- intuitively appealing

Disadvantages

- hard to get right, complex code
- efficiency issues

History of Cyclic Reference Counting

The Algorithm of Lins, Martínez & Wachenchauzer (1990)

Principle

- detect cyclic garbage by “speculative deletion”
- revert if false positive

Advantages

- easy to understand & implement
- potential for optimization & heuristics

Disadvantages

- basic algorithm speculates too often, inefficient
- thoroughly confounded by sharing

Common Special Cases

Acyclic Data

Acyclic Data

- some data may not have cycles at all

  statically by type
  dynamically by usage

- plain reference counting preferred

Fixed Data

- global constants & let bindings
- reachable by root references
- lower bound for lifetime known

Functional Programming & Cycles

Cyclical Functional Programming

In Common With Other FP Paradigms

- purely functional; immutable data
- free data types & recursion
- strict; no infinite data

Speciality: Cycles

detect by searching the call stack for recurring inputs

handle by special values & operations

Cyclical Functional Programming

Cycle Handling

Cycle Handling & Unfold

- build result top-down (*destination passing*)
- upon cycle, just copy previous result (*ditto*)
- effective for all primitively corecursive functions

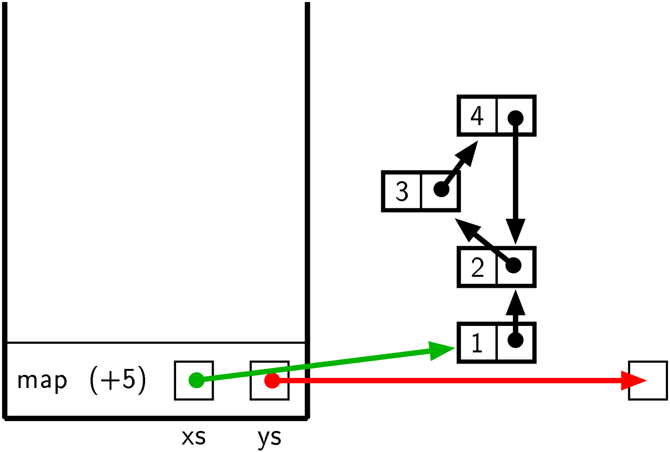
Cycle Handling & Search

- traverse recursively
- upon cycle, return truth value immediately
- fixed point semantics

false	least fixed point
true	greatest fixed point
either	intermediate fixed points

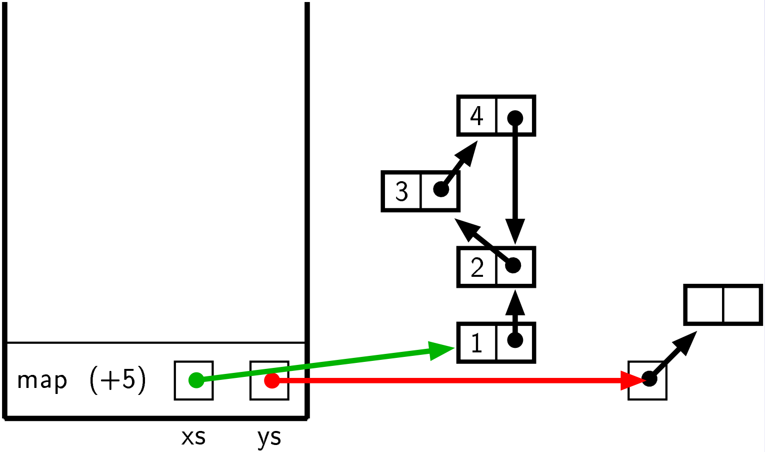
Cyclical Functional Programming

Example: map



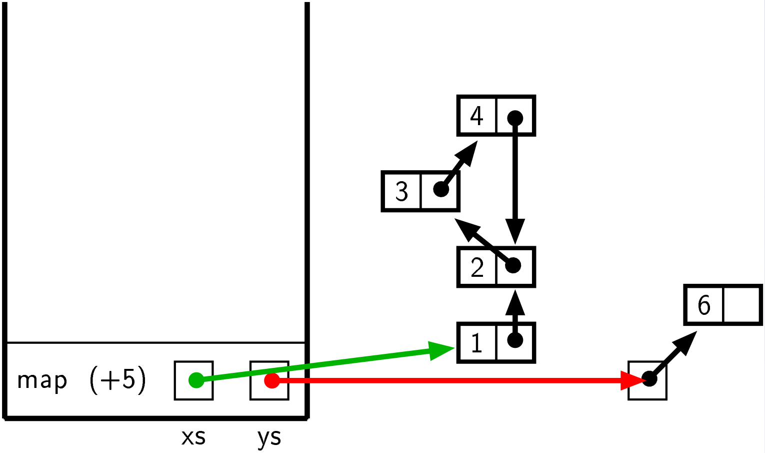
Cyclical Functional Programming

Example: map



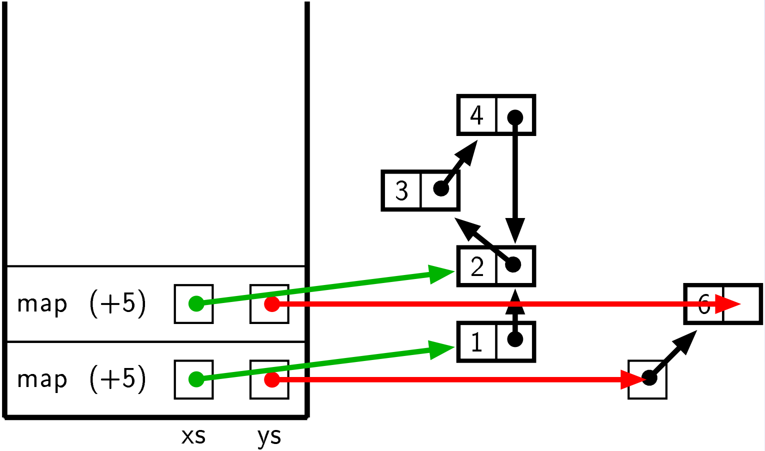
Cyclical Functional Programming

Example: map



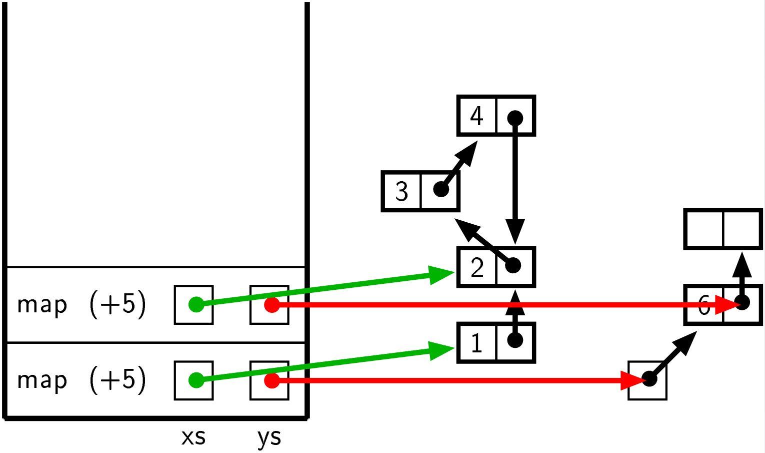
Cyclical Functional Programming

Example: map



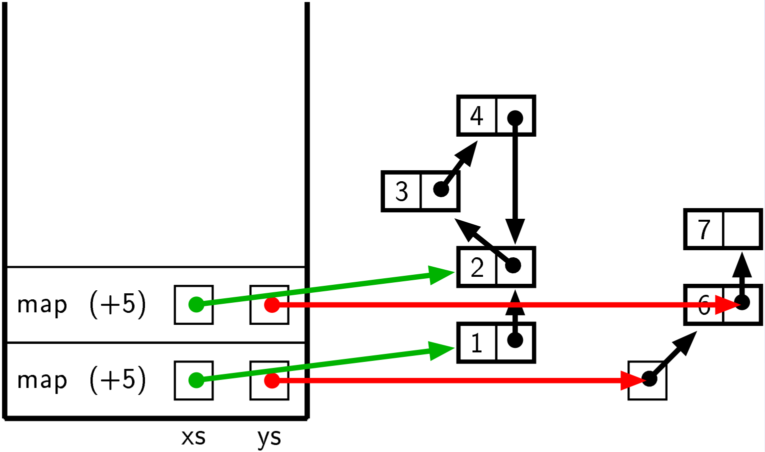
Cyclical Functional Programming

Example: map



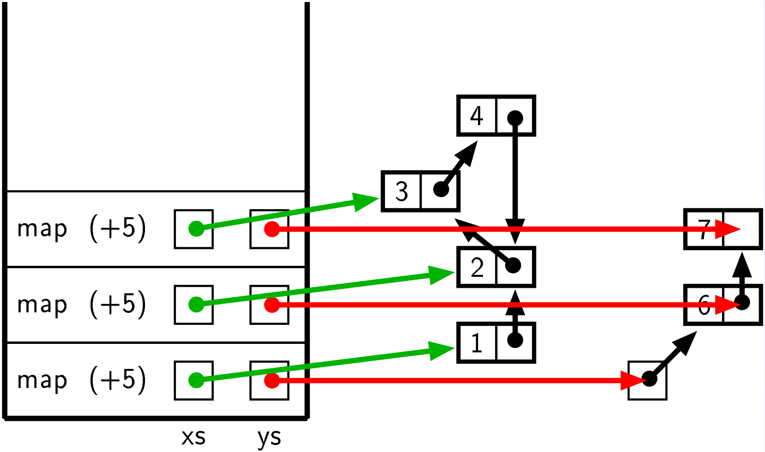
Cyclical Functional Programming

Example: map



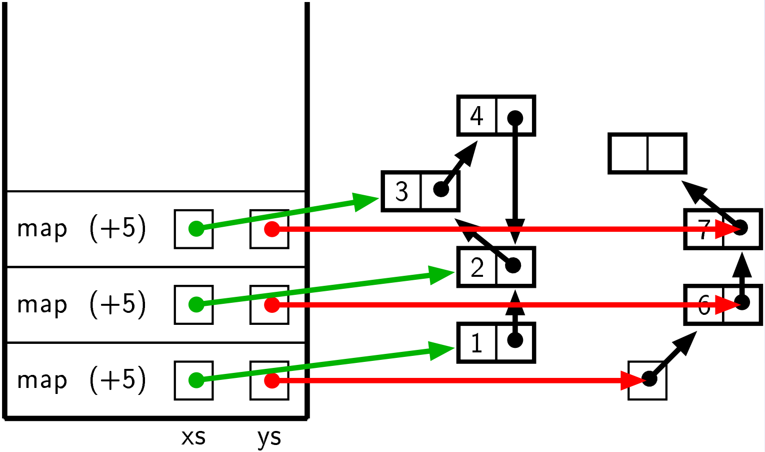
Cyclical Functional Programming

Example: map



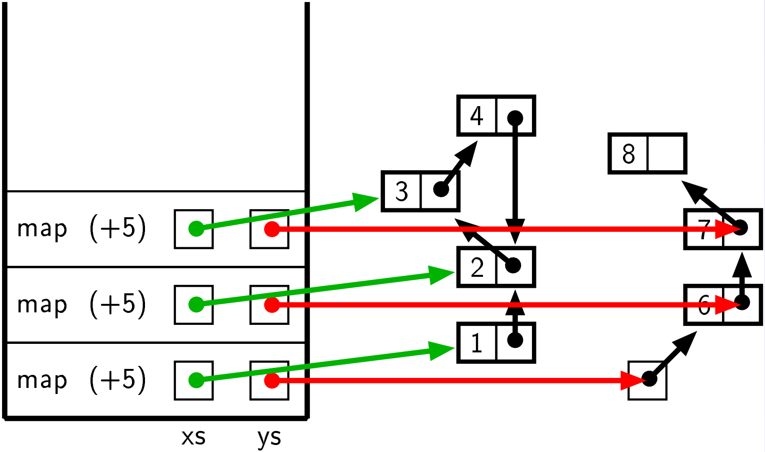
Cyclical Functional Programming

Example: map



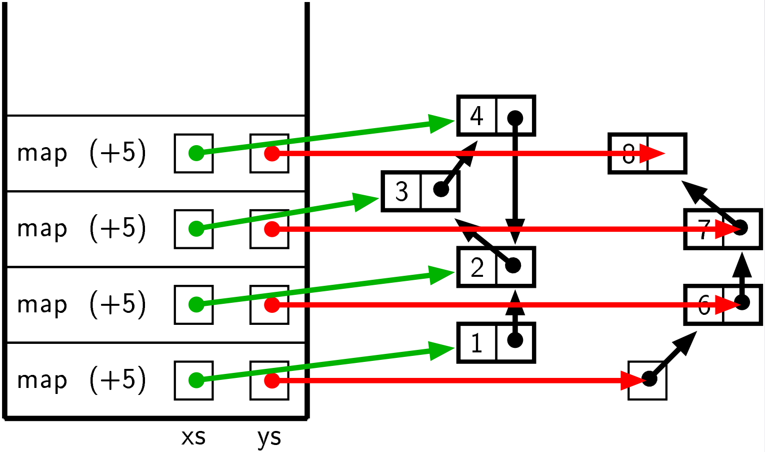
Cyclical Functional Programming

Example: map



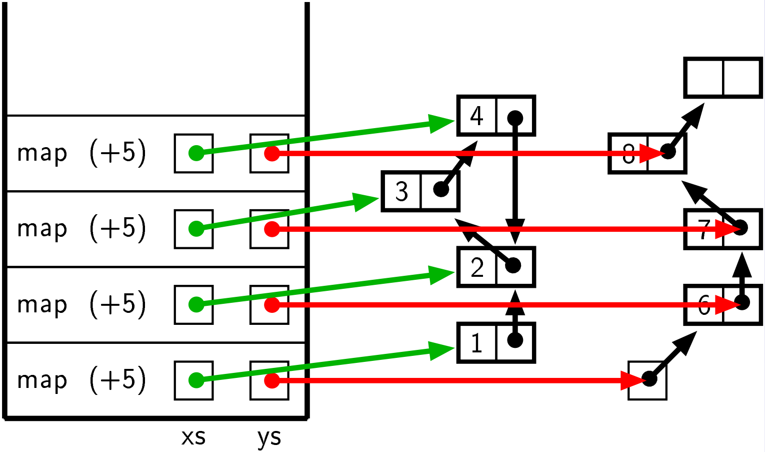
Cyclical Functional Programming

Example: map



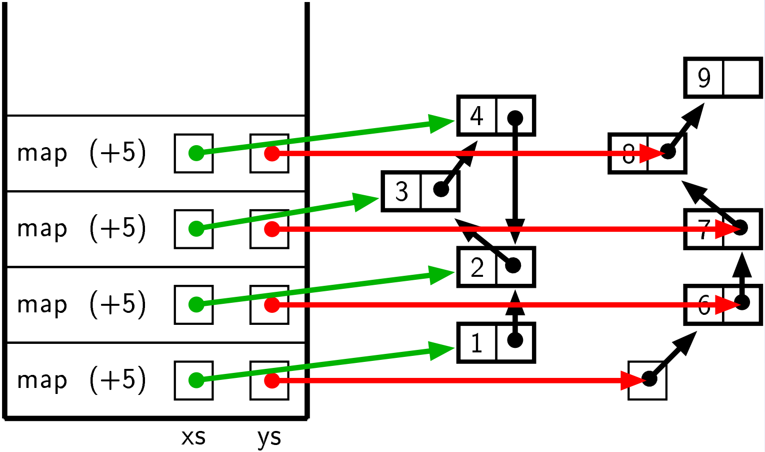
Cyclical Functional Programming

Example: map



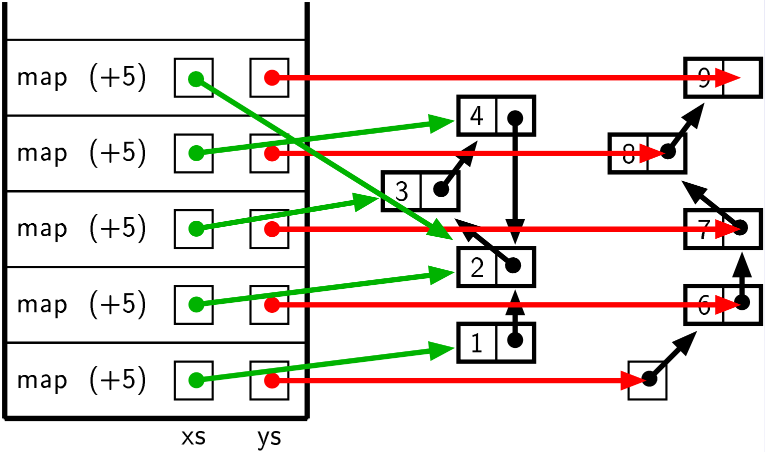
Cyclical Functional Programming

Example: map



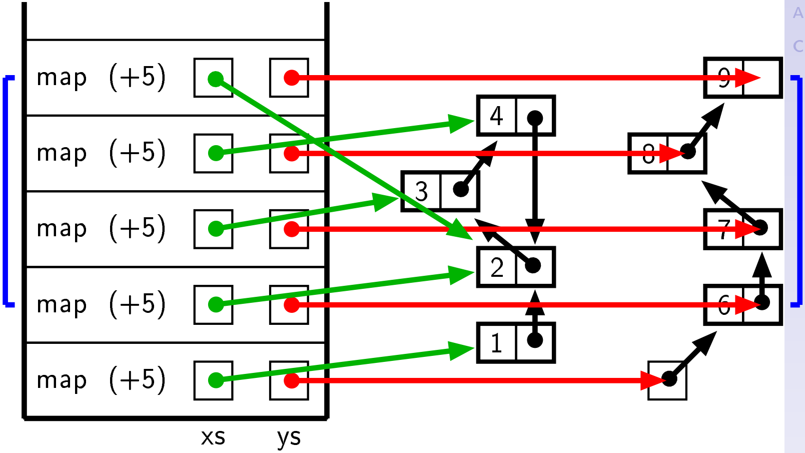
Cyclical Functional Programming

Example: map



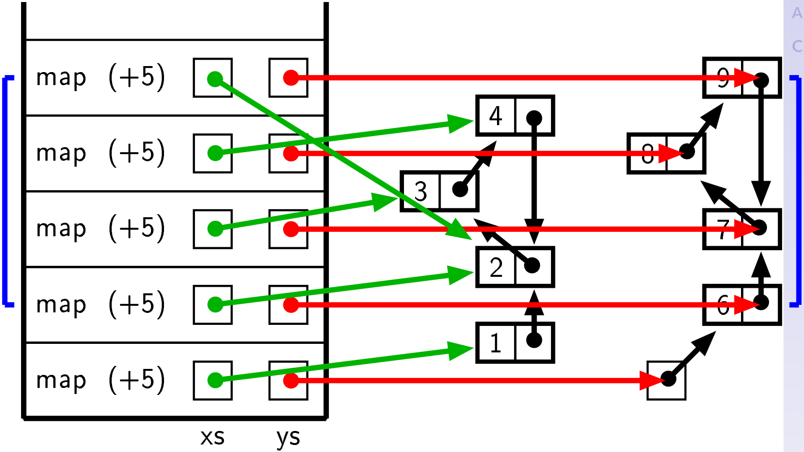
Cyclical Functional Programming

Example: map



Cyclical Functional Programming

Example: map



Implementation of CFP
Programming System

The *Malice* System

- virtual machine, supports
  - destination passing & higher-order functions
  - cycle detection & handling (ditto)
- interpreter & aot compiler

Implementation of CFP
Applications

Cyclic Lists generalization of familiar list algorithms

- *insert*, *delete*, *length*
- *map*, *filter*, *quicksort*

Rationals generalization of school math algorithms

- arithmetics, order
- period detection

Algebraic Subtyping *vtable*-like dynamic encoding

- static recursive subtype checking
- dynamic (duck typing) access

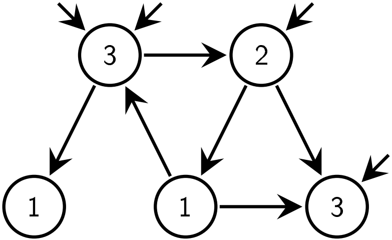
Real-World Applications

Wanted:

Reference-Counting Algorithm

Deriving an Algorithm

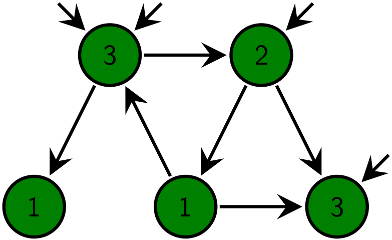
MLW At Work



Initial Situation

Deriving an Algorithm

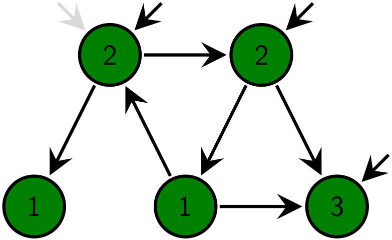
MLW At Work



Initial Situation

Deriving an Algorithm

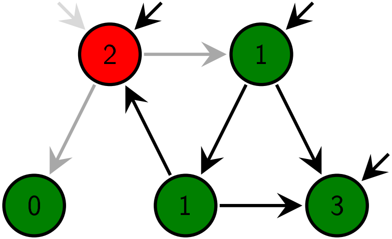
MLW At Work



Delete #1: Reachable, sharing

Deriving an Algorithm

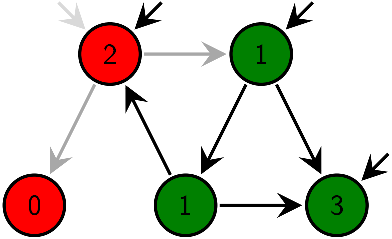
MLW At Work



Delete #1: Reachable, sharing

Deriving an Algorithm

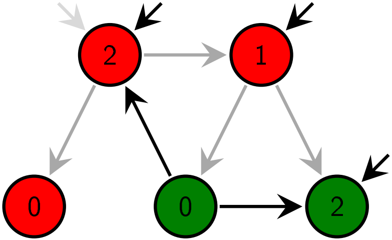
MLW At Work



Delete #1: Reachable, sharing

Deriving an Algorithm

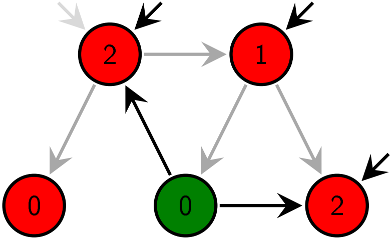
MLW At Work



Delete #1: Reachable, sharing

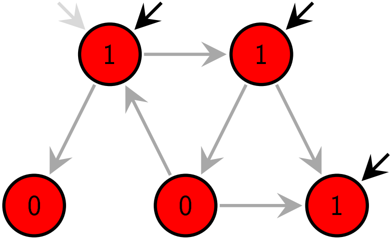
Deriving an Algorithm

MLW At Work



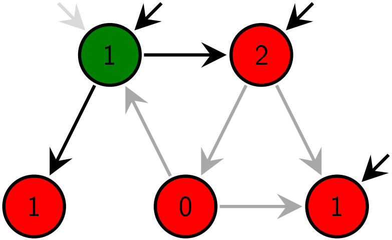
Delete #1: Reachable, sharing

Deriving an Algorithm
MLW At Work



Delete #1: Reachable, sharing

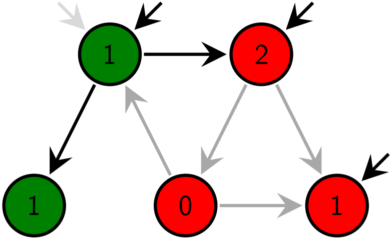
Deriving an Algorithm
MLW At Work



Delete #1: Reachable, sharing

Deriving an Algorithm

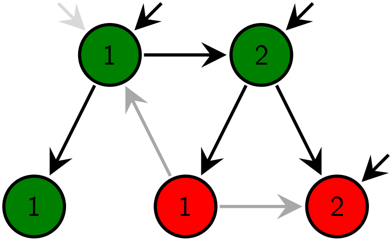
MLW At Work



Delete #1: Reachable, sharing

Deriving an Algorithm

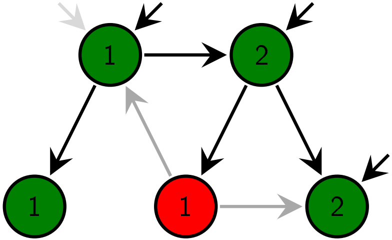
MLW At Work



Delete #1: Reachable, sharing

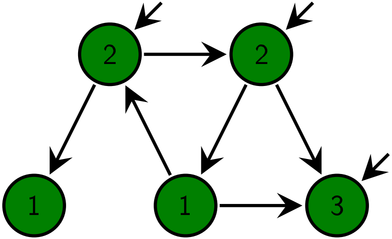
Deriving an Algorithm

MLW At Work



Delete #1: Reachable, sharing

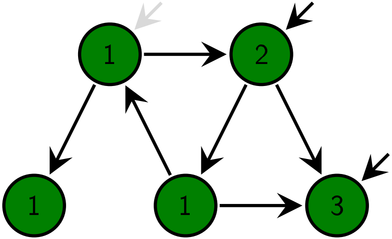
Deriving an Algorithm
MLW At Work



Delete #1: Reachable, sharing

Deriving an Algorithm

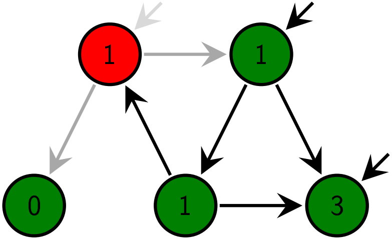
MLW At Work



Delete #2: Reachable, different way

Deriving an Algorithm

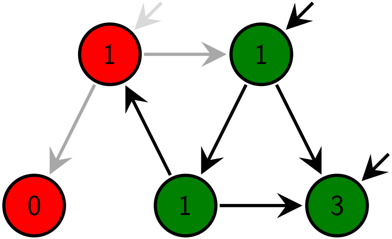
MLW At Work



Delete #2: Reachable, different way

Deriving an Algorithm

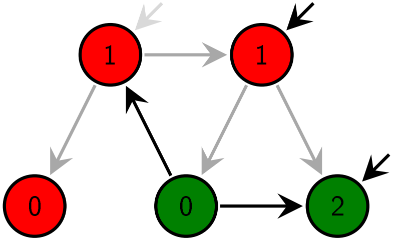
MLW At Work



Delete #2: Reachable, different way

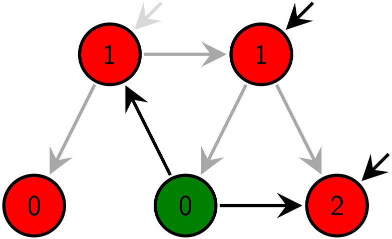
Deriving an Algorithm

MLW At Work



Delete #2: Reachable, different way

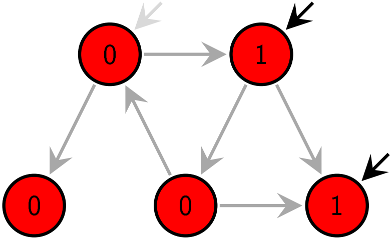
Deriving an Algorithm
MLW At Work



Delete #2: Reachable, different way

Deriving an Algorithm

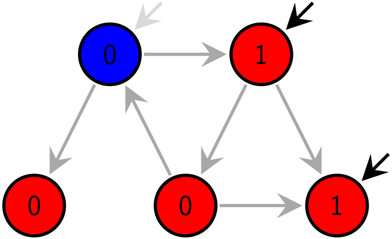
MLW At Work



Delete #2: Reachable, different way

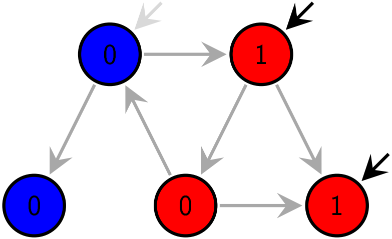
Deriving an Algorithm

MLW At Work



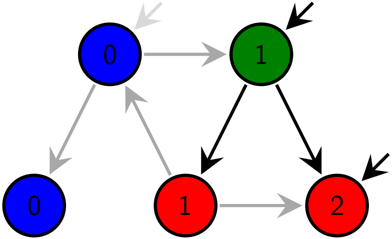
Delete #2: Reachable, different way

Deriving an Algorithm
MLW At Work



Delete #2: Reachable, different way

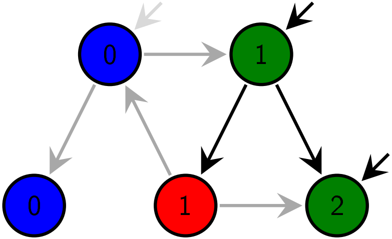
Deriving an Algorithm
MLW At Work



Delete #2: Reachable, different way

Deriving an Algorithm

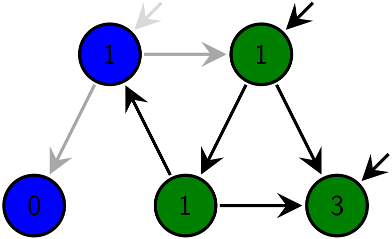
MLW At Work



Delete #2: Reachable, different way

Deriving an Algorithm

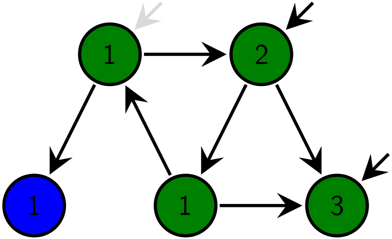
MLW At Work



Delete #2: Reachable, different way

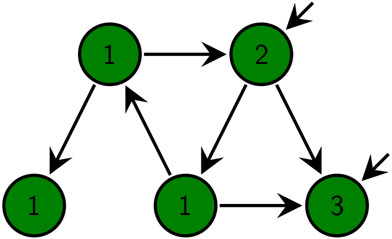
Deriving an Algorithm

MLW At Work



Delete #2: Reachable, different way

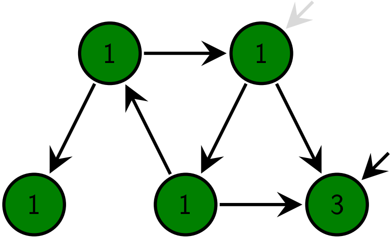
Deriving an Algorithm
MLW At Work



Delete #2: Reachable, different way

Deriving an Algorithm

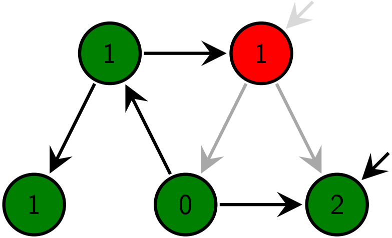
MLW At Work



Delete #3: Unreachable

Deriving an Algorithm

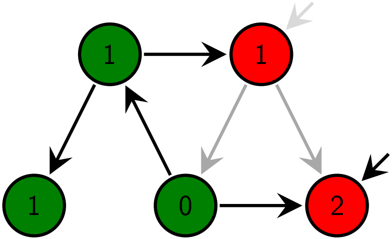
MLW At Work



Delete #3: Unreachable

Deriving an Algorithm

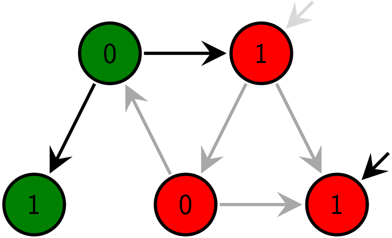
MLW At Work



Delete #3: Unreachable

Deriving an Algorithm

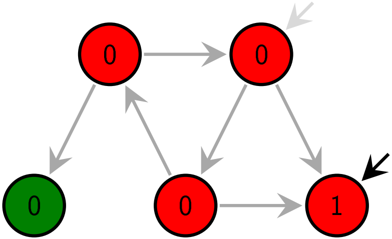
MLW At Work



Delete #3: Unreachable

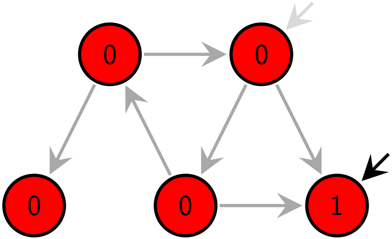
Deriving an Algorithm

MLW At Work



Delete #3: Unreachable

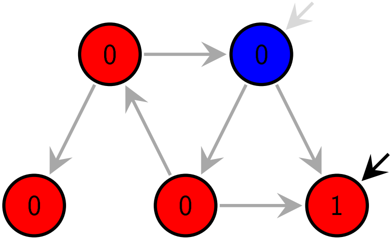
Deriving an Algorithm
MLW At Work



Delete #3: Unreachable

Deriving an Algorithm

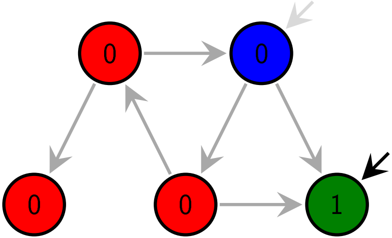
MLW At Work



Delete #3: Unreachable

Deriving an Algorithm

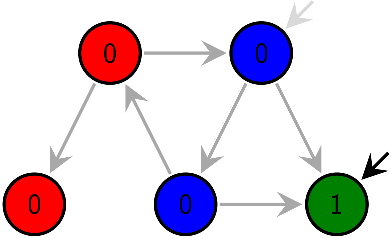
MLW At Work



Delete #3: Unreachable

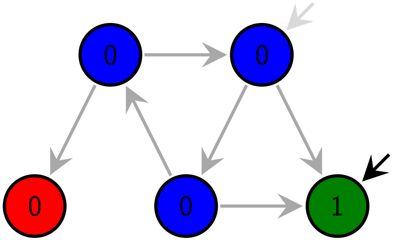
Deriving an Algorithm

MLW At Work



Delete #3: Unreachable

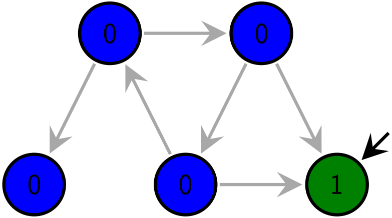
Deriving an Algorithm
MLW At Work



Delete #3: Unreachable

Deriving an Algorithm

MLW At Work



Delete #3: Unreachable

Deriving an Algorithm
Idea #1

Idea #1 — Component Analysis

- consider strongly connected components
  - all cells in an SCC die together
  - only inter-SCC references count for reachability
- maintain separate inter/intra counts
  - inter count drops to zero $\Longrightarrow$ garbage

Problems

- inter-SCC references may point to distant members (#2)
- maintaining inter/intra classification is hard

Deriving an Algorithm
Idea #2

Idea #2 — Edge Coloring

- approximate inter/intra by maintained strong/weak partition, à la Brownbridge
    - strong $\simeq$ inter
- choose invariants that are cheap to maintain
    1. no cycle is entirely strong
    2. weak in + strong out $\Longrightarrow$ strong in
- maintain separate strong/weak counts
    - strong count drops to zero $\Longrightarrow$ garbage

Deriving an Algorithm

Invariants

How To Maintain Invariant 1

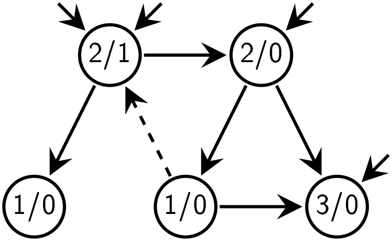
- mutator creates references in three ways only
  1. root $\implies$ strong
  2. constructor argument $\implies$ strong
  3. ditto $\implies$ weak

How To Maintain Invariant 2

- deleting (strong) references may create violations
  - weak in & no strong in & strong out
- rectify my making strong out references weak
  - propagate

Deriving an Algorithm

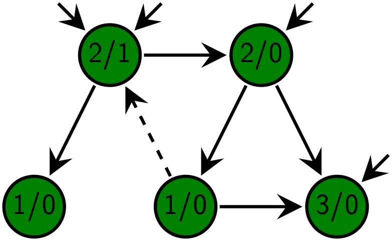
EC At Work



Initial Situation

Deriving an Algorithm

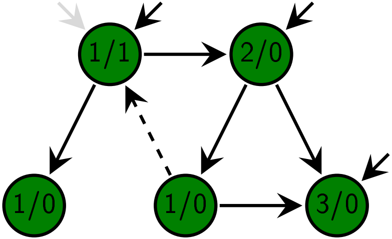
EC At Work



Initial Situation

Deriving an Algorithm

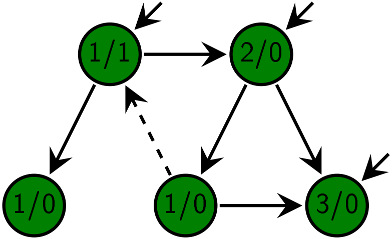
EC At Work



Delete #1: Reachable, sharing

Deriving an Algorithm

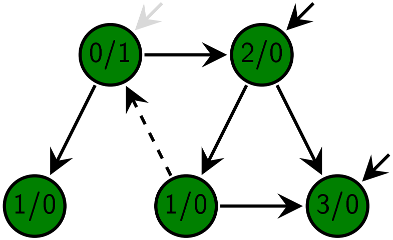
EC At Work



Delete #1: Reachable, sharing

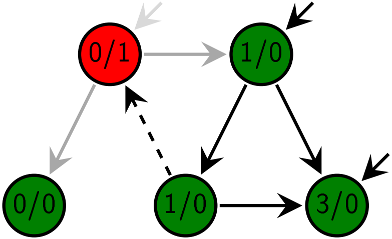
Deriving an Algorithm

EC At Work



Delete #2: Reachable, different way

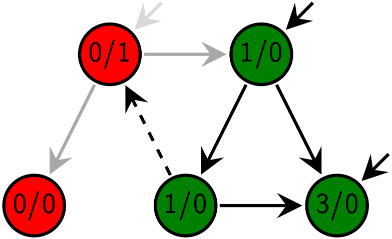
Deriving an Algorithm
EC At Work



Delete #2: Reachable, different way

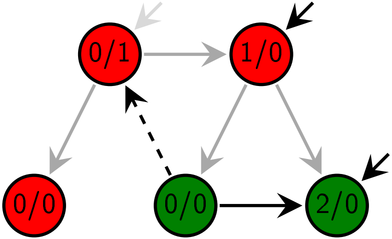
Deriving an Algorithm

EC At Work



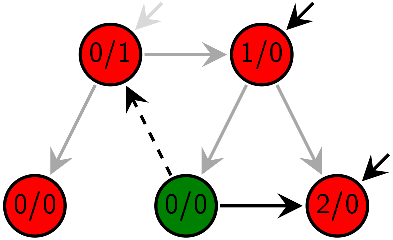
Delete #2: Reachable, different way

Deriving an Algorithm
EC At Work



Delete #2: Reachable, different way

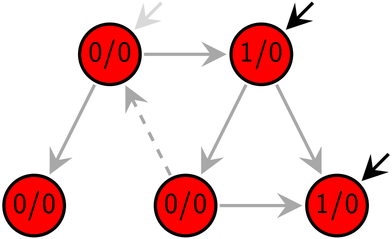
Deriving an Algorithm
EC At Work



Delete #2: Reachable, different way

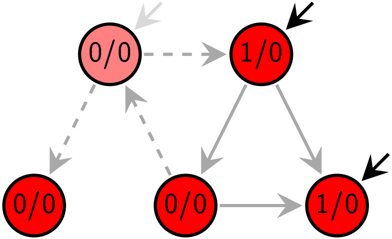
Deriving an Algorithm

EC At Work



Delete #2: Reachable, different way

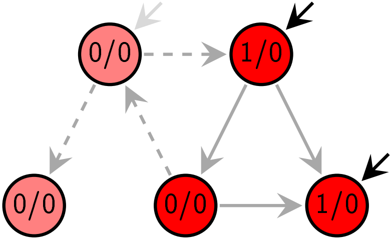
Deriving an Algorithm
EC At Work



Delete #2: Reachable, different way

Deriving an Algorithm

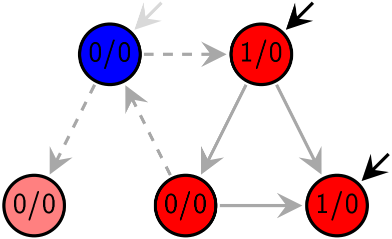
EC At Work



Delete #2: Reachable, different way

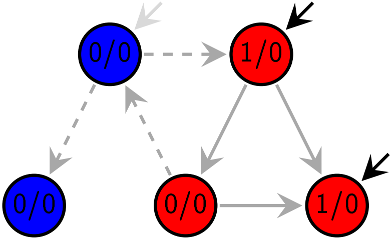
Deriving an Algorithm

EC At Work



Delete #2: Reachable, different way

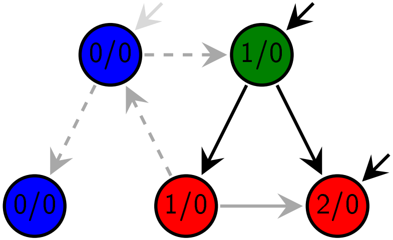
Deriving an Algorithm
EC At Work



Delete #2: Reachable, different way

Deriving an Algorithm

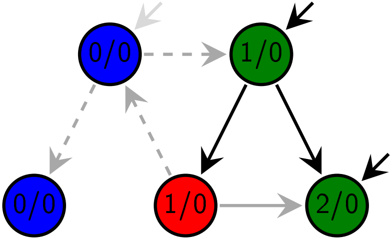
EC At Work



Delete #2: Reachable, different way

Deriving an Algorithm

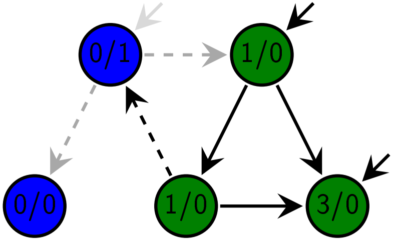
EC At Work



Delete #2: Reachable, different way

Deriving an Algorithm

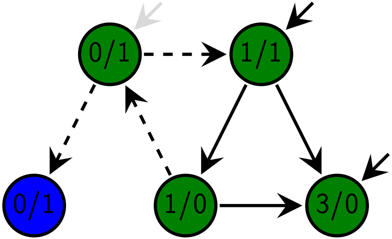
EC At Work



Delete #2: Reachable, different way

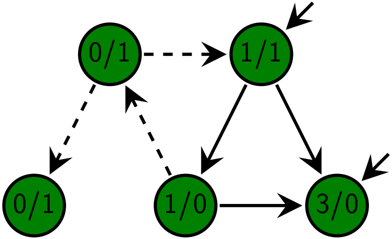
Deriving an Algorithm

EC At Work



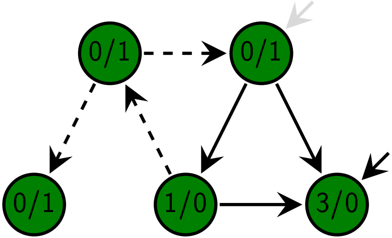
Delete #2: Reachable, different way

Deriving an Algorithm
EC At Work



Delete #2: Reachable, different way

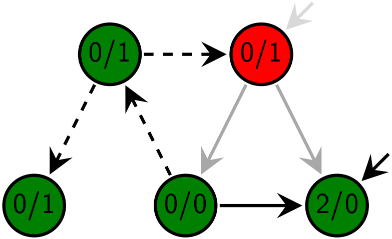
Deriving an Algorithm
EC At Work



Delete #3: Unreachable

Deriving an Algorithm

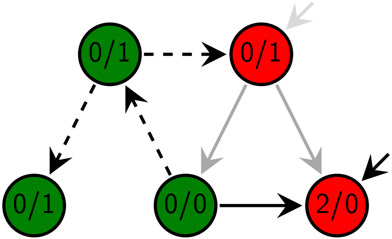
EC At Work



Delete #3: Unreachable

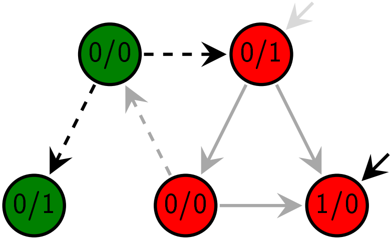
Deriving an Algorithm

EC At Work



Delete #3: Unreachable

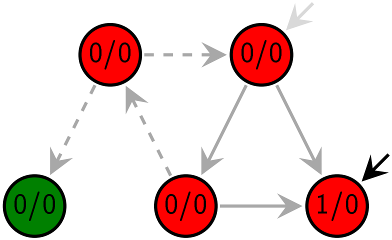
Deriving an Algorithm
EC At Work



Delete #3: Unreachable

Deriving an Algorithm

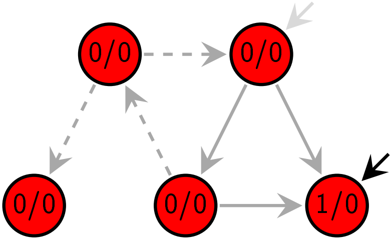
EC At Work



Delete #3: Unreachable

Deriving an Algorithm

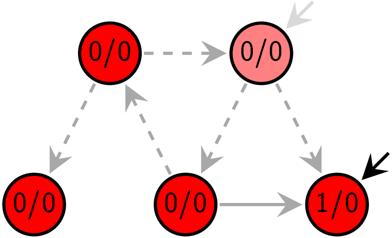
EC At Work



Delete #3: Unreachable

Deriving an Algorithm

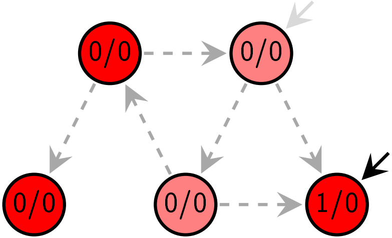
EC At Work



Delete #3: Unreachable

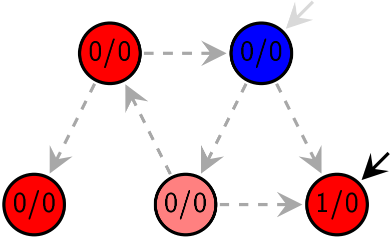
Deriving an Algorithm

EC At Work



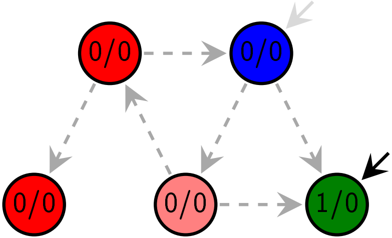
Delete #3: Unreachable

Deriving an Algorithm
EC At Work



Delete #3: Unreachable

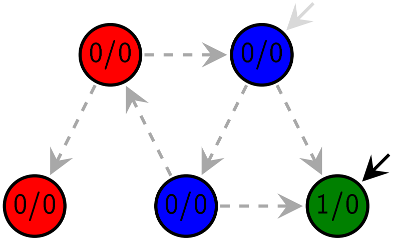
Deriving an Algorithm
EC At Work



Delete #3: Unreachable

Deriving an Algorithm

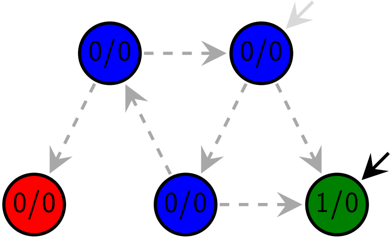
EC At Work



Delete #3: Unreachable

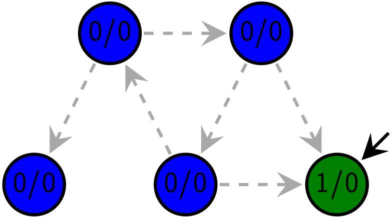
Deriving an Algorithm

EC At Work



Delete #3: Unreachable

Deriving an Algorithm
EC At Work



Delete #3: Unreachable

Deriving an Algorithm

Independence

Independence Thesis

- edge coloring is independent of other optimizations/heuristics
- "push-out" should be possible

Case Study: Deletion Queue

- queue zero-count cell to defer speculation
- process one entry $\Longrightarrow$ others redundant
- combined with edge coloring in three hours

Evaluation

Measurement

What And How To Measure

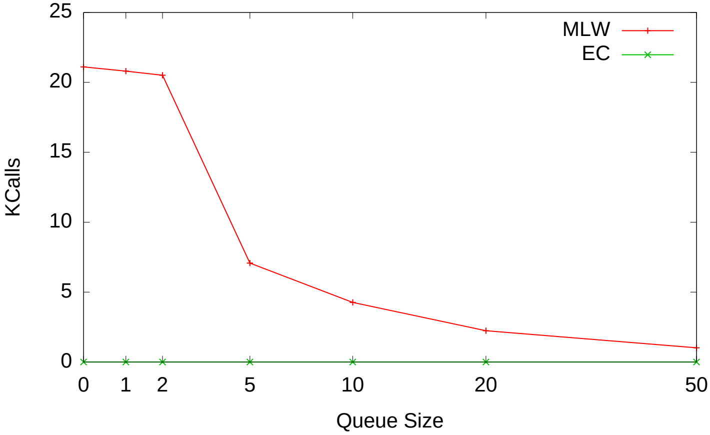
- no real-world implementation/application exists
- cannot measure runtime

Extend Idea From Original MLW Paper

- simulate & count traversal operations
- single (cyclical) algorithm, varying amount of cycles
- without queue & with different sizes
- additionally measure green–blue ratio (*overkill*)

Evaluation

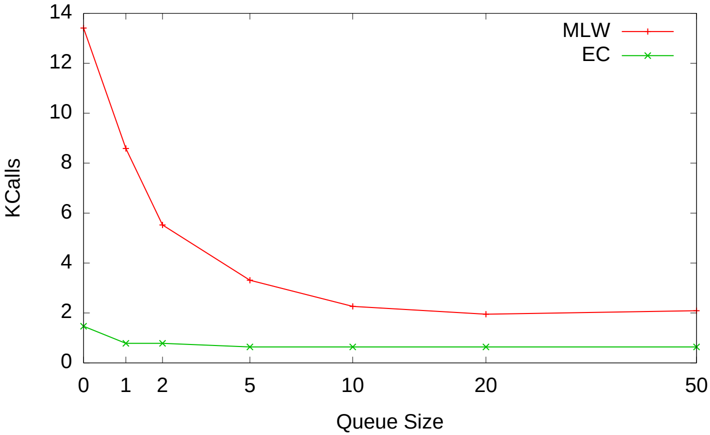
Charts: No Cycles in Input



```
quicksort l where
    l = [0..99]
```

Evaluation

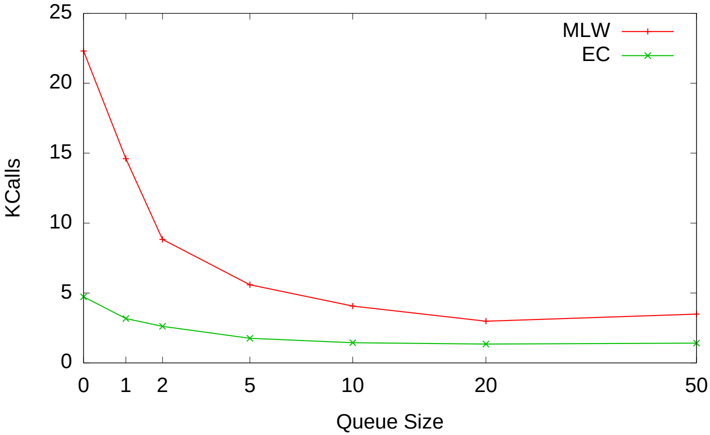
Charts: Small Cycles in Input



```
quicksort l where
    l = [0..9] ++ (cycle [8,9])
```

Evaluation

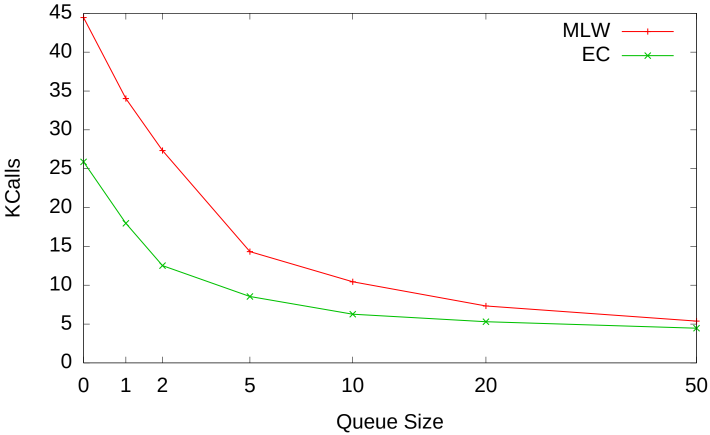
Charts: Medium Cycles in Input



```
quicksort l where
    l = [0..9] ++ (cycle [5..9])
```

Evaluation

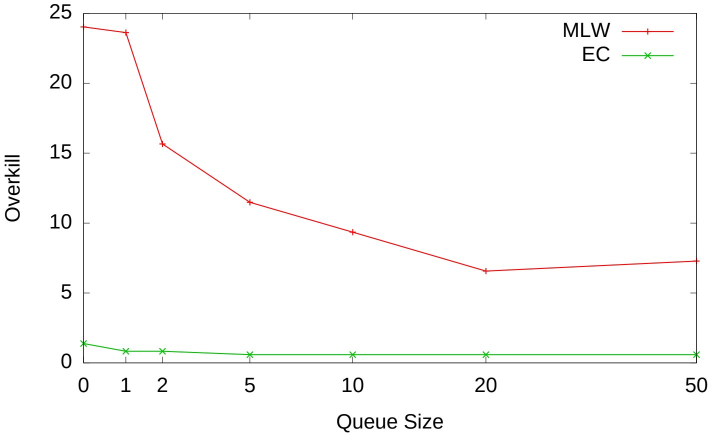
Charts: Large Cycles in Input



```
quicksort l where
    l = cycle [0..14]
```

Evaluation

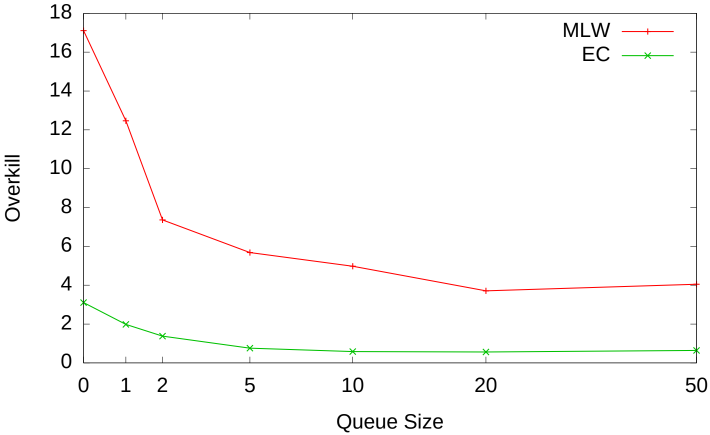
Charts: Small Cycles in Input



```
quicksort l where
    l = [0..9] ++ (cycle [8,9])
```

Evaluation

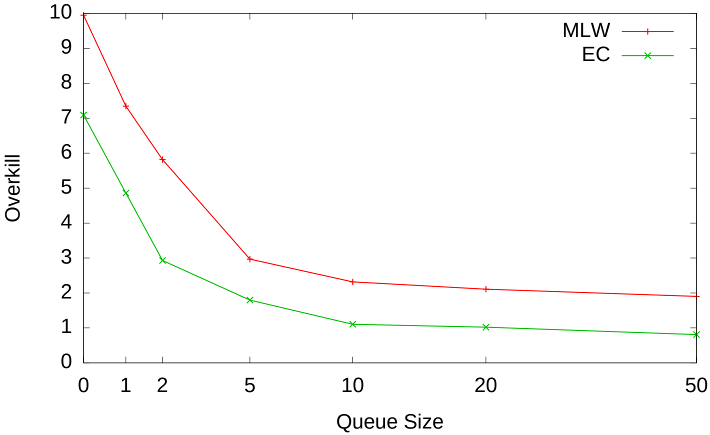
Charts: Medium Cycles in Input



```
quicksort l where
    l = [0..9] ++ (cycle [5..9])
```

Evaluation

Charts: Large Cycles in Input



```
quicksort l where
    l = cycle [0..14]
```

Evaluation

Special Cases

Special Cases Come For Free!

Acyclic Input	no weak references
Acyclic Types	as above, statically
Global Data	persistent strong references

- no speculation in either case

Conclusions

Conclusion

Summary

- CFP paradigm creates cycles in controlled way
- EC algorithm exploits control to speed up MLW
- 1–bit edge coloring, simple maintenance
- full algorithm & proof in paper
- combines with (some) other improvements
- applies where maintenance assumptions hold

Conclusion

Open Questions

- undo strong→weak conversion?
- more independent optimizations/heuristics?
- applicable/efficient beyond toy examples?
  - real-world applications of CFP?
  - other applications with same mutator behavior?

Answers Welcome

- specification & proof in the paper
- Java demo implementation available