Verification of Schorr-Waite

Richard Bubel

June 15, 2007

Overview

Algorithm characteristics:

- Graph-Marking algorithm designed by H.Schorr and W.Waite

Overview

Algorithm characteristics:

- Graph-Marking algorithm designed by H.Schorr and W.Waite
- Goal: memory efficient garbage collector for LISP

Overview

Algorithm characteristics:

- Graph-Marking algorithm designed by H.Schorr and W.Waite
- Goal: memory efficient garbage collector for LISP
- Only three additional pointers required at run-time (backtracking path encoded in graph structure)

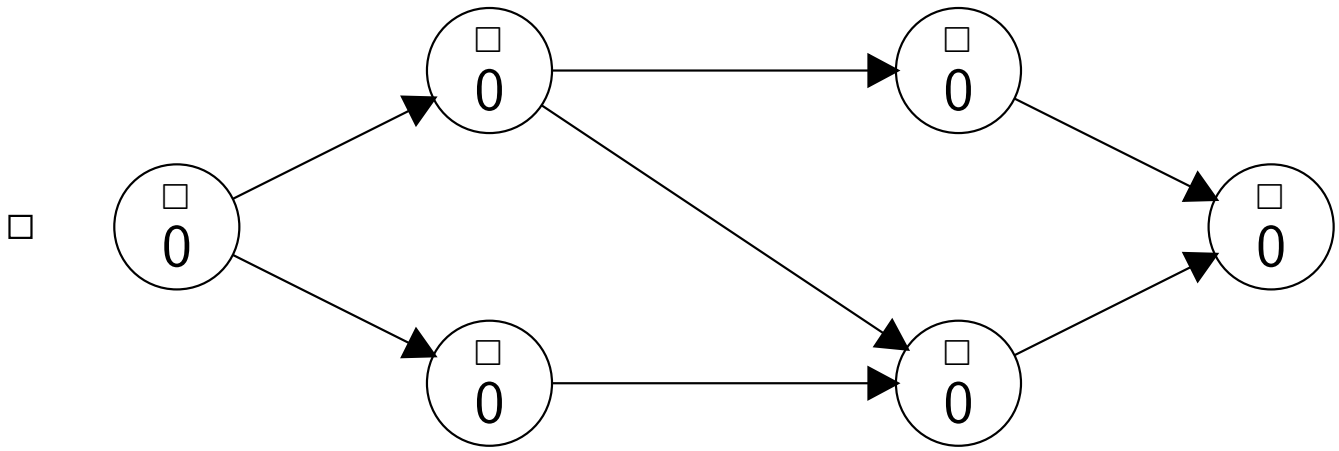
Overview

Algorithm characteristics:

- Graph-Marking algorithm designed by H.Schorr and W.Waite
- Goal: memory efficient garbage collector for LISP
- Only three additional pointers required at run-time (backtracking path encoded in graph structure)

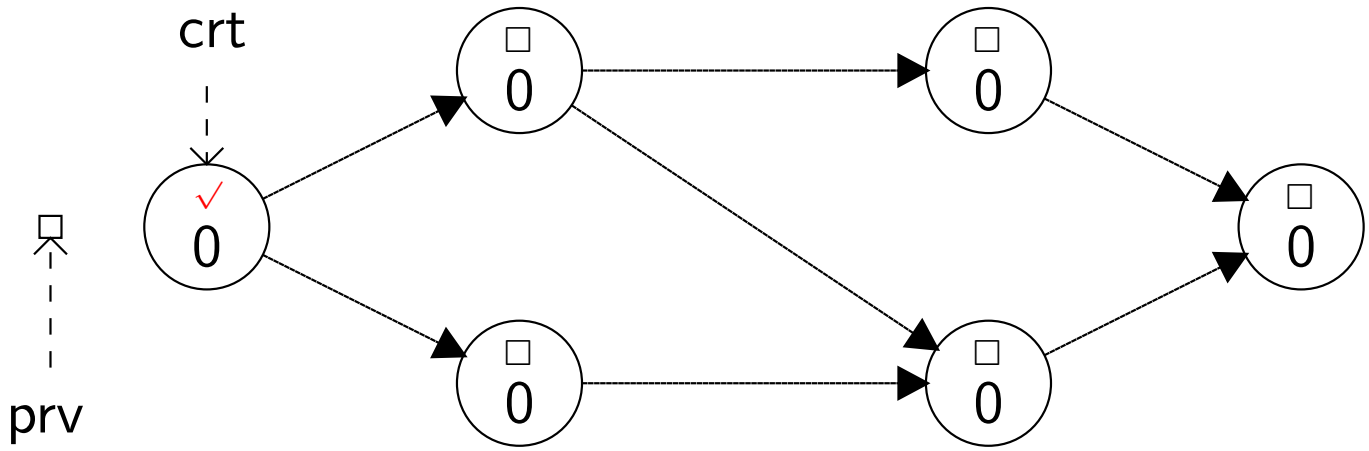
Benchmark for verification methods dealing with linked structures

Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

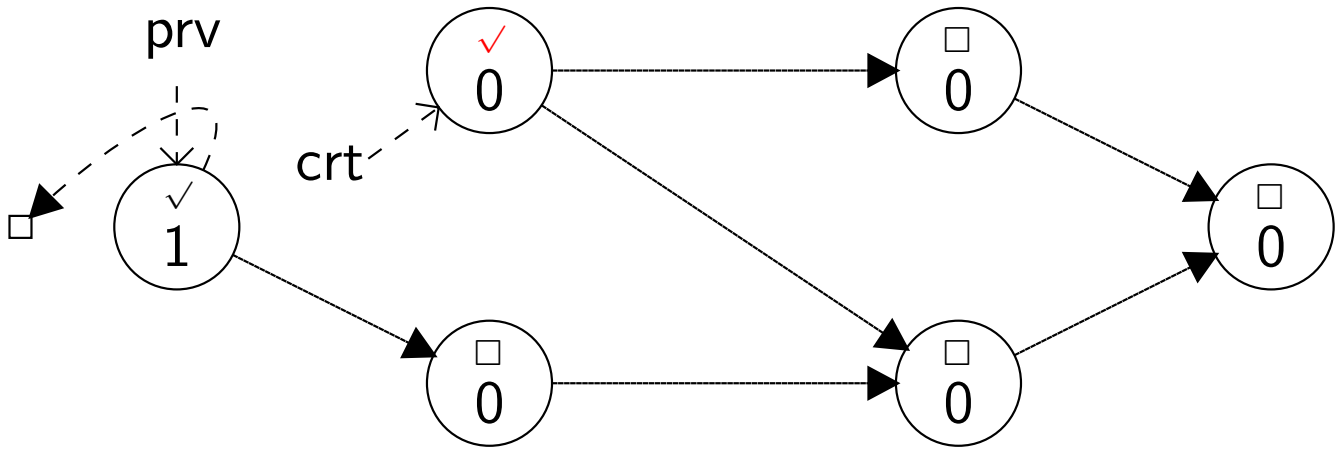
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
idx = crt.getIndex();
tmpChild = crt.getChild(idx);
if (!tmpChild.isMarked()) {
  crt.setChild(idx, prv);
  prv = crt;
  crt.incIndex();
  crt = tmpChild;
  crt.setMark(true);
} else { crt.incIndex(); }
```

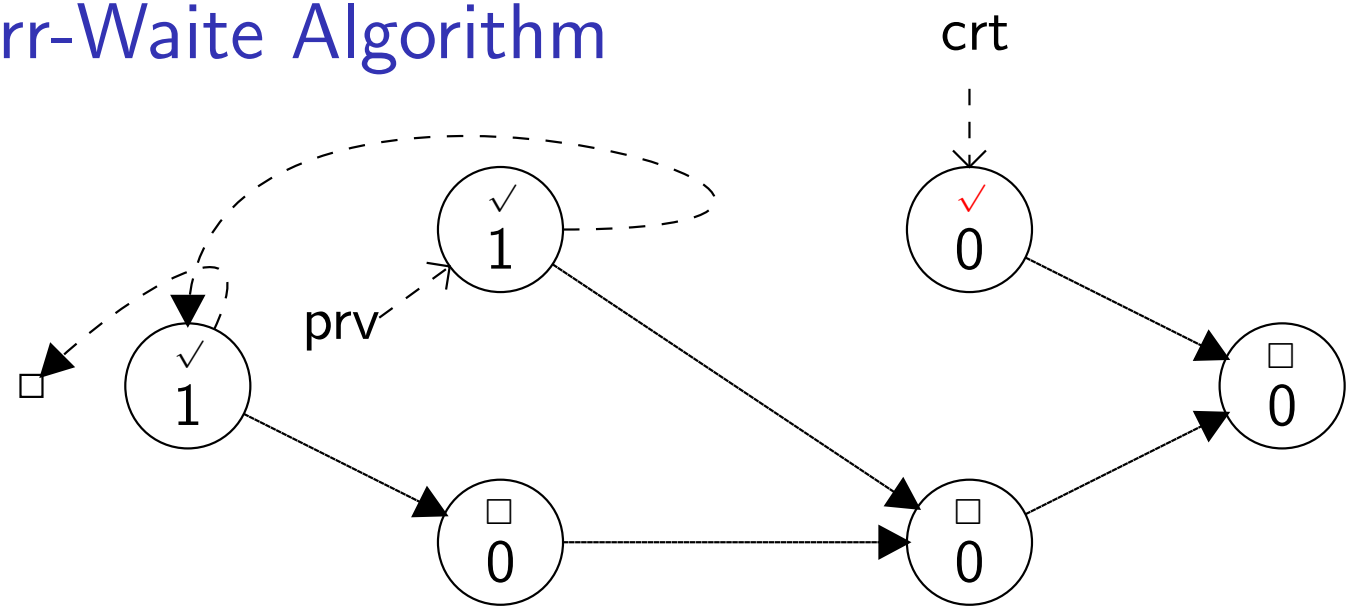
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
idx = crt.getIndex();
tmpChild = crt.getChild(idx);
if (!tmpChild.isMarked()) {
  crt.setChild(idx, prv);
  prv = crt;
  crt.incIndex();
  crt = tmpChild;
  crt.setMark(true);
} else { crt.incIndex(); }
```

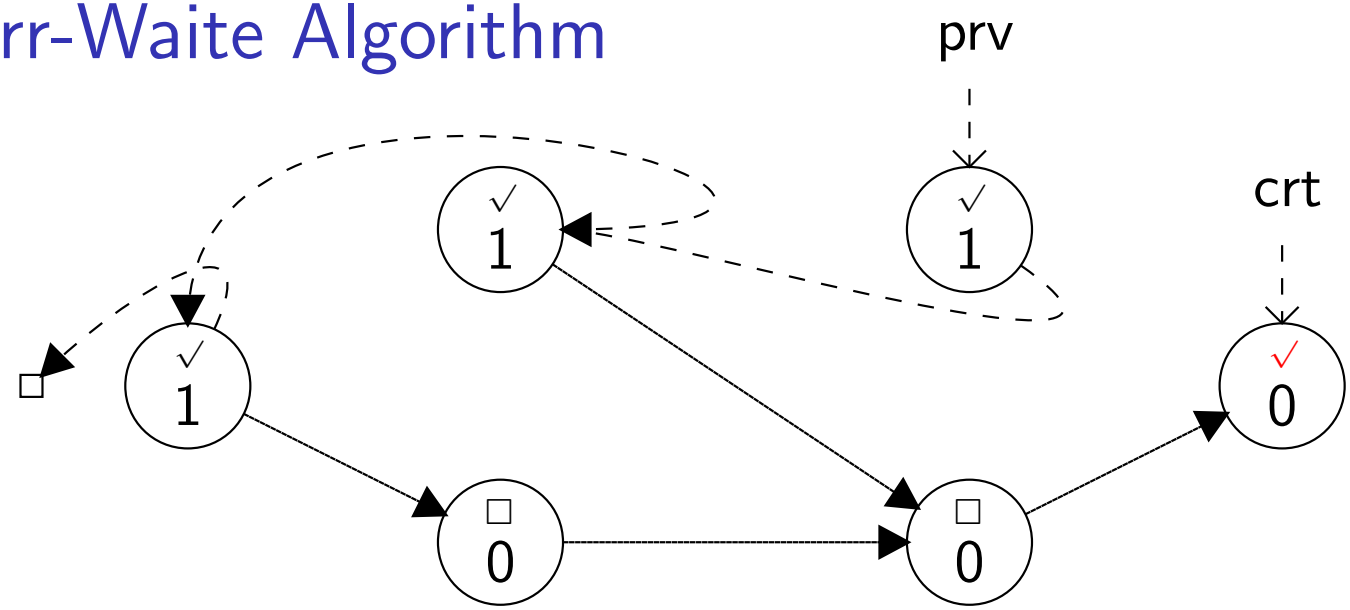
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
idx = crt.getIndex();
tmpChild = crt.getChild(idx);
if (!tmpChild.isMarked()) {
  crt.setChild(idx, prv);
  prv = crt;
  crt.incIndex();
  crt = tmpChild;
  crt.setMark(true);
} else { crt.incIndex(); }
```

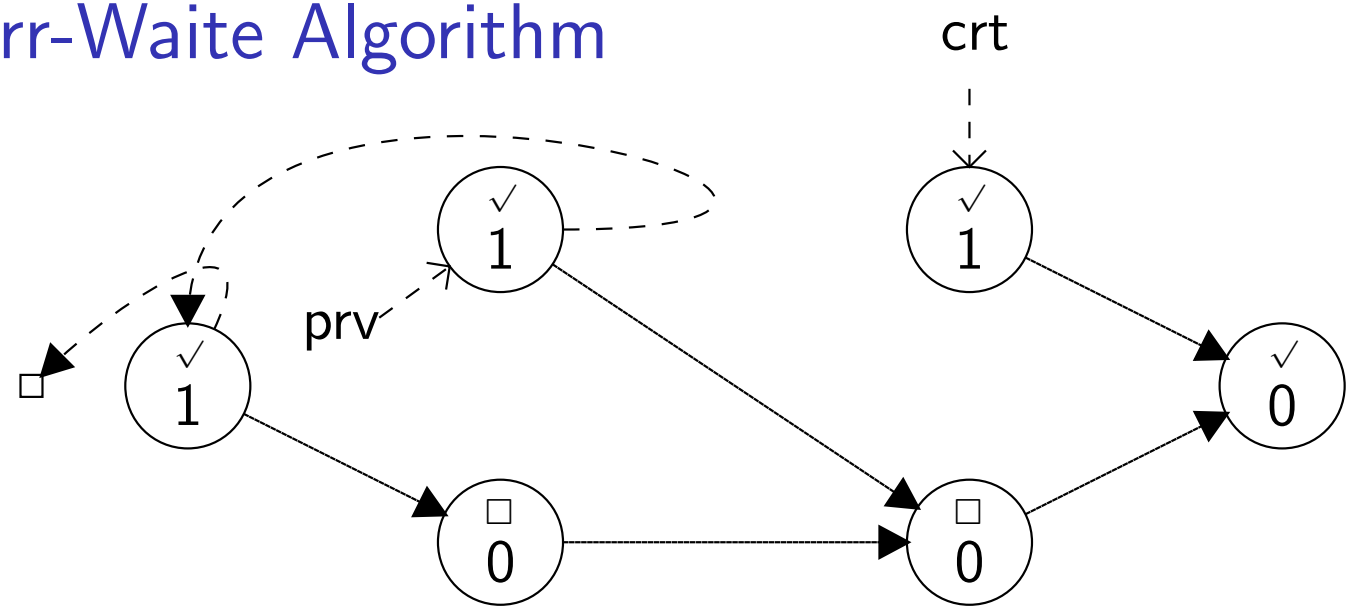
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
ref2restore =
       prv.getIndex() - 1;
tmpChild =
       prv.getChild(ref2restore);
prv.setChild(ref2restore, crt);
crt = prv;
prv = tmpChild;
```

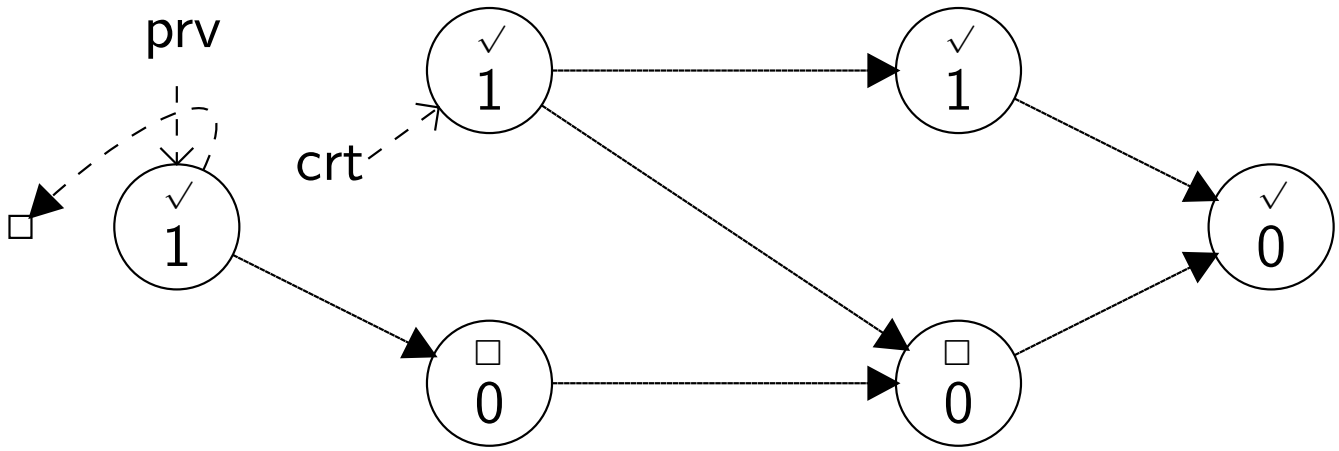
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
ref2restore =
       prv.getIndex() - 1;
tmpChild =
       prv.getChild(ref2restore);
prv.setChild(ref2restore, crt);
crt = prv;
prv = tmpChild;
```

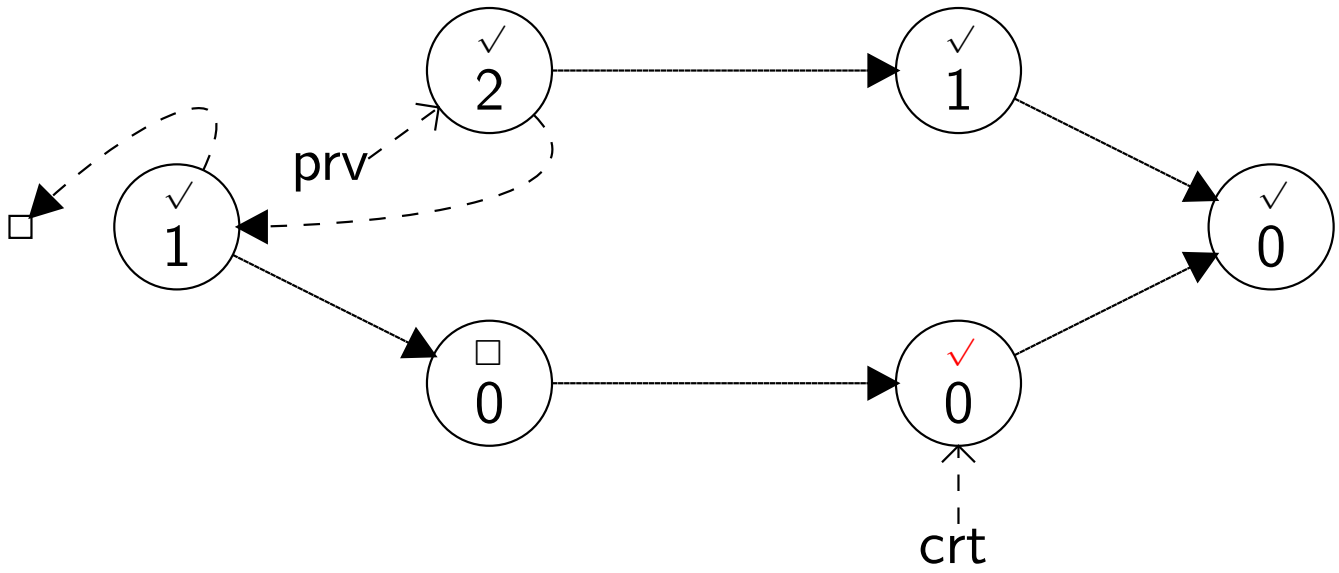
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
idx = crt.getIndex();
tmpChild = crt.getChild(idx);
if (!tmpChild.isMarked()) {
  crt.setChild(idx, prv);
  prv = crt;
  crt.incIndex();
  crt = tmpChild;
  crt.setMark(true);
} else { crt.incIndex(); }
```

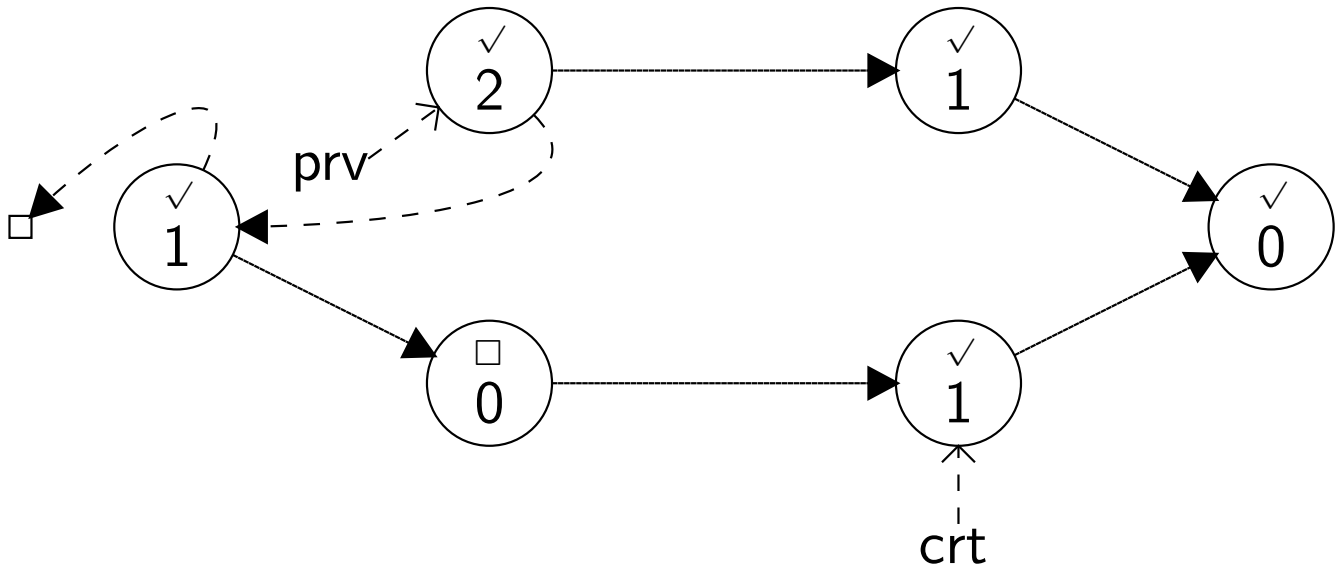
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
idx = crt.getIndex();
tmpChild = crt.getChild(idx);
if (!tmpChild.isMarked()) {
  crt.setChild(idx, prv);
  prv = crt;
  crt.incIndex();
  crt = tmpChild;
  crt.setMark(true);
} else { crt.incIndex(); }
```

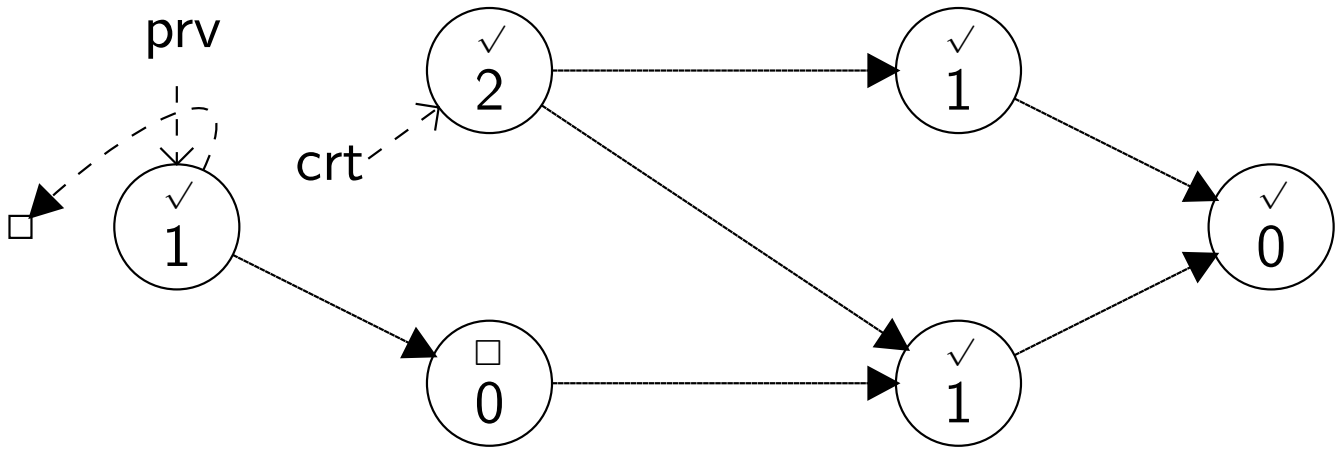
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
ref2restore =
       prv.getIndex() - 1;
tmpChild =
       prv.getChild(ref2restore);
prv.setChild(ref2restore, crt);
crt = prv;
prv = tmpChild;
```

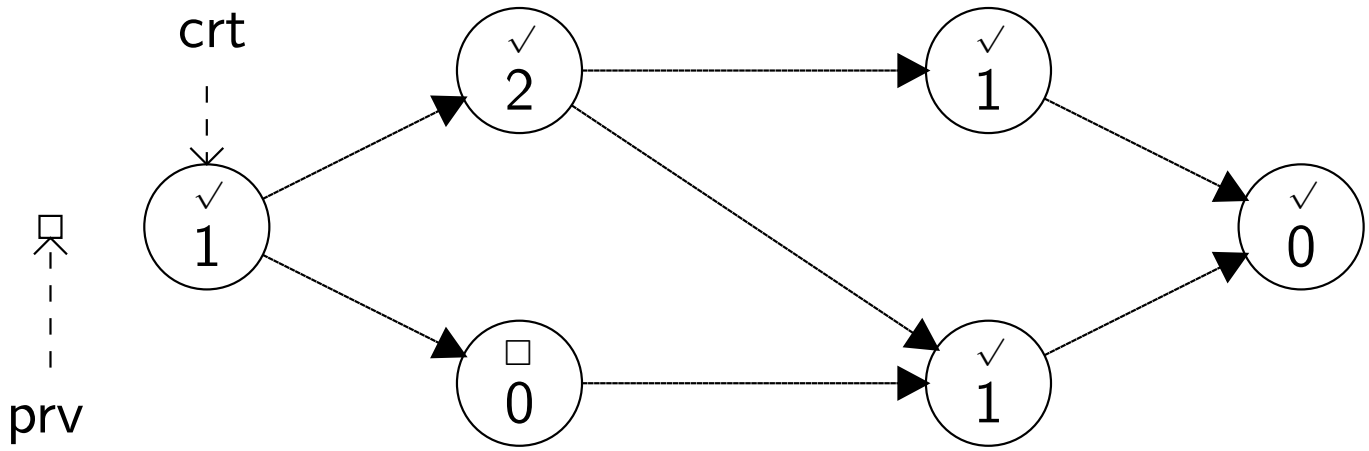
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
ref2restore =
       prv.getIndex() - 1;
tmpChild =
       prv.getChild(ref2restore);
prv.setChild(ref2restore, crt);
crt = prv;
prv = tmpChild;
```

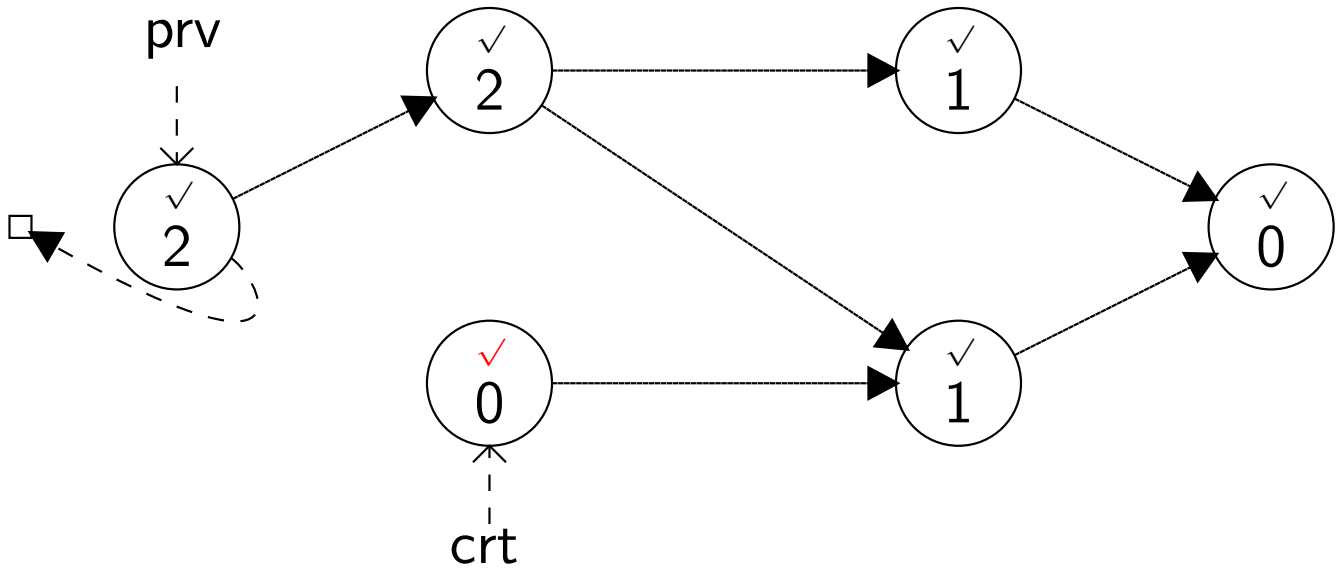
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
idx = crt.getIndex();
tmpChild = crt.getChild(idx);
if (!tmpChild.isMarked()) {
  crt.setChild(idx, prv);
  prv = crt;
  crt.incIndex();
  crt = tmpChild;
  crt.setMark(true);
} else { crt.incIndex(); }
```

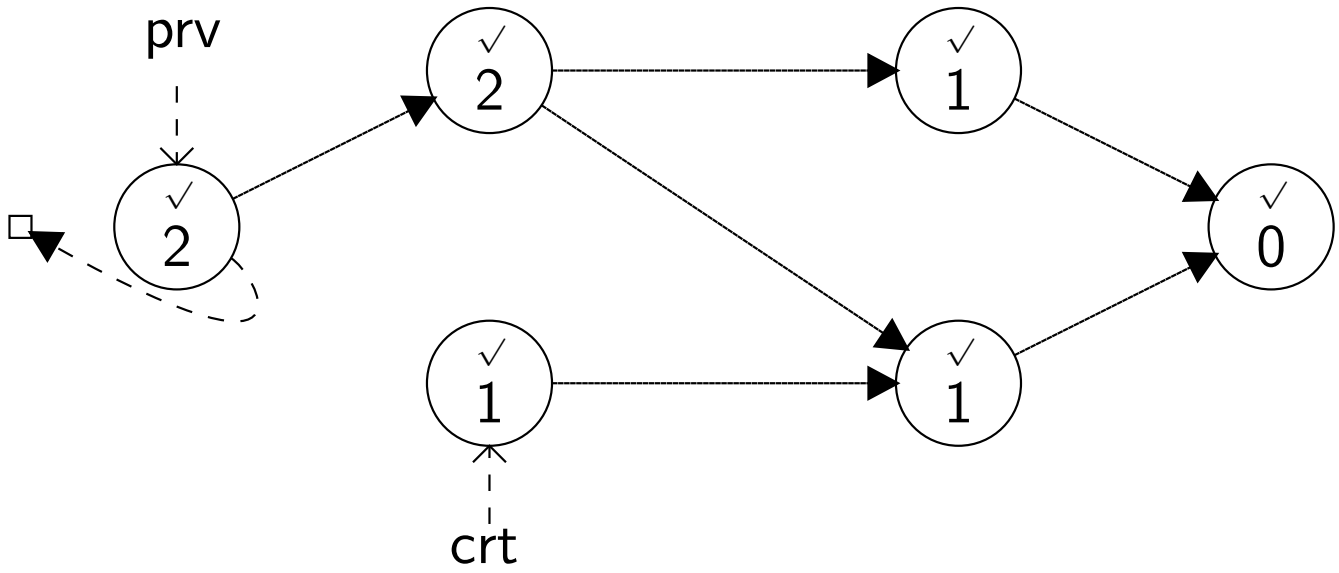
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
idx = crt.getIndex();
tmpChild = crt.getChild(idx);
if (!tmpChild.isMarked()) {
  crt.setChild(idx, prv);
  prv = crt;
  crt.incIndex();
  crt = tmpChild;
  crt.setMark(true);
} else { crt.incIndex(); }
```

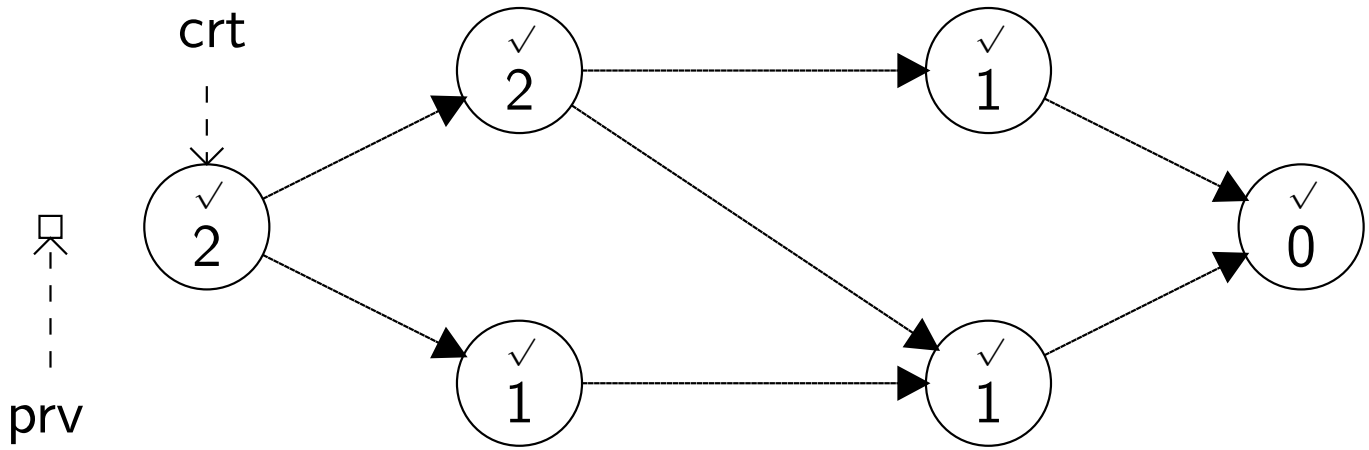
Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

```
ref2restore =
       prv.getIndex() - 1;
tmpChild =
       prv.getChild(ref2restore);
prv.setChild(ref2restore, crt);
crt = prv;
prv = tmpChild;
```

Schorr-Waite Algorithm




```
prv = null; crt = start;
crt.setMark(true);
while (start != crt ||
       start.hasNext()) {
  if (crt.hasNext()) {
     ...; // forward
  } else {
     ...; // backward
 }
}
```

Reasoning in presence of non-rigid symbols

Problem: Assume a method getA returning the value of attribute a

```
    {x.b:=c || o.a:=3}o.getA() = 3
==>
    {o.a:=3}o.getA() = 3
```

- Update simplification too coarse (not possible)

Reasoning in presence of non-rigid symbols

Problem: Assume a method getA returning the value of attribute a

```
    {x.b:=c || o.a:=3}o.getA() = 3
==>
    {o.a:=3}o.getA() = 3
```

- Update simplification too coarse (not possible)
- Need to know the definition of method getA

Non-rigid symbols with explicit dependencies

$$\langle symbolname \rangle \{ \langle locationDescriptor \rangle \} : T_1 \times \cdots \times T_n (\rightarrow T_{n+1})_{opt}$$

Example:

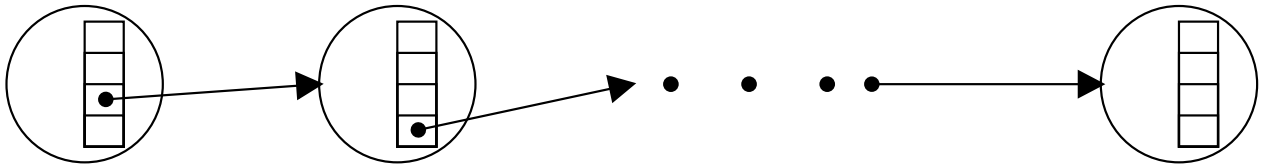
$$\texttt{nonNull}\{\backslash for\ T\ t; \backslash for\ int\ i; t.a[i]\} : T$$

Possible applications:

- modelling queries
- auxiliary functions like `permutation` predicate

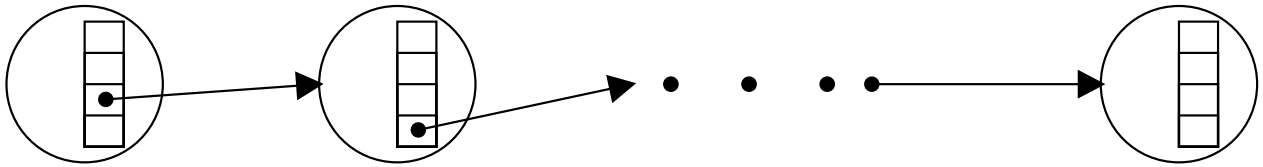
Modelling reachability

Let C denote a class with a field a of array type $C[]$



Modelling reachability

Let C denote a class with a field a of array type $C[]$



Definition (Reachable)

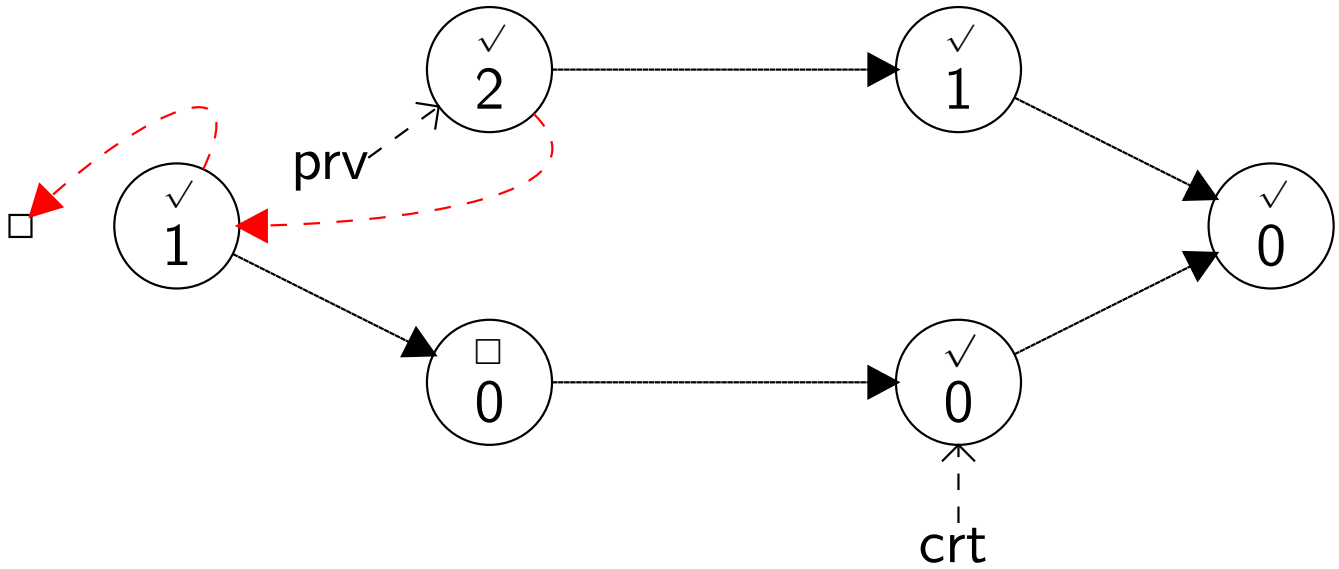
The predicate $reach[...](o, u, n)$ evaluates to *true* iff. u can be accessed from o via a chain $o.a[i_1]....a[i_n] \doteq u$.

KeY

```
\find (reach[\for (hov; iv) hov.children[iv];
              \for (hov2) hov2.children.length](t1, t2, n))
\varcond (\notFreeIn(k, t1, t2, n))
\replacewith (t1 = t2 & n = 0 |
 (t1 != null & n > 0 &
  \exists k;(k>=0 & k<t1.children.length &
              reach[...](t1.children[k], t2, n-1))))
```

KeY

Backtracking Path Encoding



Use again a non-rigid predicate: `onPath[...](prv,u,step)`

KeY

```
\find ( onPath[\for (hov; iv) hov.children[iv];
               \for (hov2) hov2.children.length;
               \for (hov3) hov3.nextChild](t1, t2, n) )
\replacewith( n >= 0 \& ( ( t1 = t2 \& n = 0) |
      ( t1 != null \& t1.nextChild > 0 \&
        t1.nextChild <= t1.children.length \&
        onPath[...](t1.children[t1.nextChild-1], t2, n-1))))
```

KeY

Proofobligation

```
inReachableState & ... &
\forall HeapObject ho; (!ho = null ->
   ho.visited = FALSE & ho.nextChild = 0)
->
\[{ sw.mark(startNode); }\]
  \forall HeapObject x; \forall int n;
    ( !x = null &
      reach[\for (HeapObject x; int i) x.children[i];
            \for HeapObject x; x.children.length
           ](startNode, x, n)
    -> ( x.visited = TRUE &
           \forall int i;(i>=0 & i<x.children.length ->
              x.children[i] = childrenPre(x,i)) ))
```

All nodes reachable from the starting node (not changed by mark)

1. are marked visited

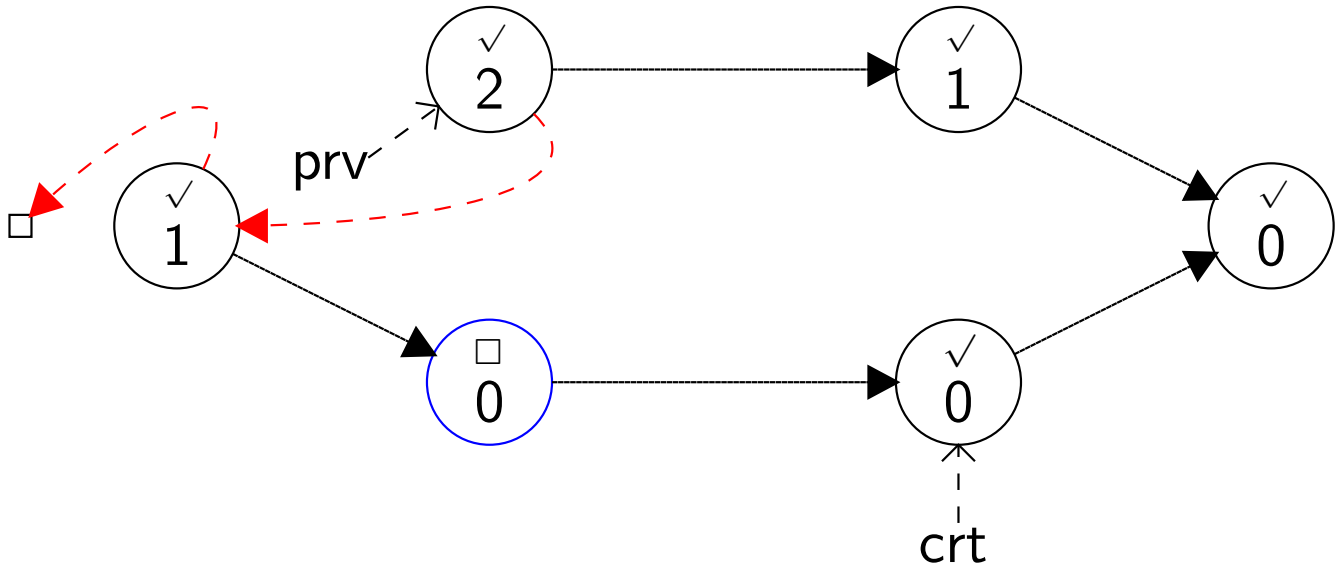
Proofobligation

```
inReachableState & ... &
\forall HeapObject ho; (!ho = null ->
   ho.visited = FALSE & ho.nextChild = 0)
->
\[{ sw.mark(startNode); }\]
  \forall HeapObject x; \forall int n;
    ( !x = null &
      reach[\for (HeapObject x; int i) x.children[i];
            \for HeapObject x; x.children.length
           ](startNode, x, n)
    -> ( x.visited = TRUE &
           \forall int i;(i>=0 & i<x.children.length ->
              x.children[i] = childrenPre(x,i)) ))
```

All nodes reachable from the starting node (not changed by mark)

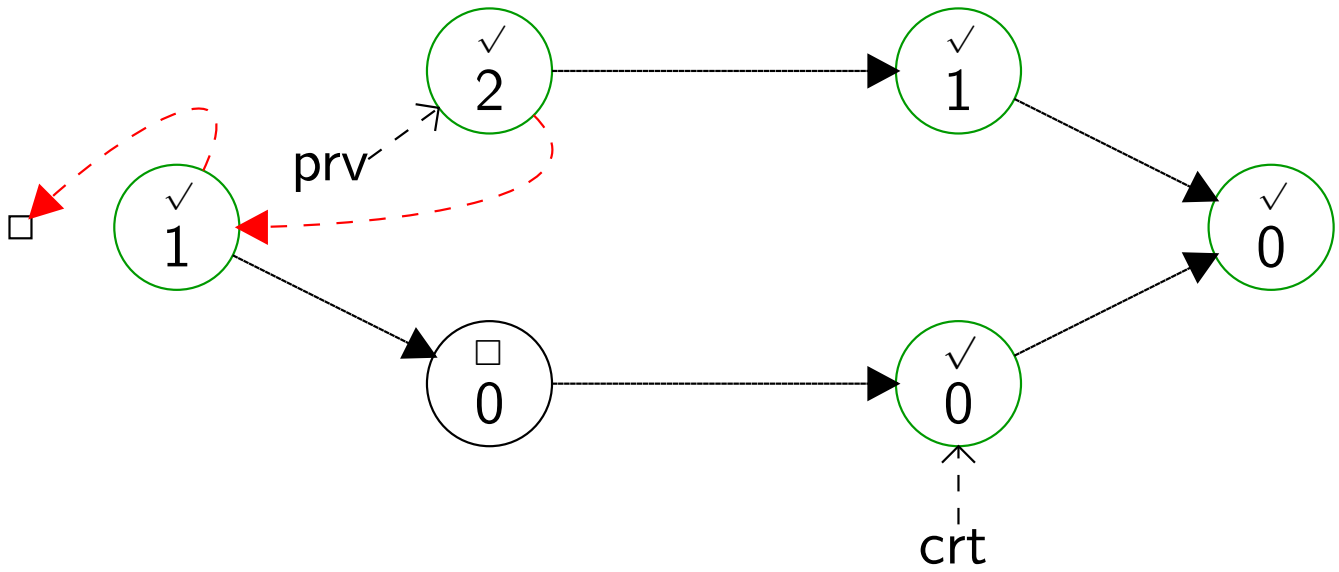
1. are marked visited
2. have same children as before (the graph structure unchanged)

Loop Inv. – Unvisited nodes have not been changed



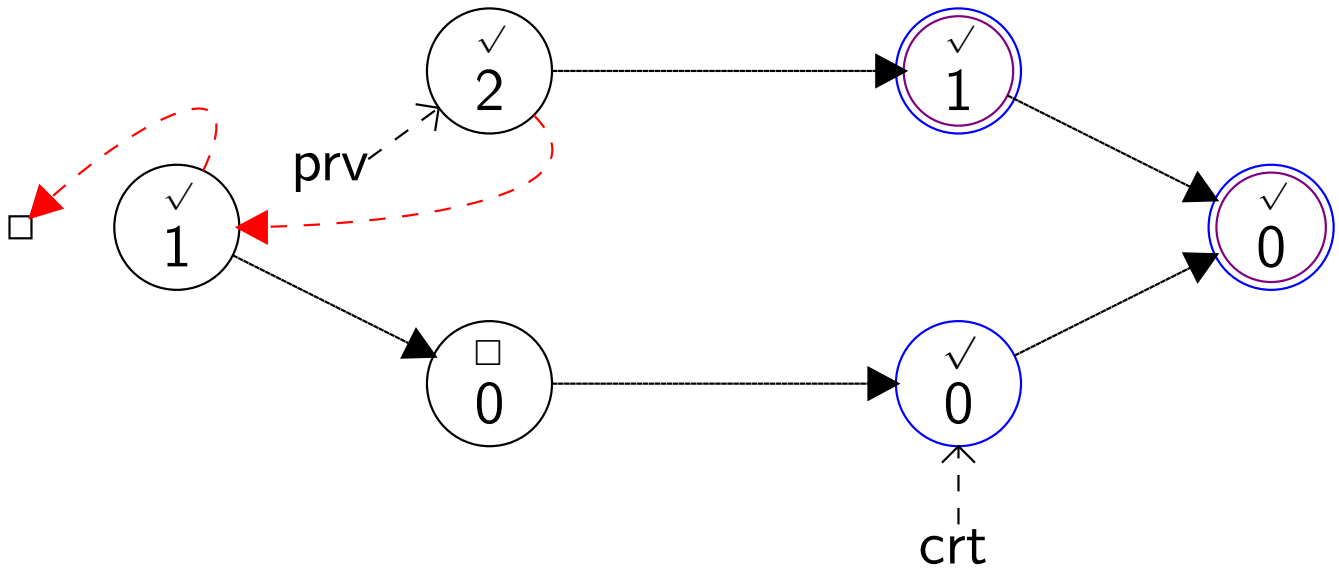

```
\forall HeapObject ho;(ho != null & ho.visited=FALSE ->
  (ho.nextChild = 0 &
    \forall int i;((i>=0 & i<ho.children.length) ->
                              ho.children[i]=childrenPre(ho,i))))
```

Loop Inv. – Visited nodes (*not* on the backtracking path)



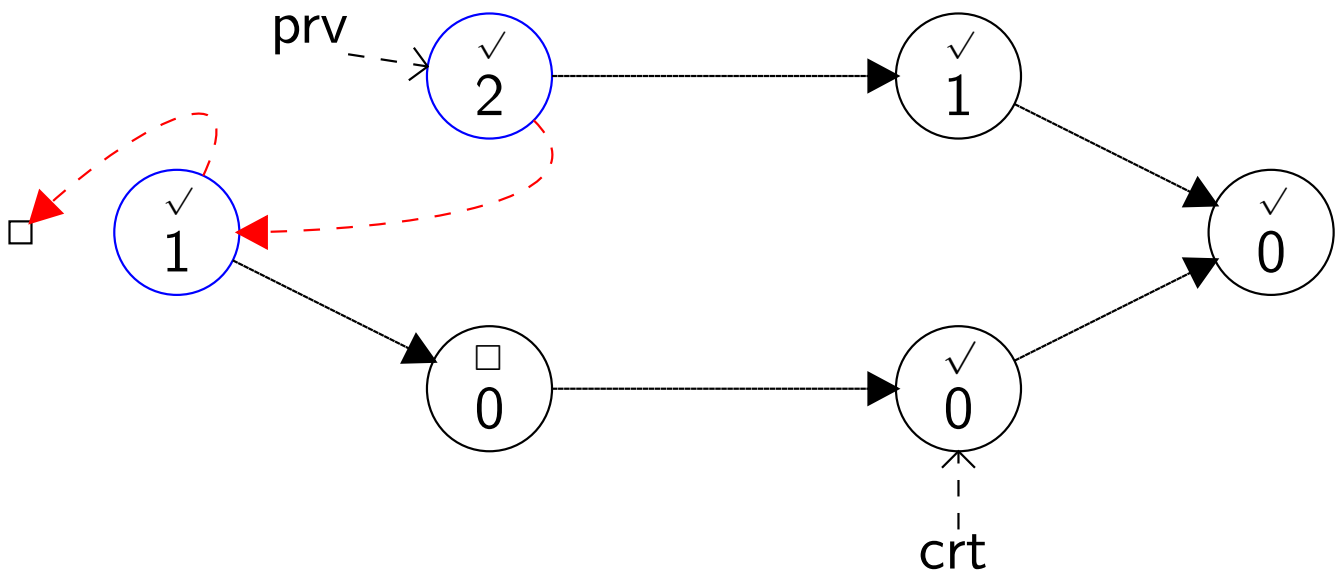

```
\forall HeapObject h;((h!=null & h.visited=TRUE) ->
  (\forall int i;((i >= 0 & i < h.nextChild - 1) ->
      (h.children[i].visited = TRUE &
       h.children[i]=childrenPre(h,i)))
  & ((!(\exists int dist; onPath[...](sw.previous, h, dist)))
    -> ( (h != sw.current -> h.nextChild = h.children.length) &
         (h.nextChild > 0 ->
             (h.children[h.nextChild-1].visited = TRUE   &
              h.children[h.nextChild-1]=childrenPre(h,h.nextChild-1)))
   ))))
```

Loop Inv. – Visited nodes (*not* on the backtracking path)



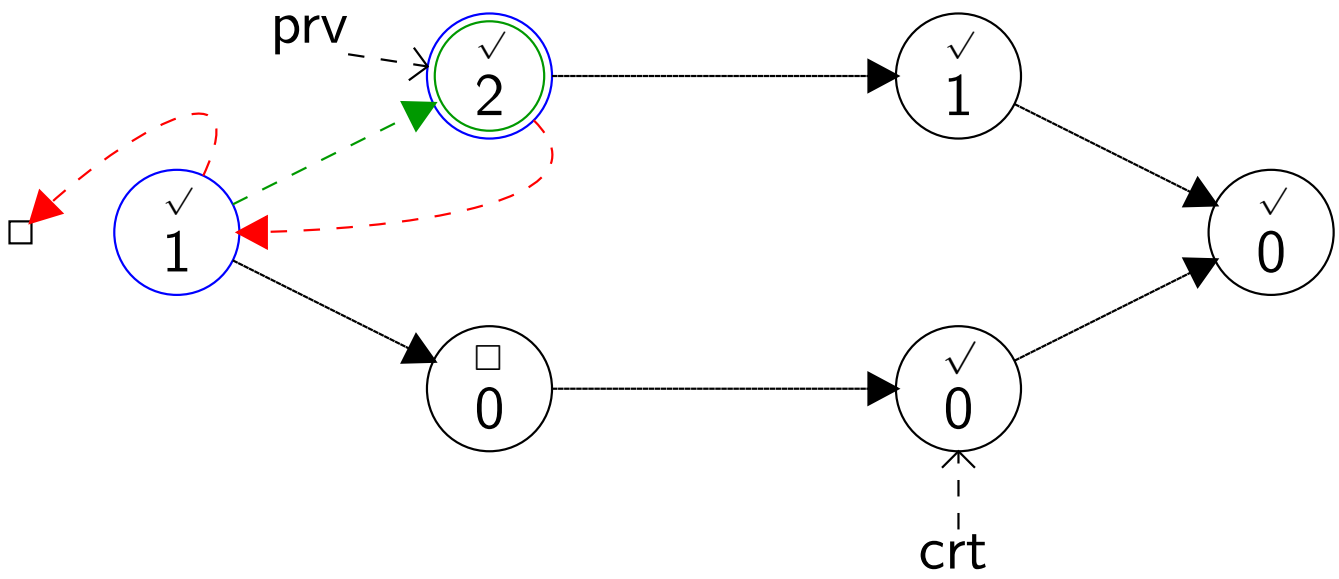

```
\forall HeapObject h;((h!=null & h.visited=TRUE) ->
  (\forall int i;((i >= 0 & i < h.nextChild - 1) ->
      (h.children[i].visited = TRUE &
       h.children[i]=childrenPre(h,i)))
  & ((!(\exists int dist; onPath[...](sw.previous, h, dist)))
    -> ( (h != sw.current -> h.nextChild = h.children.length) &
         (h.nextChild > 0 ->
             (h.children[h.nextChild-1].visited = TRUE   &
              h.children[h.nextChild-1]=childrenPre(h,h.nextChild-1)))
   ))))
```

Loop Inv. – Visited nodes *on* the backtracking path




```
\forall HeapObject ho;( (ho != null &
  \exists int i; onPath[...](sw.previous, ho, i)) ->
     (( childrenPre(ho.children[ho.nextChild-1],
         ho.children[ho.nextChild-1].nextChild - 1) = ho
      | (ho.children[ho.nextChild-1] = null &
          ho = startNode))
     & ho.visited = TRUE & ho.nextChild >= 1))
```

Loop Inv. – Visited nodes *on* the backtracking path




```
\forall HeapObject ho;( (ho != null &
  \exists int i; onPath[...](sw.previous, ho, i)) ->
     (( childrenPre(ho.children[ho.nextChild-1],
         ho.children[ho.nextChild-1].nextChild - 1) = ho
      | (ho.children[ho.nextChild-1] = null &
          ho = startNode))
     & ho.visited = TRUE & ho.nextChild >= 1))
```

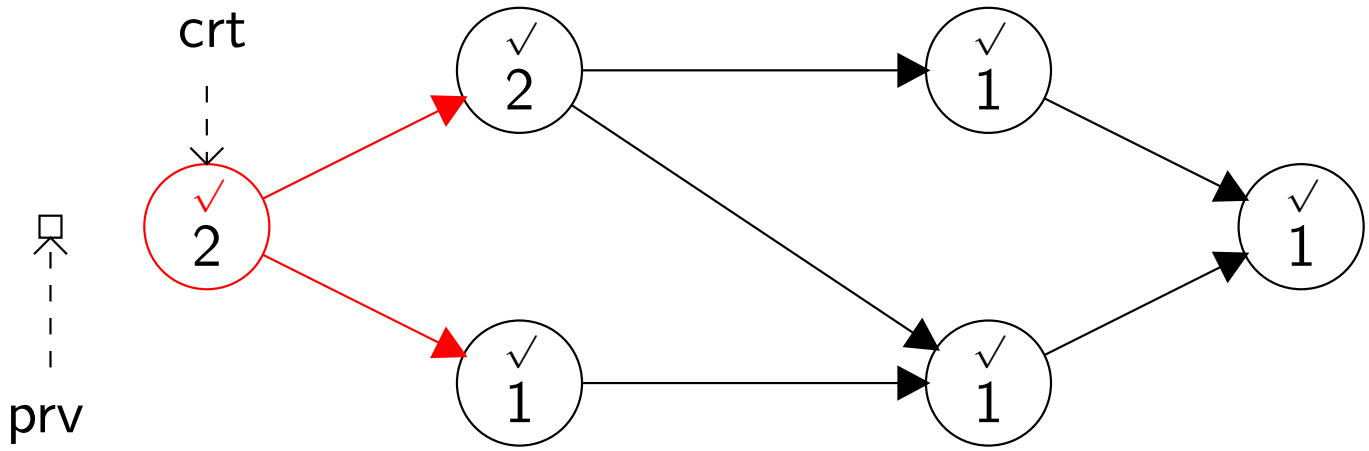
Proofobligation

```
inReachableState & ... &
\forall HeapObject ho; (!ho = null ->
   ho.visited = FALSE & ho.nextChild = 0)
->
\[{ sw.mark(startNode); }\]
  \forall HeapObject x; \forall int n;
    ( !x = null &
      reach[\for (HeapObject x; int i) x.children[i];
            \for HeapObject x; x.children.length
           ](startNode, x, n)
    -> ( x.visited = TRUE &
           \forall int i;(i>=0 & i<x.children.length ->
              x.children[i] = childrenPre(x,i)) ))
```

All nodes reachable from the starting node (not changed by mark)

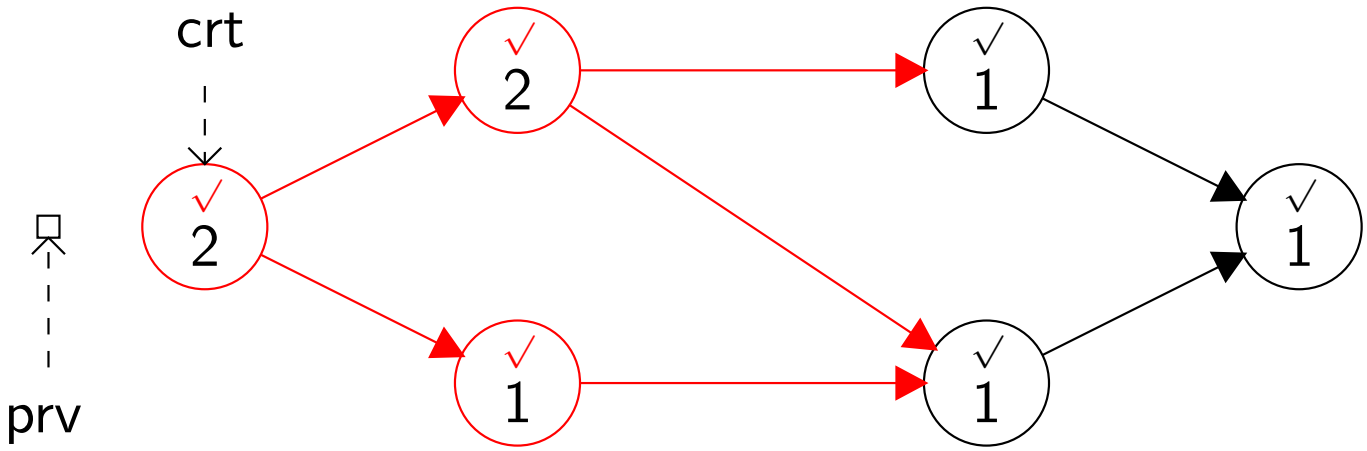
1. are marked visited
2. have same children as before (the graph structure unchanged)

Verification–Use Case



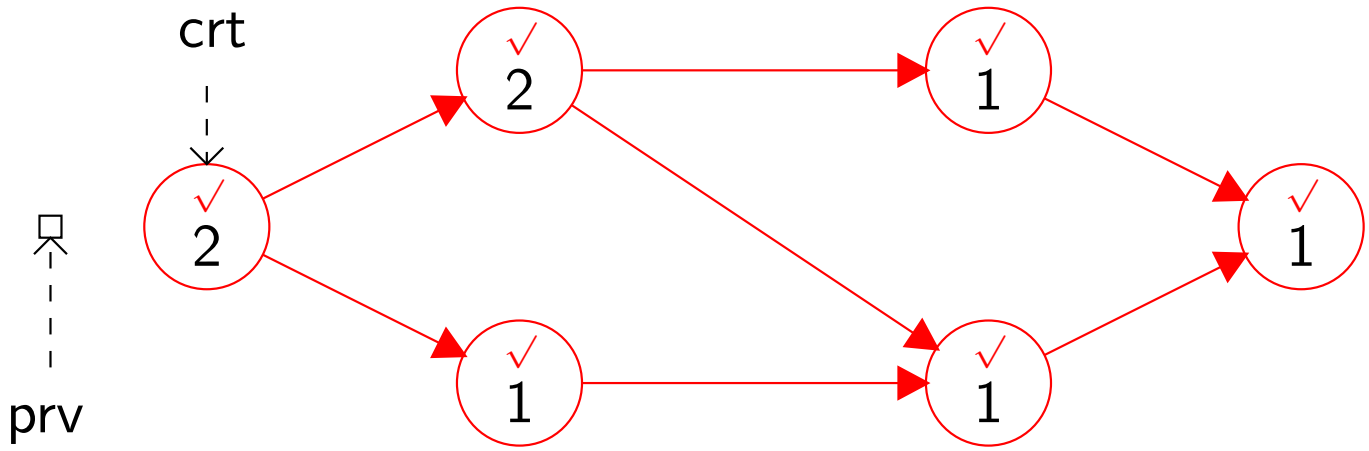
1. **Base Case:** use loop invariant conjunction est. the property for the start node (not shown)

Verification–Use Case



1. **Base Case:** use loop invariant conjunction est. the property for the start node (not shown)
2. **Step Case:** use loop invariant for visited nodes not on the backtracking path

Verification–Use Case

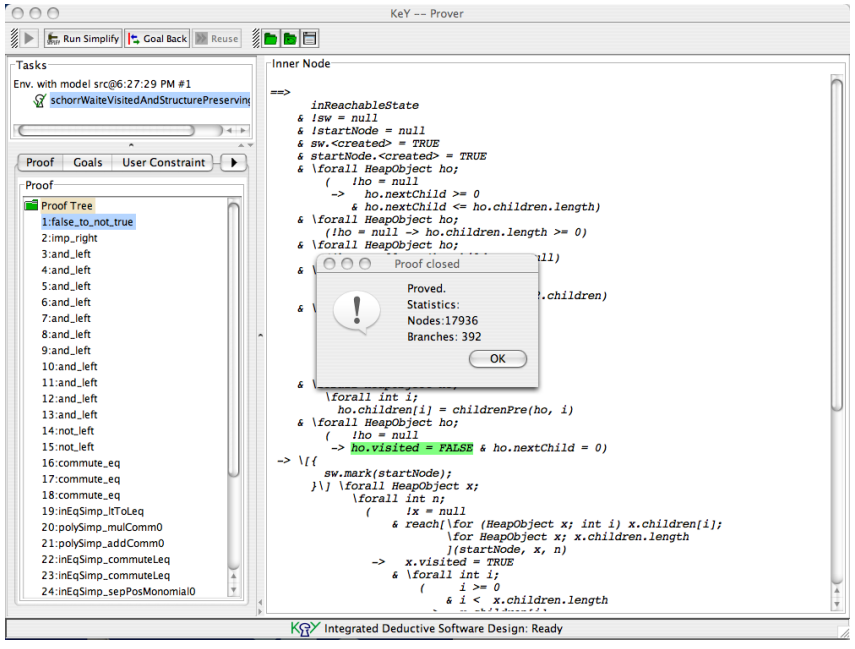


1. **Base Case:** use loop invariant conjunction est. the property for the start node (not shown)
2. **Step Case:** use loop invariant for visited nodes not on the backtracking path
3. **Use Case:** all nodes are visited and graph structure is unchanged

Verification–Statistics

Some statistics

proof steps	17936
interactive steps	1017
branches	392



Interactive steps have been required for

- quantifier instantiation
- *reachable* and *onPath* rule applications
- induction closing the use case

Conclusions

Verified Schorr-Waite algorithm implementation working on general graphs.

But too many user interactions required:

- possible to remove redundancies in loop invariant
- use KeY version with automatic quantifier treatment