Strong connectivity in directed graphs

- Nodes u and v are **mutually reachable** if there is a path from u to v and a path from v to u
- A directed graph is **strongly connected** if every pair of nodes are mutually reachable

Lemma

Let s be any node in G. G is strongly connected $\iff$ every node is reachable from s and s is reachable from any node.

Proof.

$\Rightarrow$ follows directly from the definition of strongly connected G. $\Leftarrow$ follows by constructing two paths:

- a path from u to v as $p = (u, ..., s, ..., v)$, and
- a path from v to u as $q = (v, ..., s, ..., u)$.

$\square$

Determining strong connectivity

- Select any node $s \in V$
- Use BFS on G from s and check if all of V is reached
- Construct G^r from G by reversing all edges
- Use BFS on G^r from s and check if all of V is reached
- If s can reach a node u in G^r, then u can reach s in G.
- If all of V is reached in both searches, G is strongly connected
- $O(n + m)$
- We will next see Tarjan's algorithm which instead uses depth first search and also lists all the strongly connected components

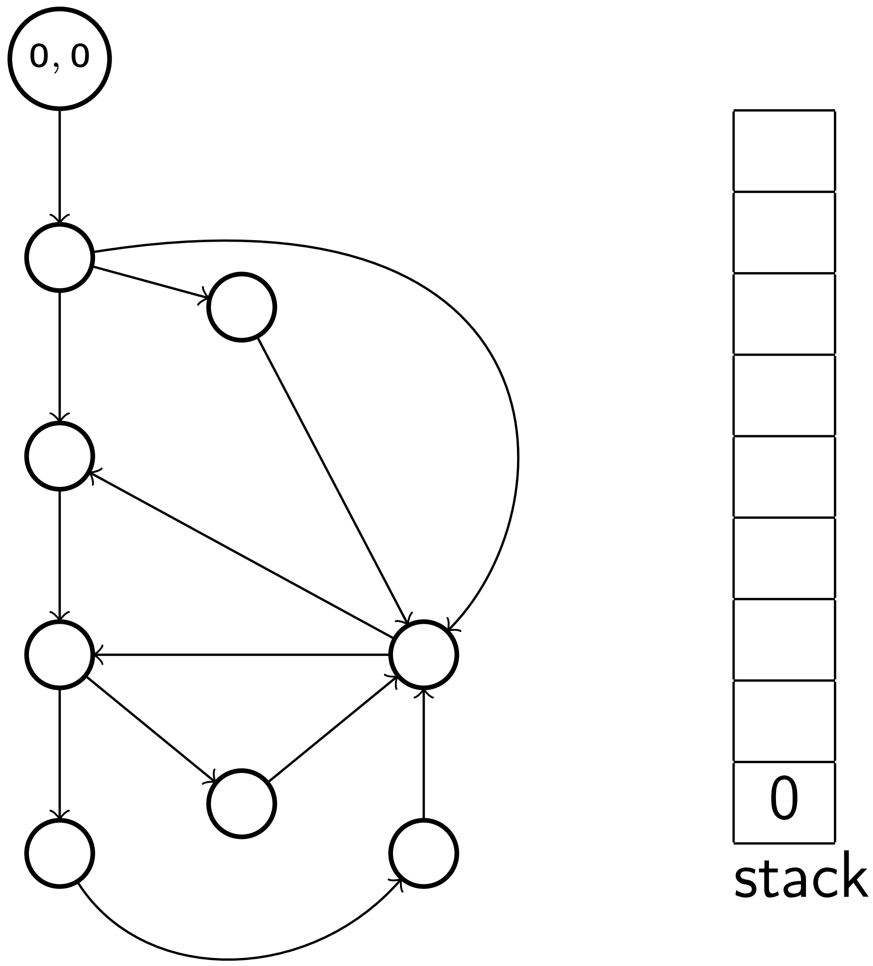
Tarjan's Algorithm: Initial Processing of 0

```
int     dfnum

procedure strong_connect (v)
    dfn (v) ← dfnum
    lowlink (v) ← dfnum
    visited (v) ← true
    push (v)
    dfnum ← dfnum +1

    for each w ∈ succ(v) do
        if (not visited (w)) {
            strong_connect (w)
            lowlink (v) ← min (lowlink (v), lowlink (w))
        } else if (dfn (w) < dfn (v)  and w is on stack)
            lowlink (v) ← min (lowlink (v), dfn (w))

    if (lowlink (v) = dfn (v))
        scc ← ∅
        do
            w ← pop ()
            add w to scc
        while (w ≠ v)
        process_scc (scc)
end
```



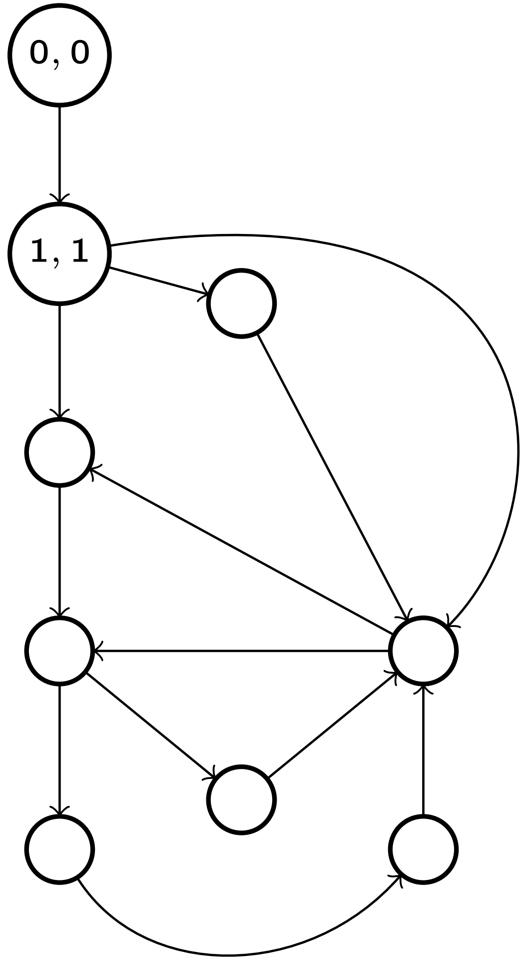
Tarjan's Algorithm: Initial Processing of 1

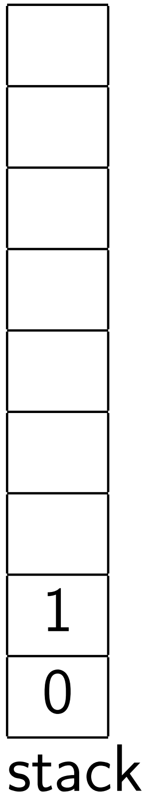
int *dfnum*

procedure *strong_connect* (v)
    $dfn(v) \leftarrow dfnum$
    $lowlink(v) \leftarrow dfnum$
    $visited(v) \leftarrow$ true
    $push(v)$
    $dfnum \leftarrow dfnum + 1$

    for each $w \in succ(v)$ **do**
        if (**not** $visited(w)$) {
            $strong\_connect(w)$
            $lowlink(v) \leftarrow min(lowlink(v), lowlink(w))$
        } **else if** ($dfn(w) < dfn(v)$ **and** w is on stack)
            $lowlink(v) \leftarrow min(lowlink(v), dfn(w))$

    if ($lowlink(v) = dfn(v)$)
        $scc \leftarrow \emptyset$
        do
            $w \leftarrow pop()$
            add w to scc
        while $(w \neq v)$
        $process\_scc(scc)$
end





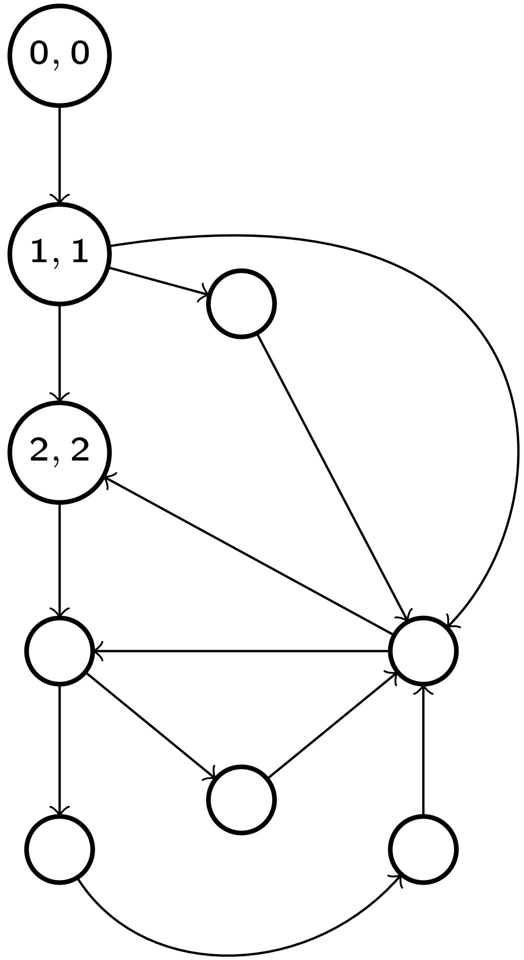
Tarjan's Algorithm: Initial Processing of 2

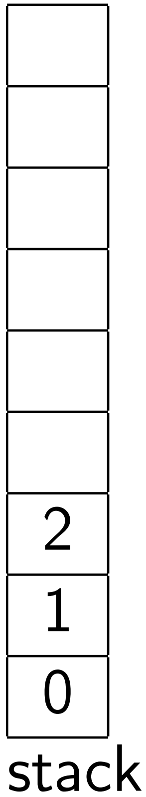
int *dfnum*

procedure *strong_connect* (v)
    dfn (v) $\leftarrow$ *dfnum*
    lowlink (v) $\leftarrow$ *dfnum*
    visited (v) $\leftarrow$ true
    push (v)
    dfnum $\leftarrow$ *dfnum* $+1$

    for each $w \in succ(v)$ **do**
        if (**not** *visited* (w)) {
            strong_connect (w)
            lowlink (v) $\leftarrow$ *min* (*lowlink* (v), *lowlink* (w))
        } **else if** (*dfn* (w) $<$ *dfn* (v) **and** w is on stack)
            lowlink (v) $\leftarrow$ *min* (*lowlink* (v), *dfn* (w))

    if (*lowlink* (v) $=$ *dfn* (v))
        scc $\leftarrow \emptyset$
        do
            $w \leftarrow$ *pop* ()
            add w to *scc*
        while ($w \neq v$)
        process_scc (*scc*)
end





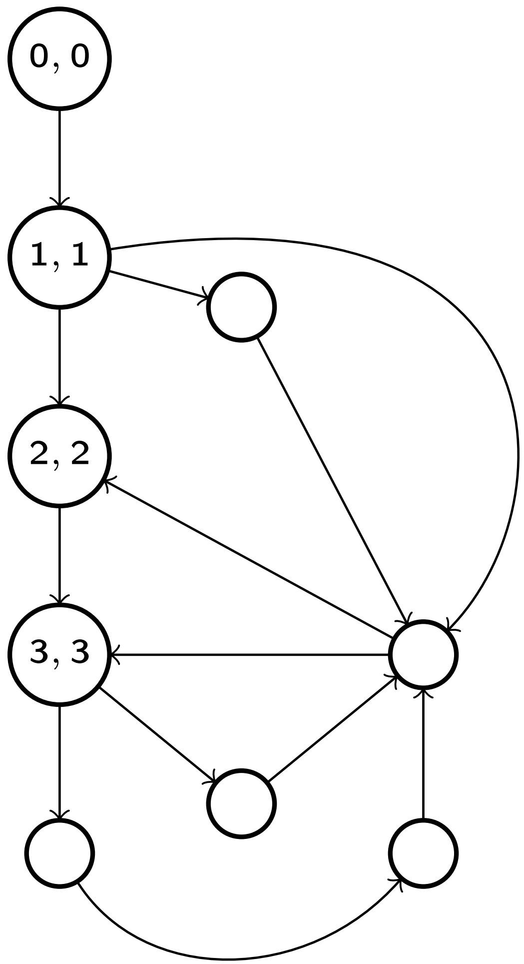
Tarjan's Algorithm: Initial Processing of 3

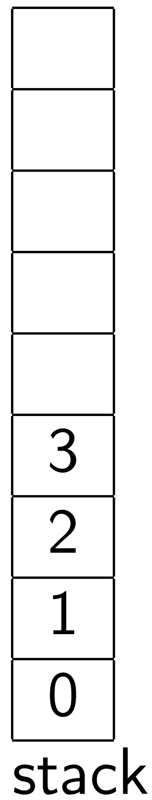
```
int     dfnum

procedure strong_connect(v)
        dfn(v) ← dfnum
        lowlink(v) ← dfnum
        visited(v) ← true
        push(v)
        dfnum ← dfnum +1

        for each w ∈ succ(v) do
                if (not visited(w)) {
                        strong_connect(w)
                        lowlink(v) ← min(lowlink(v), lowlink(w))
                } else if (dfn(w) < dfn(v)  and w is on stack)
                        lowlink(v) ← min(lowlink(v), dfn(w))

        if (lowlink(v) = dfn(v))
                scc ← ∅
                do
                        w ← pop()
                        add w to scc
                while (w ≠ v)
                process_scc(scc)
end
```





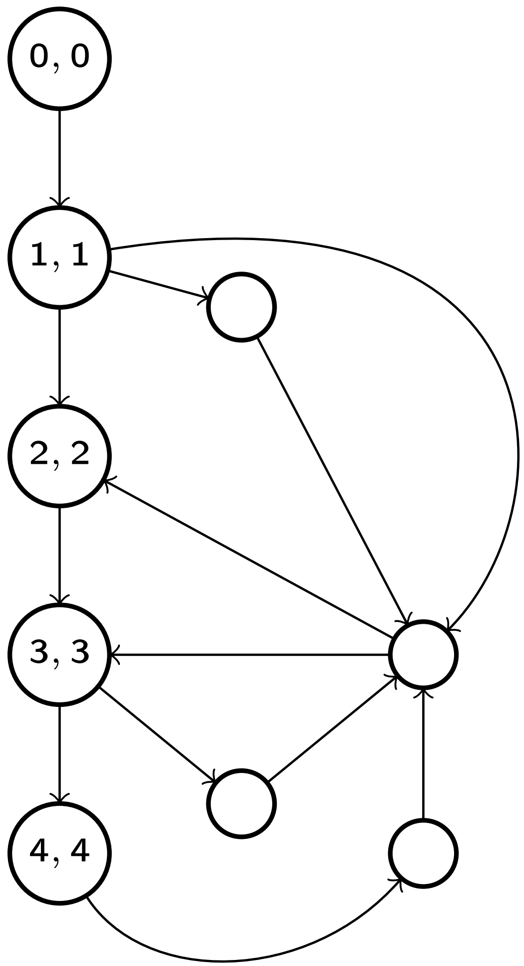
Tarjan's Algorithm: Initial Processing of 4

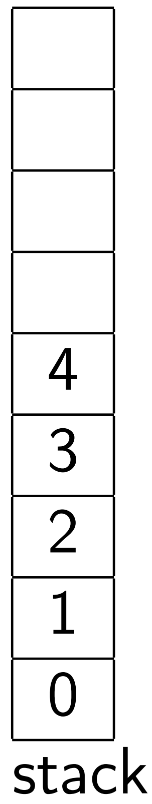
```
int     dfnum

procedure strong_connect (v)
    dfn (v) ← dfnum
    lowlink (v) ← dfnum
    visited (v) ← true
    push (v)
    dfnum ← dfnum +1

    for each w ∈ succ(v) do
        if (not visited (w)) {
            strong_connect (w)
            lowlink (v) ← min (lowlink (v), lowlink (w))
        } else if (dfn (w) < dfn (v)  and w is on stack)
            lowlink (v) ← min (lowlink (v), dfn (w))

    if (lowlink (v) = dfn (v))
        scc ← ∅
        do
            w ← pop ()
            add w to scc
        while (w ≠ v)
        process_scc (scc)
end
```





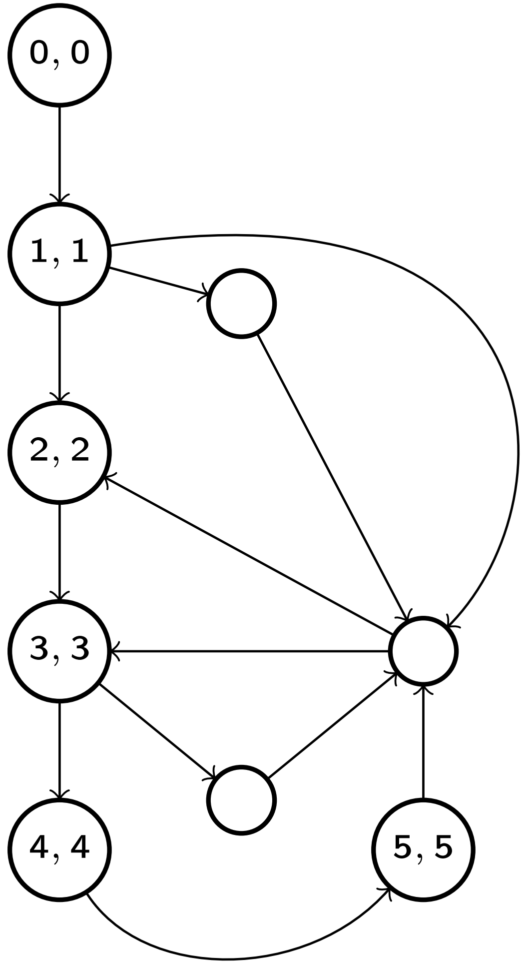
Tarjan's Algorithm: Initial Processing of 5

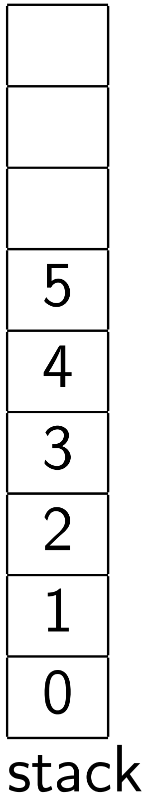
int *dfnum*

procedure *strong_connect* (v)
    dfn (v) $\leftarrow$ *dfnum*
    lowlink (v) $\leftarrow$ *dfnum*
    visited (v) $\leftarrow$ true
    push (v)
    dfnum $\leftarrow$ *dfnum* $+1$

    for each $w \in$ *succ*(v) **do**
        if (**not** *visited* (w)) {
            strong_connect (w)
            lowlink (v) $\leftarrow$ *min* (*lowlink* (v), *lowlink* (w))
        } **else if** (*dfn* (w) $<$ *dfn* (v) **and** w is on stack)
            lowlink (v) $\leftarrow$ *min* (*lowlink* (v), *dfn* (w))

    if (*lowlink* (v) $=$ *dfn* (v))
        scc $\leftarrow \emptyset$
        do
            $w \leftarrow$ *pop* ()
            add w to *scc*
        while ($w \neq v$)
        process_scc (*scc*)
end





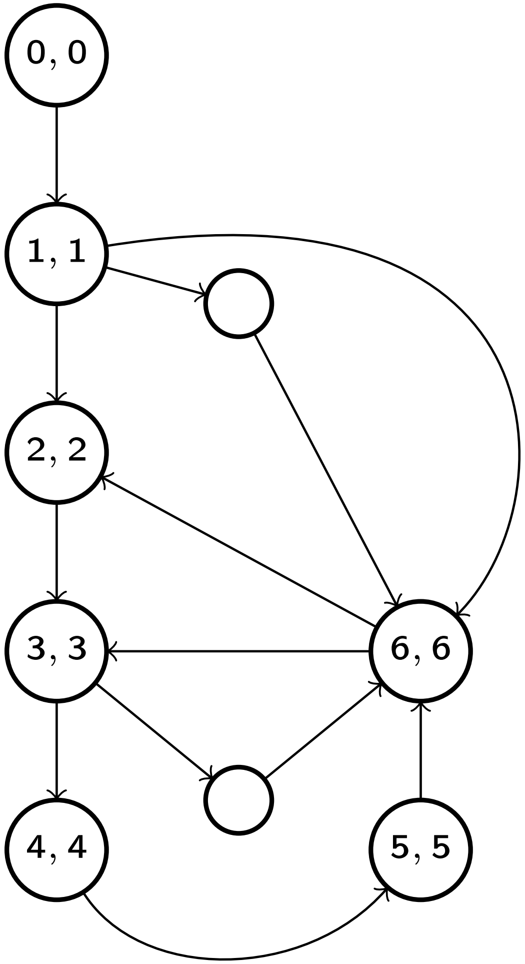
Tarjan's Algorithm: Initial Processing of 6

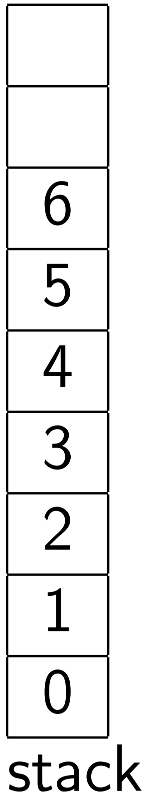
```
int      dfnum

procedure strong_connect (v)
    dfn (v) ← dfnum
    lowlink (v) ← dfnum
    visited (v) ← true
    push (v)
    dfnum ← dfnum +1

    for each w ∈ succ(v) do
        if (not visited (w)) {
            strong_connect (w)
            lowlink (v) ← min (lowlink (v), lowlink (w))
        } else if (dfn (w) < dfn (v)  and w is on stack)
            lowlink (v) ← min (lowlink (v), dfn (w))

    if (lowlink (v) = dfn (v))
        scc ← ∅
        do
            w ← pop ()
            add w to scc
        while (w ≠ v)
        process_scc (scc)
end
```





Tarjan's Algorithm: More Processing of 6

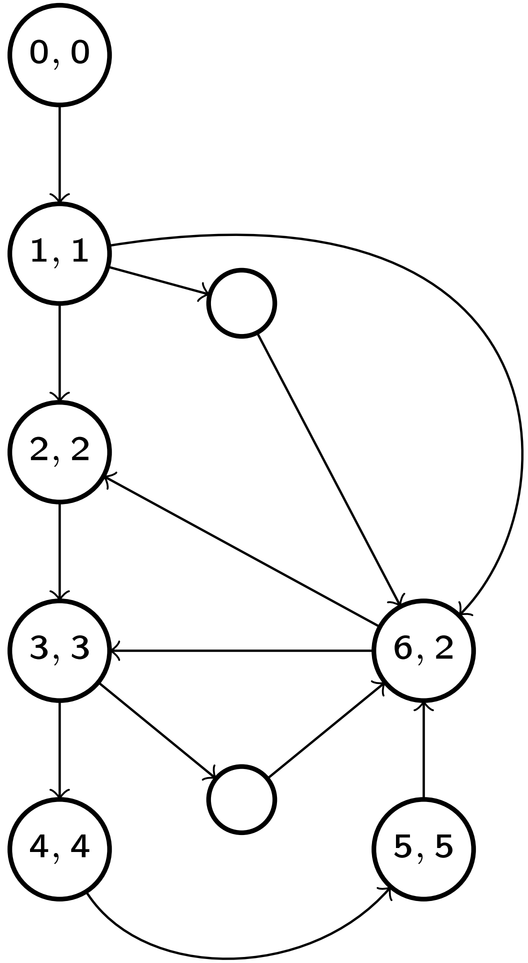
- $(6, 2) \Rightarrow 6$ in same scc as 2.

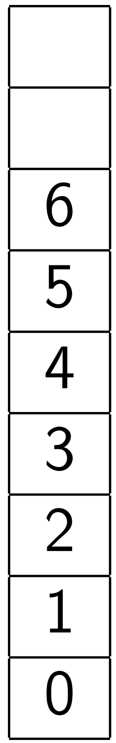
```
int     dfnum

procedure strong_connect(v)
        dfn(v) ← dfnum
        lowlink(v) ← dfnum
        visited(v) ← true
        push(v)
        dfnum ← dfnum +1

        for each w ∈ succ(v) do
                if (not visited(w)) {
                        strong_connect(w)
                        lowlink(v) ← min(lowlink(v), lowlink(w))
                } else if (dfn(w) < dfn(v)  and w is on stack)
                        lowlink(v) ← min(lowlink(v), dfn(w))

        if (lowlink(v) = dfn(v))
                scc ← ∅
                do
                        w ← pop()
                        add w to scc
                while (w ≠ v)
                process_scc(scc)
end
```





stack

Tarjan's Algorithm: More Processing of 6

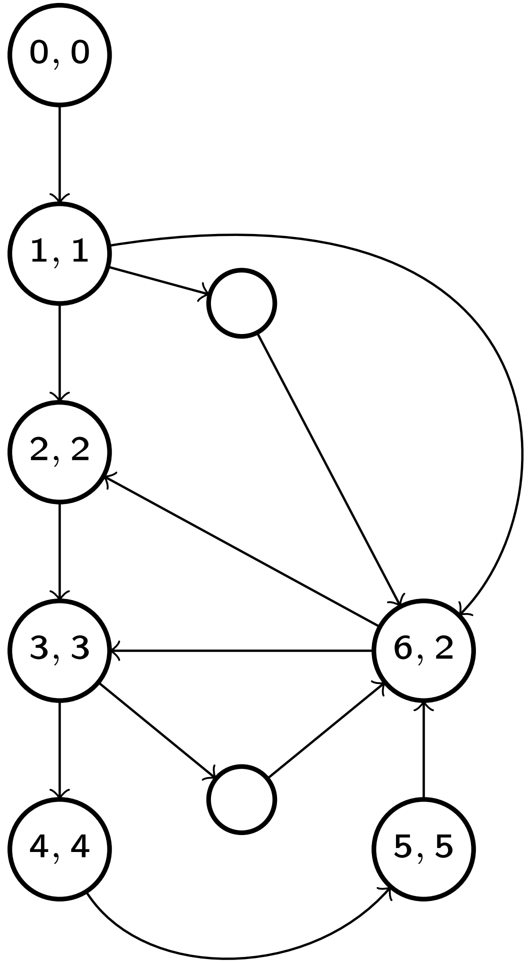
- (6, 3). no action.

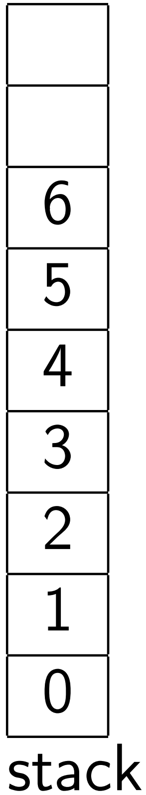
```
int      dfnum

procedure strong_connect (v)
    dfn (v) ← dfnum
    lowlink (v) ← dfnum
    visited (v) ← true
    push (v)
    dfnum ← dfnum +1

    for each w ∈ succ(v) do
        if (not visited (w)) {
            strong_connect (w)
            lowlink (v) ← min (lowlink (v), lowlink (w))
        } else if (dfn (w) < dfn (v)  and w is on stack)
            lowlink (v) ← min (lowlink (v), dfn (w))

    if (lowlink (v) = dfn (v))
        scc ← ∅
        do
            w ← pop ()
            add w to scc
        while (w ≠ v)
        process_scc (scc )
end
```





Tarjan's Algorithm: More Processing of 6

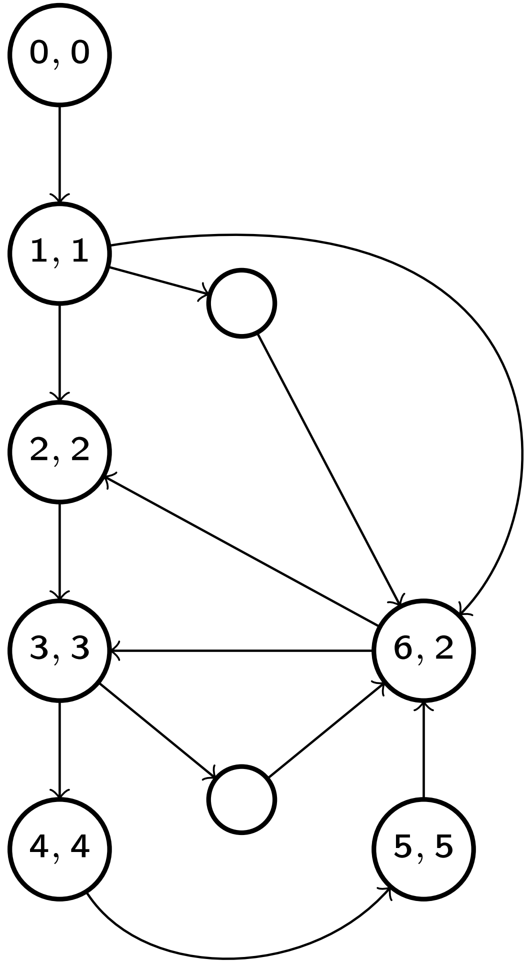
- 6 remains on the stack.

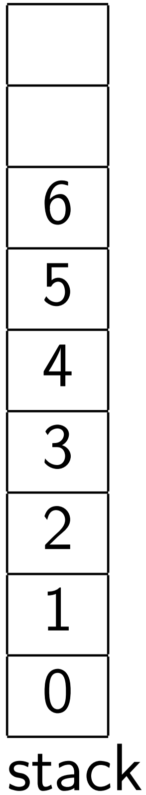
int *dfnum*

procedure *strong_connect* (v)
  $dfn(v) \leftarrow dfnum$
  $lowlink(v) \leftarrow dfnum$
  $visited(v) \leftarrow$ true
  $push(v)$
  $dfnum \leftarrow dfnum + 1$

  for each $w \in succ(v)$ **do**
    if (not $visited(w)$**) {**
      $strong\_connect(w)$
      $lowlink(v) \leftarrow min(lowlink(v), lowlink(w))$
    } else if ($dfn(w) < dfn(v)$ **and** w is on stack)
      $lowlink(v) \leftarrow min(lowlink(v), dfn(w))$

  if ($lowlink(v) = dfn(v)$)
    $scc \leftarrow \emptyset$
    do
      $w \leftarrow pop()$
      add w to scc
    while ($w \neq v$)
    $process\_scc(scc)$
end





Tarjan's Algorithm: More Processing of 5

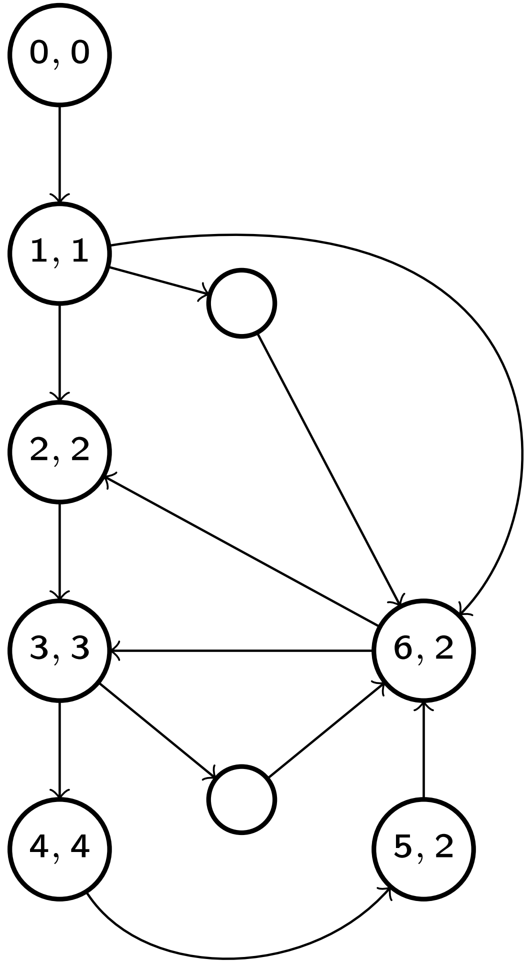
- New lowlink and remains.

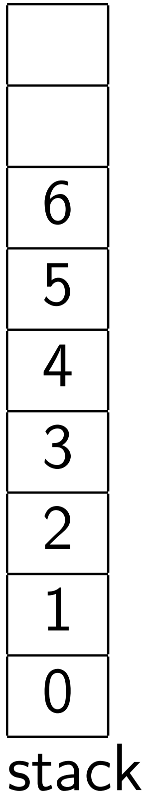
int *dfnum*

procedure *strong_connect* (v)
    dfn (v) $\leftarrow$ *dfnum*
    lowlink (v) $\leftarrow$ *dfnum*
    visited (v) $\leftarrow$ true
    push (v)
    dfnum $\leftarrow$ *dfnum* $+1$

    for each $w \in$ *succ*(v) **do**
        if (**not** *visited* (w)) {
            strong_connect (w)
            lowlink (v) $\leftarrow$ *min* (*lowlink* (v), *lowlink* (w))
        } **else if** (*dfn* (w) $<$ *dfn* (v) **and** w is on stack)
            lowlink (v) $\leftarrow$ *min* (*lowlink* (v), *dfn* (w))

    if (*lowlink* (v) $=$ *dfn* (v))
        scc $\leftarrow \emptyset$
        do
            $w \leftarrow$ *pop* ()
            add w to *scc*
        while ($w \neq v$)
        process_scc (*scc*)
end





Tarjan's Algorithm: More Processing of 4

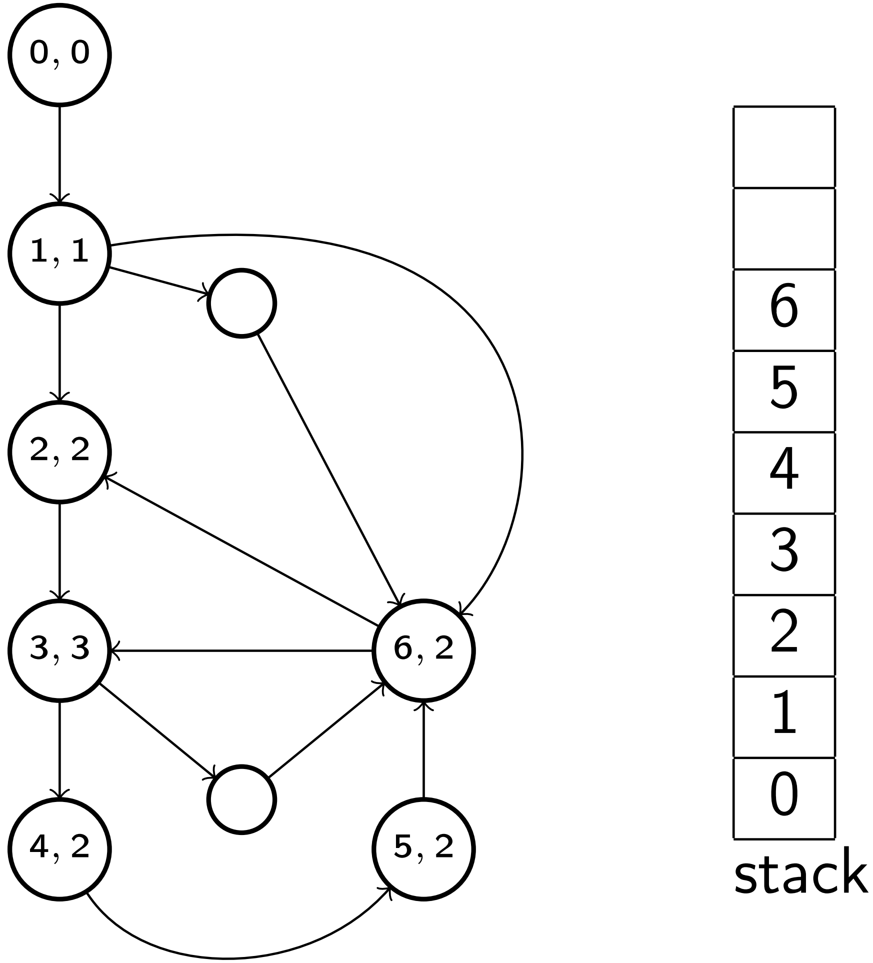
- New lowlink and remains.

int *dfnum*

procedure $strong\_connect\,(v)$
    $dfn\,(v) \leftarrow dfnum$
    $lowlink\,(v) \leftarrow dfnum$
    $visited\,(v) \leftarrow$ true
    $push\,(v)$
    $dfnum \leftarrow dfnum\ {+1}$

    for each $w \in succ(v)$ **do**
        if (not $visited\,(w)$**)** {
            $strong\_connect\,(w)$
            $lowlink\,(v) \leftarrow min\,(lowlink\,(v),\ lowlink\,(w))$
        } **else if** $(dfn\,(w) < dfn\,(v)$ **and** w is on stack)
            $lowlink\,(v) \leftarrow min\,(lowlink\,(v),\ dfn\,(w))$

    if $(lowlink\,(v) = dfn\,(v))$
        $scc \leftarrow \emptyset$
        do
            $w \leftarrow pop\,()$
            add w to scc
        while $(w \neq v)$
        $process\_scc\,(scc)$
end



Tarjan's Algorithm: More Processing of 3

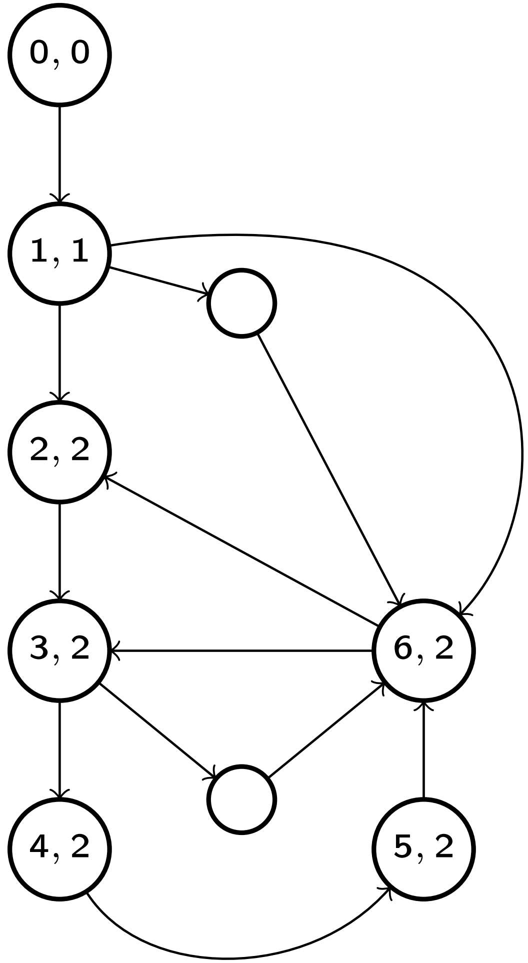
- New lowlink. Next 7.

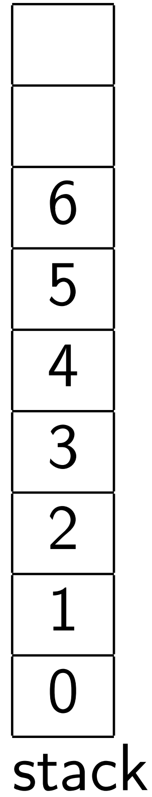
int *dfnum*

procedure *strong_connect* (v)
  $dfn(v) \leftarrow dfnum$
  $lowlink(v) \leftarrow dfnum$
  $visited(v) \leftarrow$ true
  $push(v)$
  $dfnum \leftarrow dfnum + 1$

  for each $w \in succ(v)$ **do**
    if (not $visited(w)$**) {**
      $strong\_connect(w)$
      $lowlink(v) \leftarrow min(lowlink(v), lowlink(w))$
    } else if $(dfn(w) < dfn(v)$ **and** w is on stack)
      $lowlink(v) \leftarrow min(lowlink(v), dfn(w))$

  if $(lowlink(v) = dfn(v))$
    $scc \leftarrow \emptyset$
    do
      $w \leftarrow pop()$
      add w to scc
    while $(w \neq v)$
    $process\_scc(scc)$
end





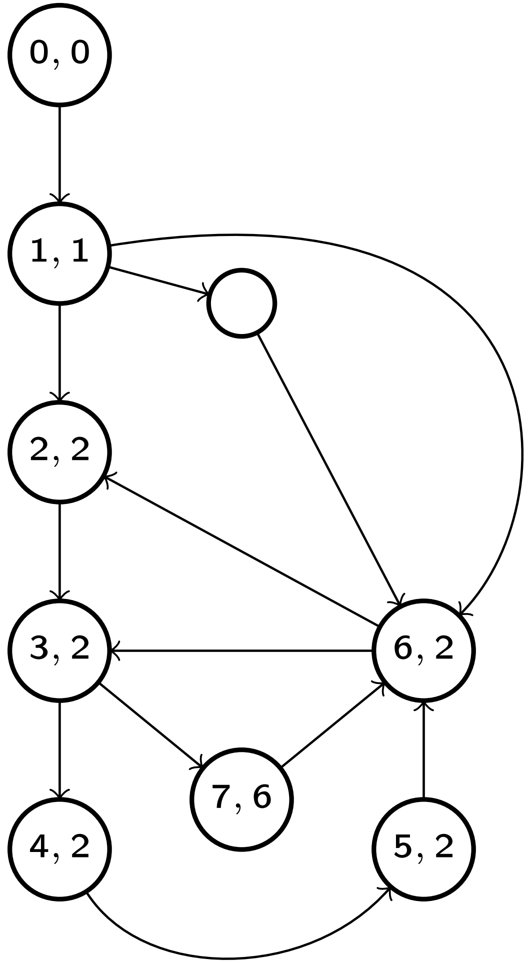
- Lowlink is set.

```
int     dfnum

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v)  and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```



7
6
5
4
3
2
1
0

stack

Tarjan's Algorithm: More Processing of 2

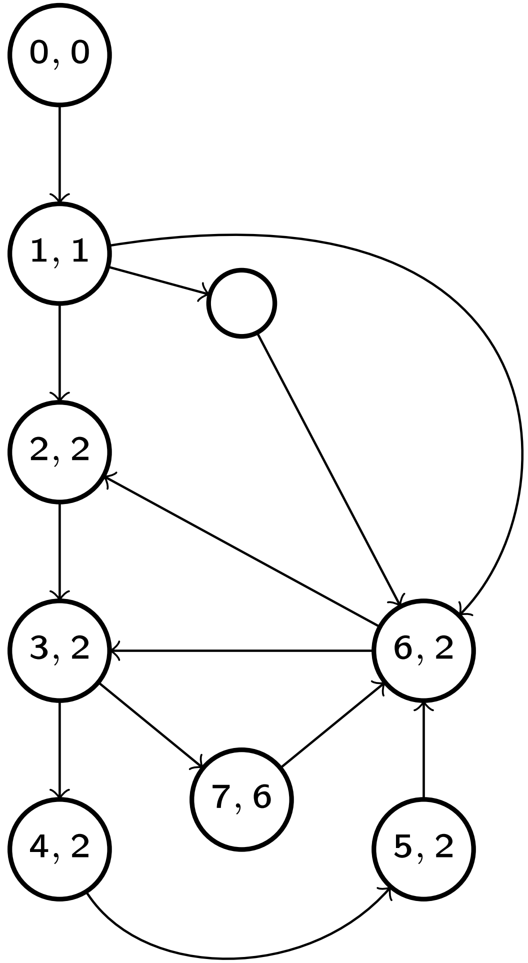
- Remove SCC from stack

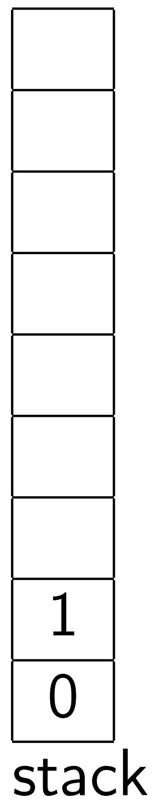
```
int     dfnum

procedure strong_connect (v)
    dfn (v) ← dfnum
    lowlink (v) ← dfnum
    visited (v) ← true
    push (v)
    dfnum ← dfnum +1

    for each w ∈ succ(v) do
        if (not visited (w)) {
            strong_connect (w)
            lowlink (v) ← min (lowlink (v), lowlink (w))
        } else if (dfn (w) < dfn (v)  and w is on stack)
            lowlink (v) ← min (lowlink (v), dfn (w))

    if (lowlink (v) = dfn (v))
        scc ← ∅
        do
            w ← pop ()
            add w to scc
        while (w ≠ v)
        process_scc (scc )
end
```





Tarjan's Algorithm: Initial Processing of 8

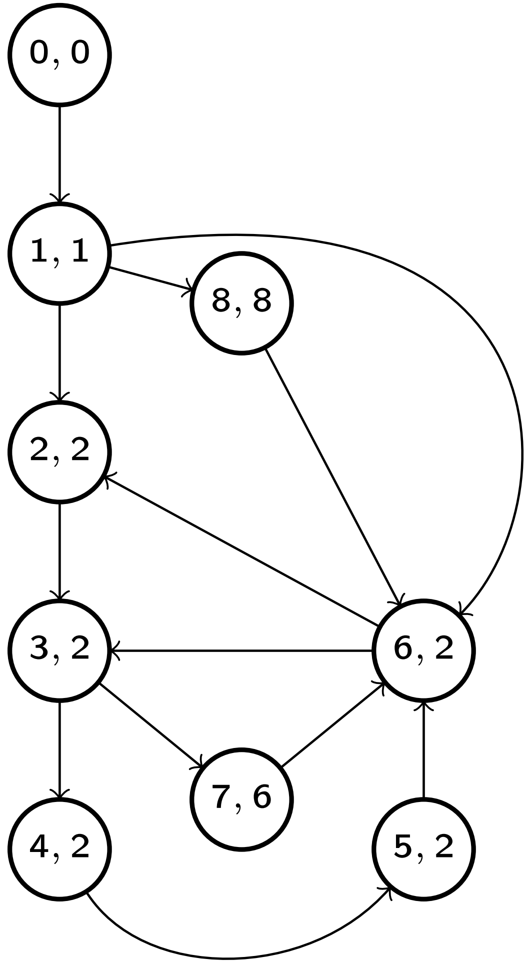
- No path from 2 to 8.

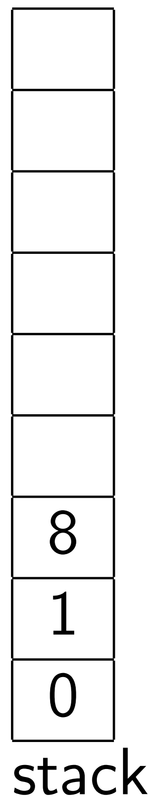
int *dfnum*

procedure *strong_connect* (v)
  $dfn(v) \leftarrow dfnum$
  $lowlink(v) \leftarrow dfnum$
  $visited(v) \leftarrow$ true
  $push(v)$
  $dfnum \leftarrow dfnum + 1$

  for each $w \in succ(v)$ **do**
    if (not $visited(w)$**) {**
      $strong\_connect(w)$
      $lowlink(v) \leftarrow min(lowlink(v), lowlink(w))$
    } else if $(dfn(w) < dfn(v)$ **and** w is on stack)
      $lowlink(v) \leftarrow min(lowlink(v), dfn(w))$

  if $(lowlink(v) = dfn(v))$
    $scc \leftarrow \emptyset$
    do
      $w \leftarrow pop()$
      add w to scc
    while $(w \neq v)$
    $process\_scc(scc)$
end





Tarjan's Algorithm: More Processing of 8

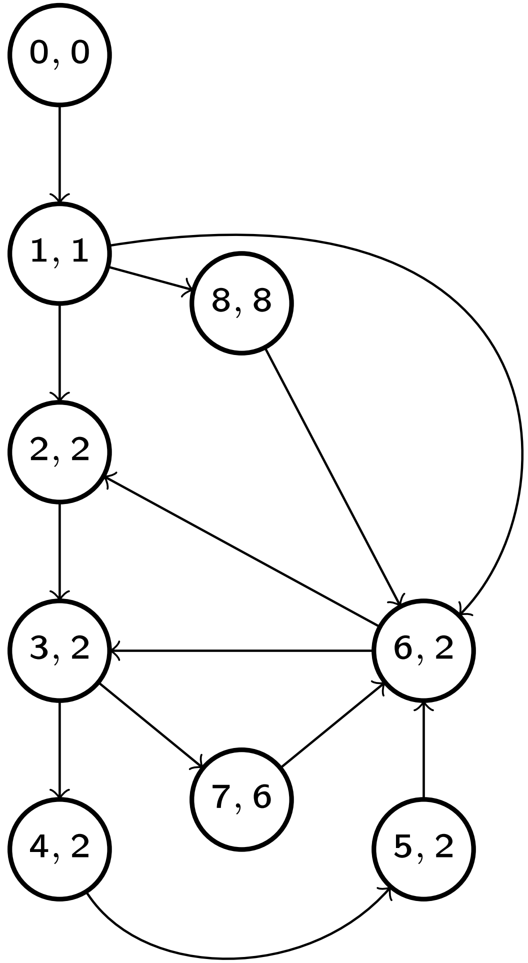
- 8 is its own SCC.

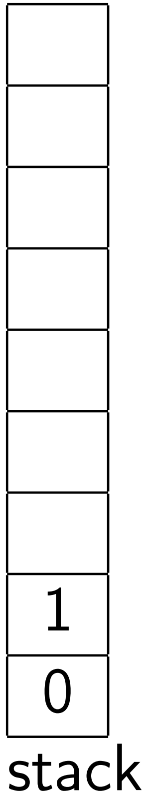
```
int    dfnum

procedure strong_connect (v)
    dfn (v) ← dfnum
    lowlink (v) ← dfnum
    visited (v) ← true
    push (v)
    dfnum ← dfnum +1

    for each w ∈ succ(v) do
        if (not visited (w)) {
            strong_connect (w)
            lowlink (v) ← min (lowlink (v), lowlink (w))
        } else if (dfn (w) < dfn (v)  and w is on stack)
            lowlink (v) ← min (lowlink (v), dfn (w))

    if (lowlink (v) = dfn (v))
        scc ← ∅
        do
            w ← pop ()
            add w to scc
        while (w ≠ v)
        process_scc (scc )
end
```





Tarjan's Algorithm: More Processing of 1

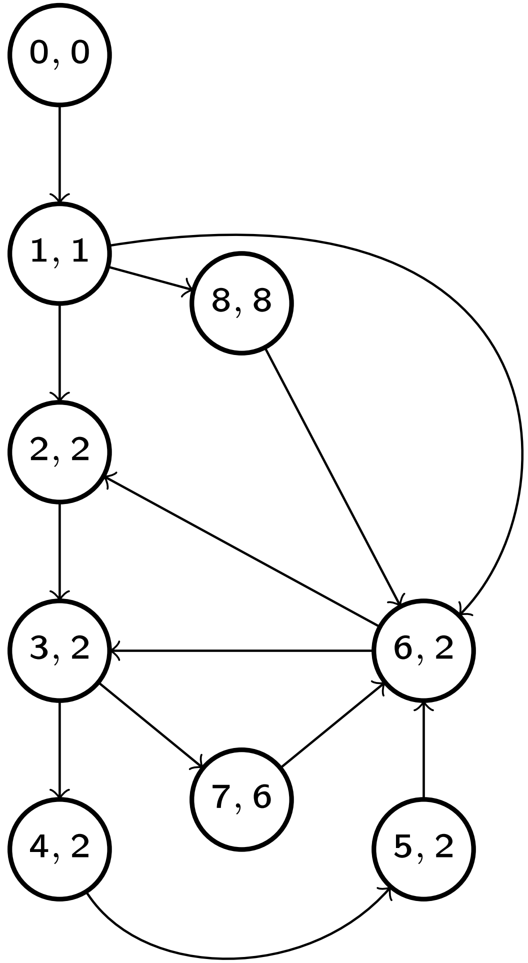
- 1 is its own SCC.

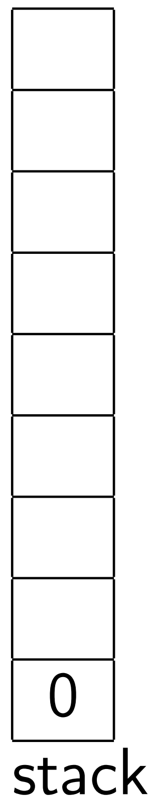
```
int     dfnum

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum + 1

    for each w ∈ succ(v) do
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v)  and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```





Tarjan's Algorithm: More Processing of 0

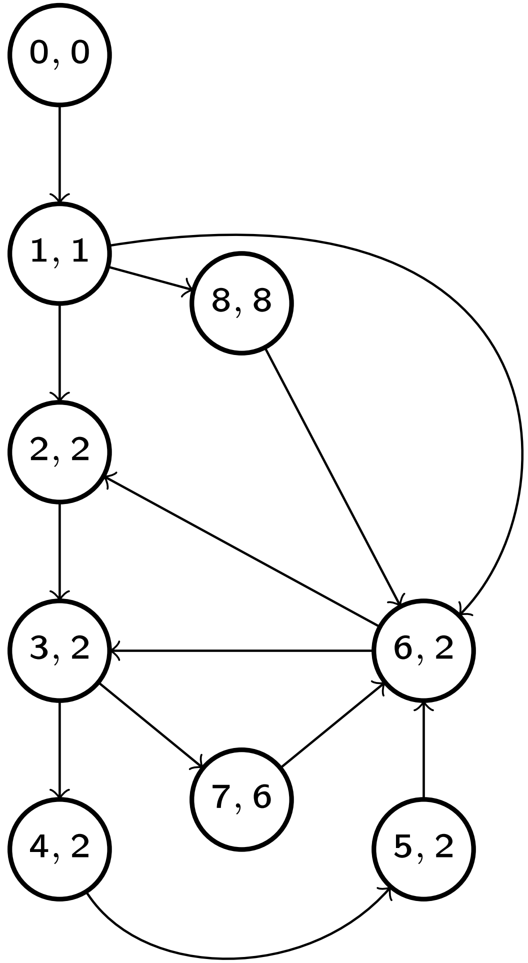
- 0 is its own SCC.

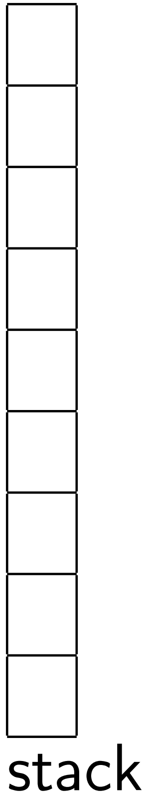
```
int      dfnum

procedure strong_connect(v)
    dfn(v) ← dfnum
    lowlink(v) ← dfnum
    visited(v) ← true
    push(v)
    dfnum ← dfnum +1

    for each w ∈ succ(v) do
        if (not visited(w)) {
            strong_connect(w)
            lowlink(v) ← min(lowlink(v), lowlink(w))
        } else if (dfn(w) < dfn(v)  and w is on stack)
            lowlink(v) ← min(lowlink(v), dfn(w))

    if (lowlink(v) = dfn(v))
        scc ← ∅
        do
            w ← pop()
            add w to scc
        while (w ≠ v)
        process_scc(scc)
end
```





Tarjan's Algorithm: Remarks

- Consider the edge (v, w).
- When w is not yet visited we must visit it by calling $strong\_connect(w)$.
- If w has been visited, we have two main cases:
  1. w is not on the stack, because it has already found its SCC.
  2. w is on the stack, because it's waiting for being popped.
     - If $dfn(w) < dfn(v)$ then v must set its lowlink so it does not think it is its own SCC.
     - If $dfn(w) \geq dfn(v)$ then no more information for v is available. There is another path from v to w due to which they will belong to the same SCC.

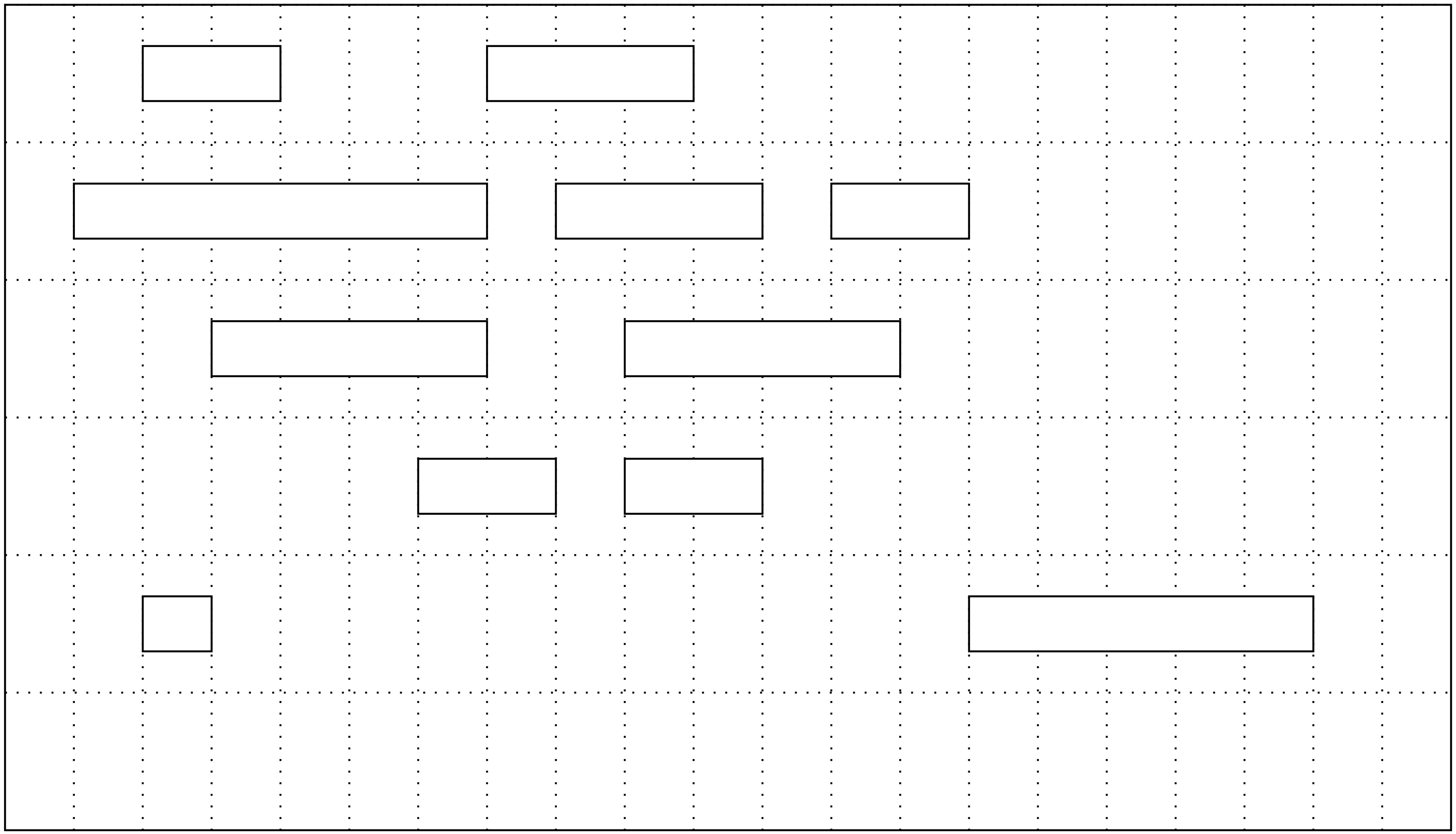
Greedy algorithms / Giriga algoritmer

- It is not trivial to define precisely what makes an algorithm greedy.
- The main idea is to use a simple rule to make decisions without taking "all" information into account.
- The challenge is to find a simple rule which solves a problem optimally
- Two approaches to prove that a greedy algorithm is optimal:
  1. The greedy algorithm "stays ahead" — by proving it is always at least as good as an optimal algorithm
  2. Exchange argument (utbytesargument) — transform the output of an optimal algorithm (without changing its quality) to the output of the greedy algorithm

Interval scheduling / Intervallschemaläggning

- One resource
- A set R of requests, r_i, with a start time $s(i)$ and a finish time $f(i)$
- A set of requests is **compatible** if they do not overlap in time
- The **interval scheduling problem** is to find the largest subset $S \subseteq R$ such that S is compatible
- All requests have equal value and it is the size of S we want to maximize
- A compatible set of maximum size is called an optimal schedule

An example set R



A greedy algorithm for interval scheduling

procedure *schedule*(R)
$S \leftarrow \emptyset$ /* S is a sequence */
while $R \neq null$
    $r \leftarrow$ select a request from R
    remove r from R
    add r to the end of S
    remove all request in R which overlap with r
return S

- Our problem is to figure out a clever *select* function
- Any suggestions?

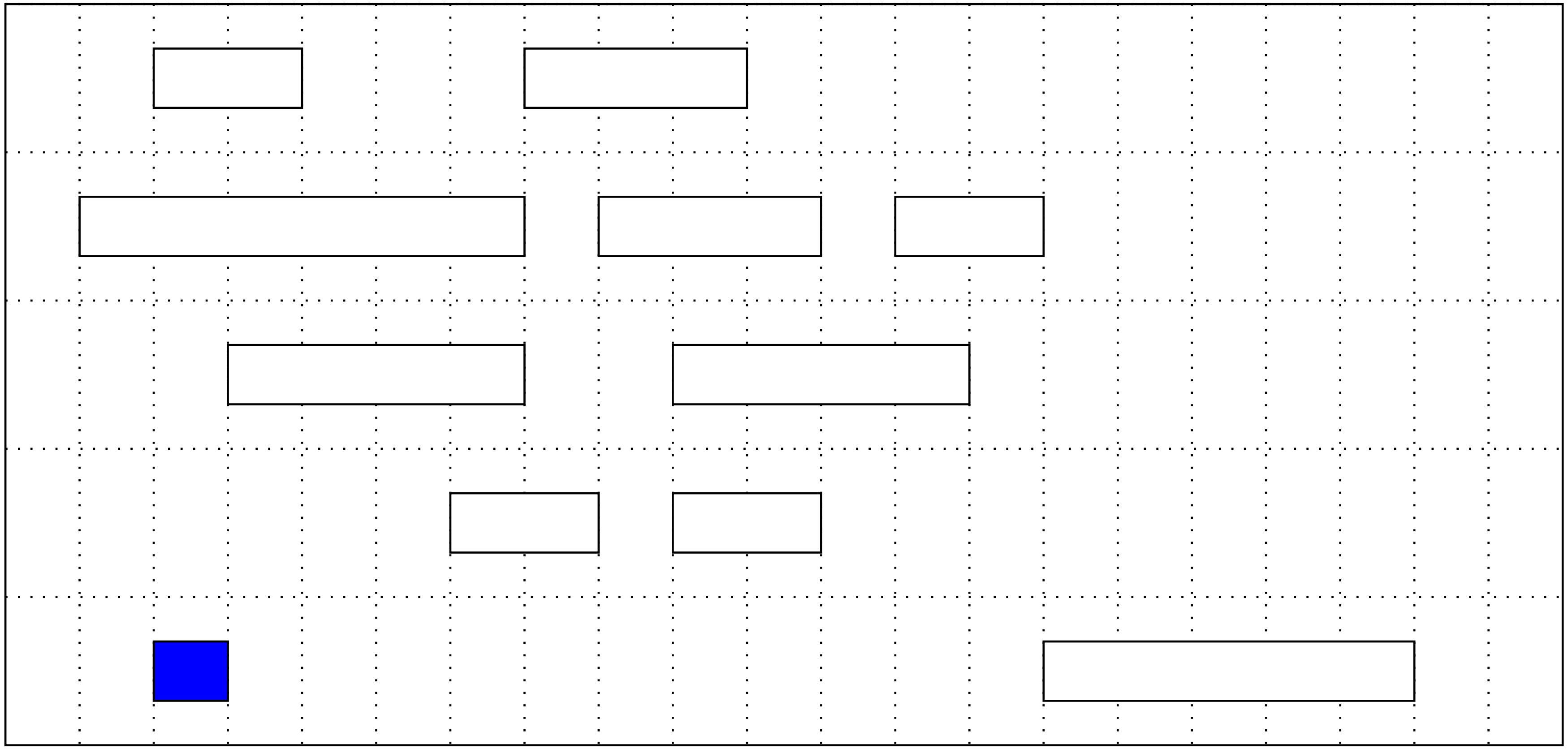
Ideas for the select function

- Take the request with shortest interval
- Take the request which starts first
- Take the request with fewest conflicts

None of these lead to an optimal solution

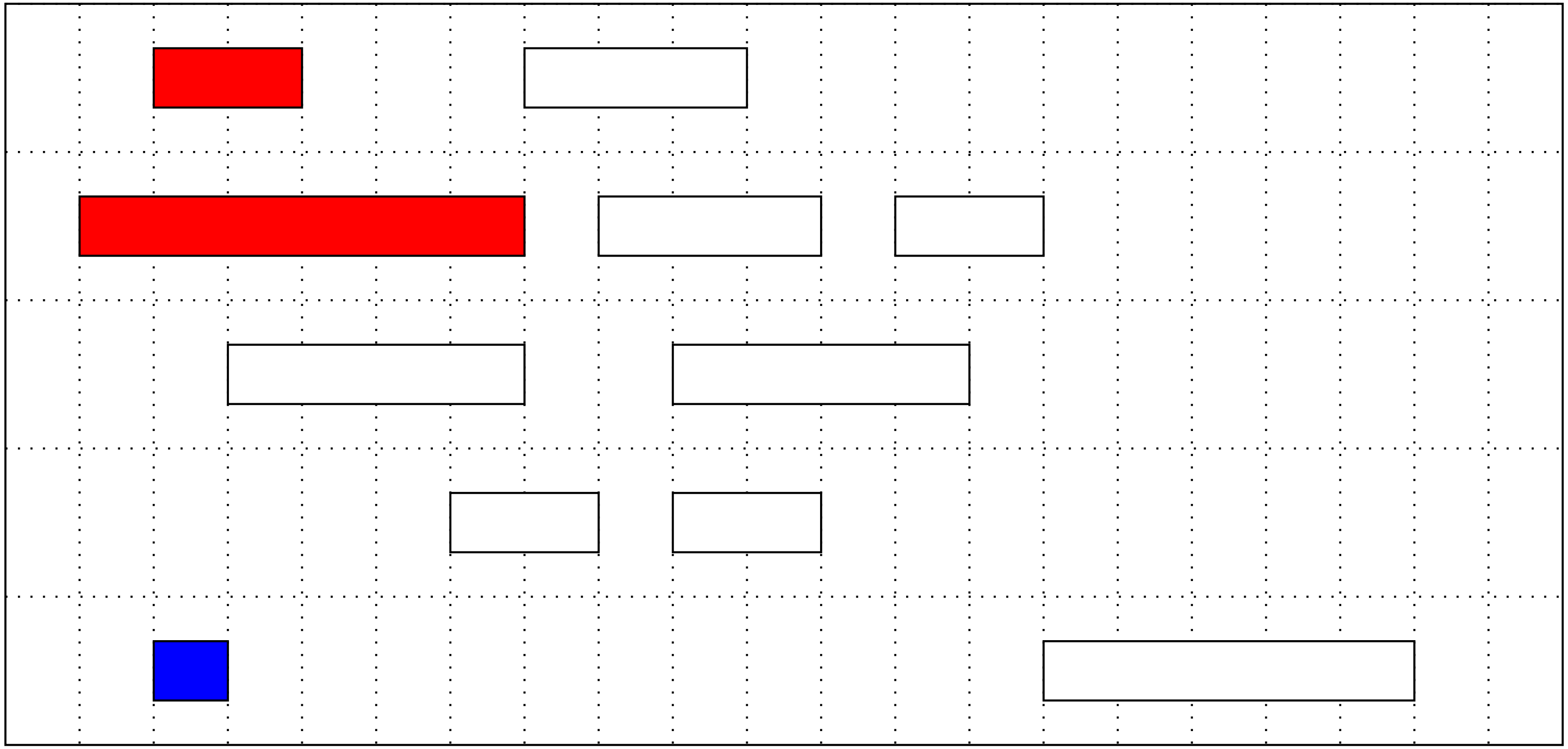
A better select function

- Take the request which finishes earliest
- Is this optimal?
- Select first request:



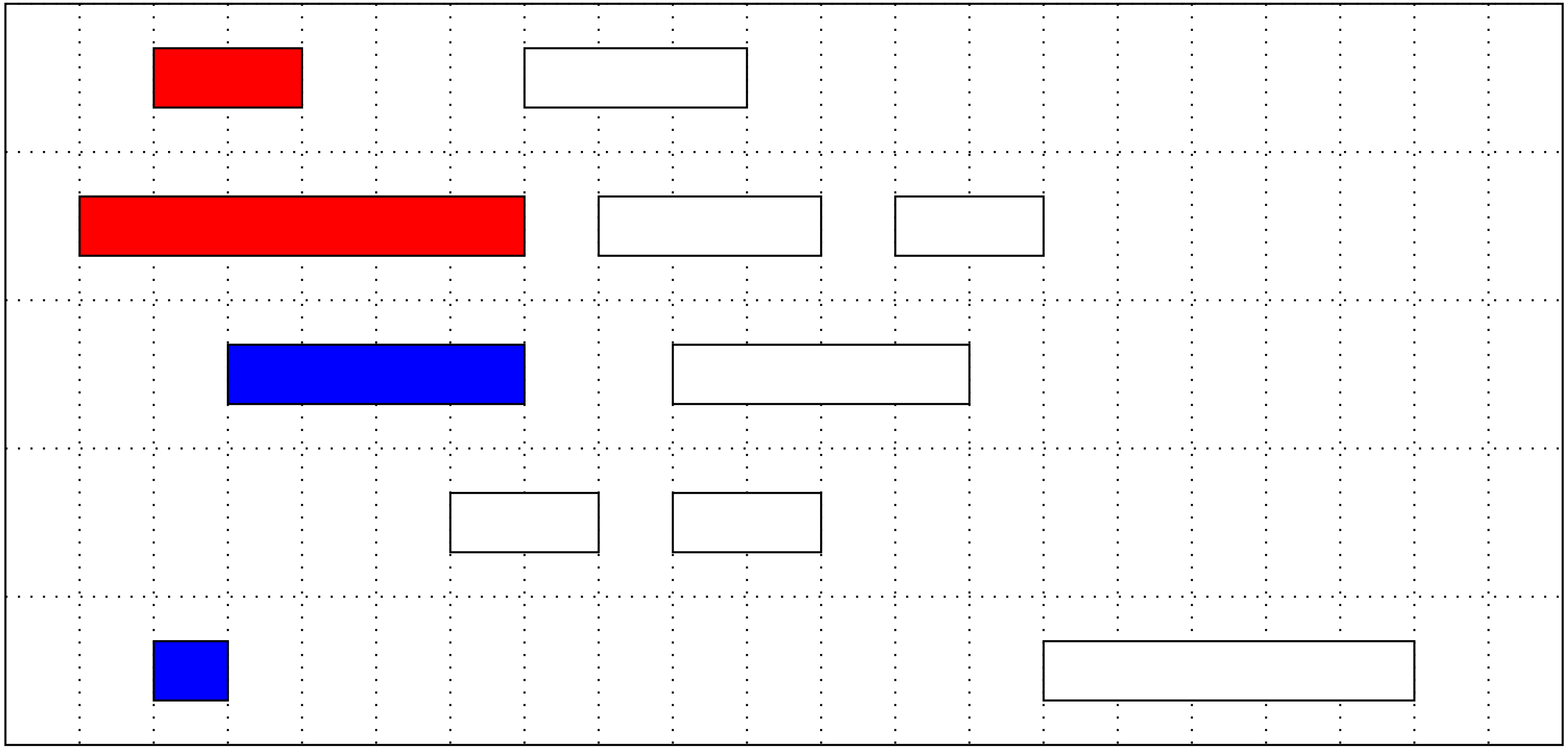
Select request which finishes first

- Remove incompatible requests:



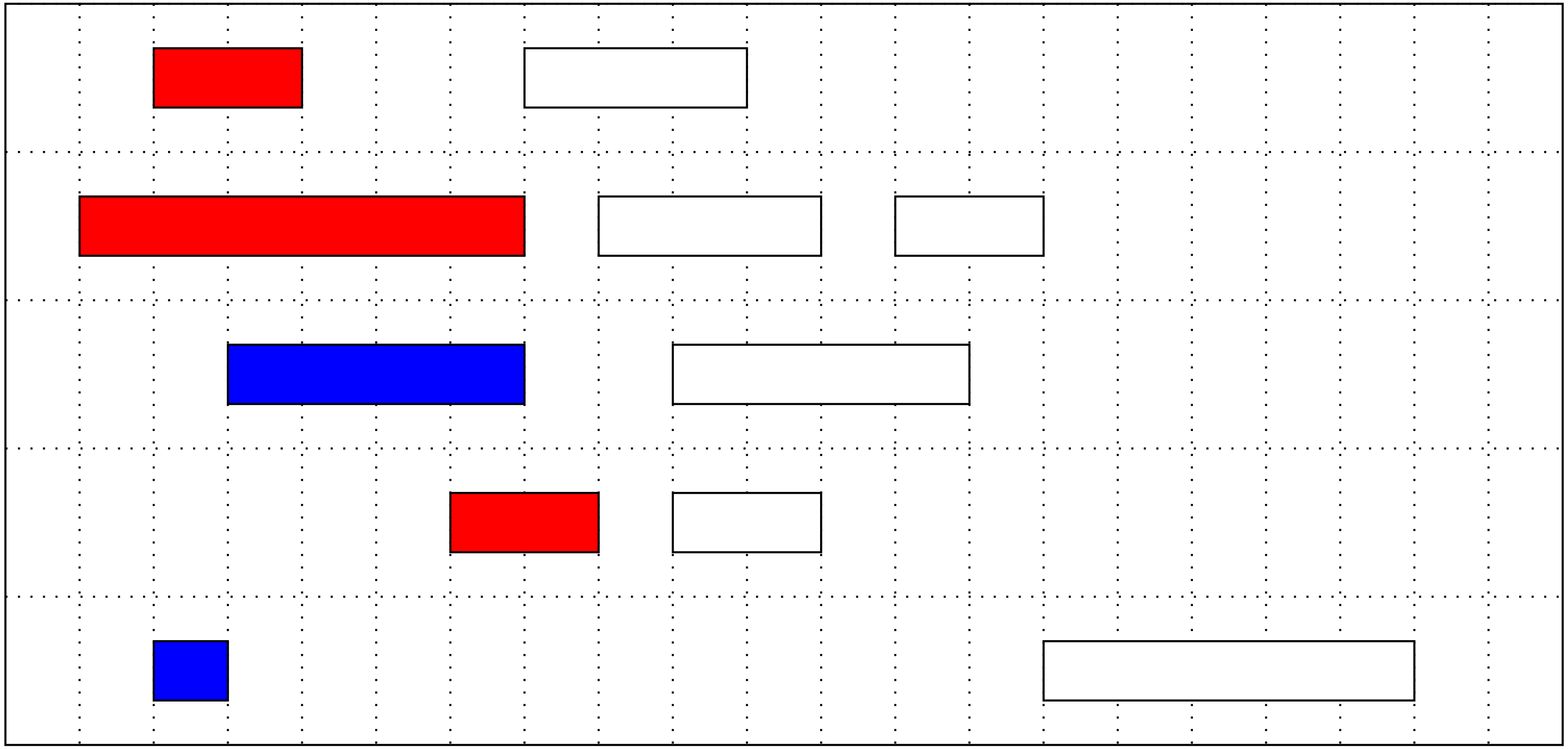
Select request which finishes first

- Select next request:



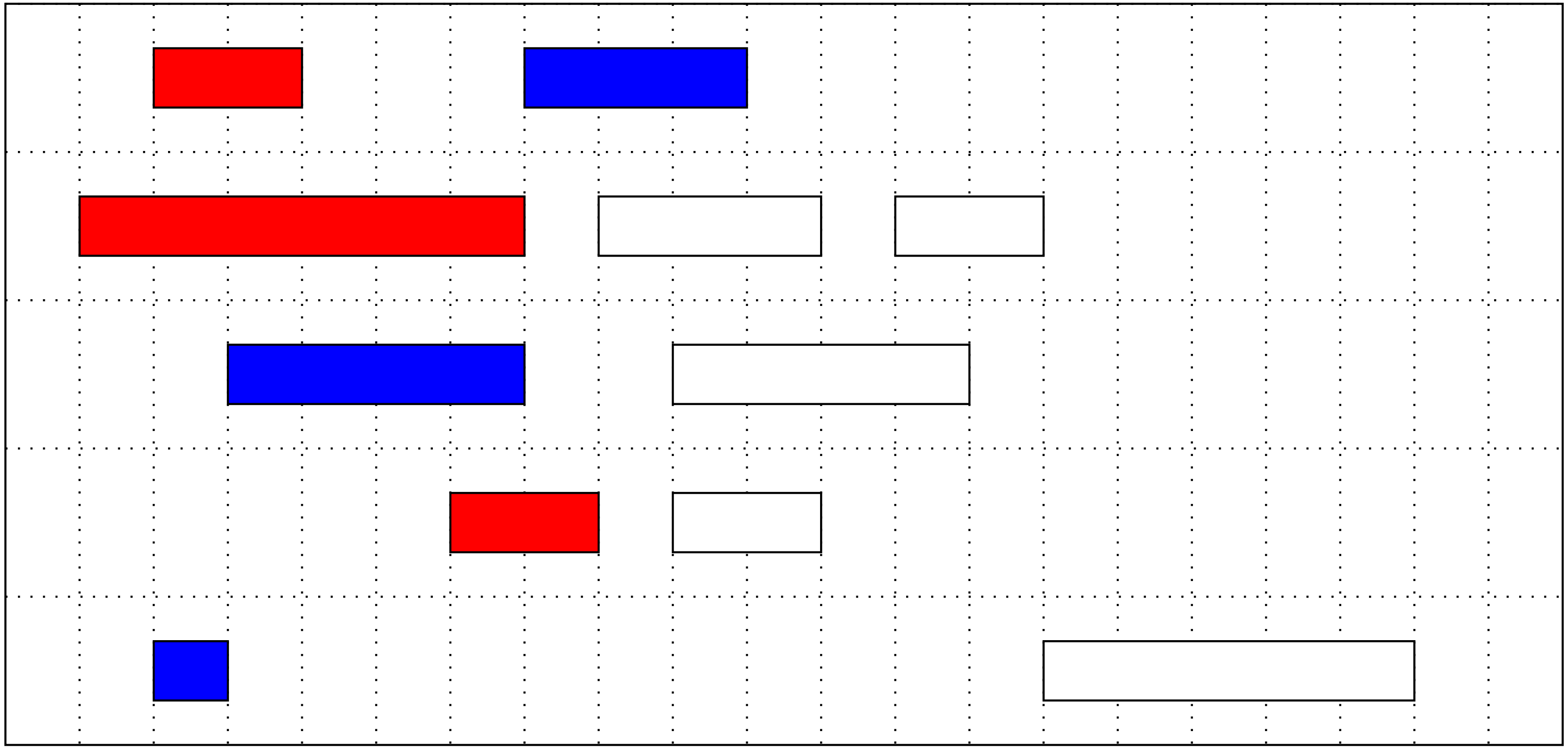
Select request which finishes first

- Remove incompatible request:



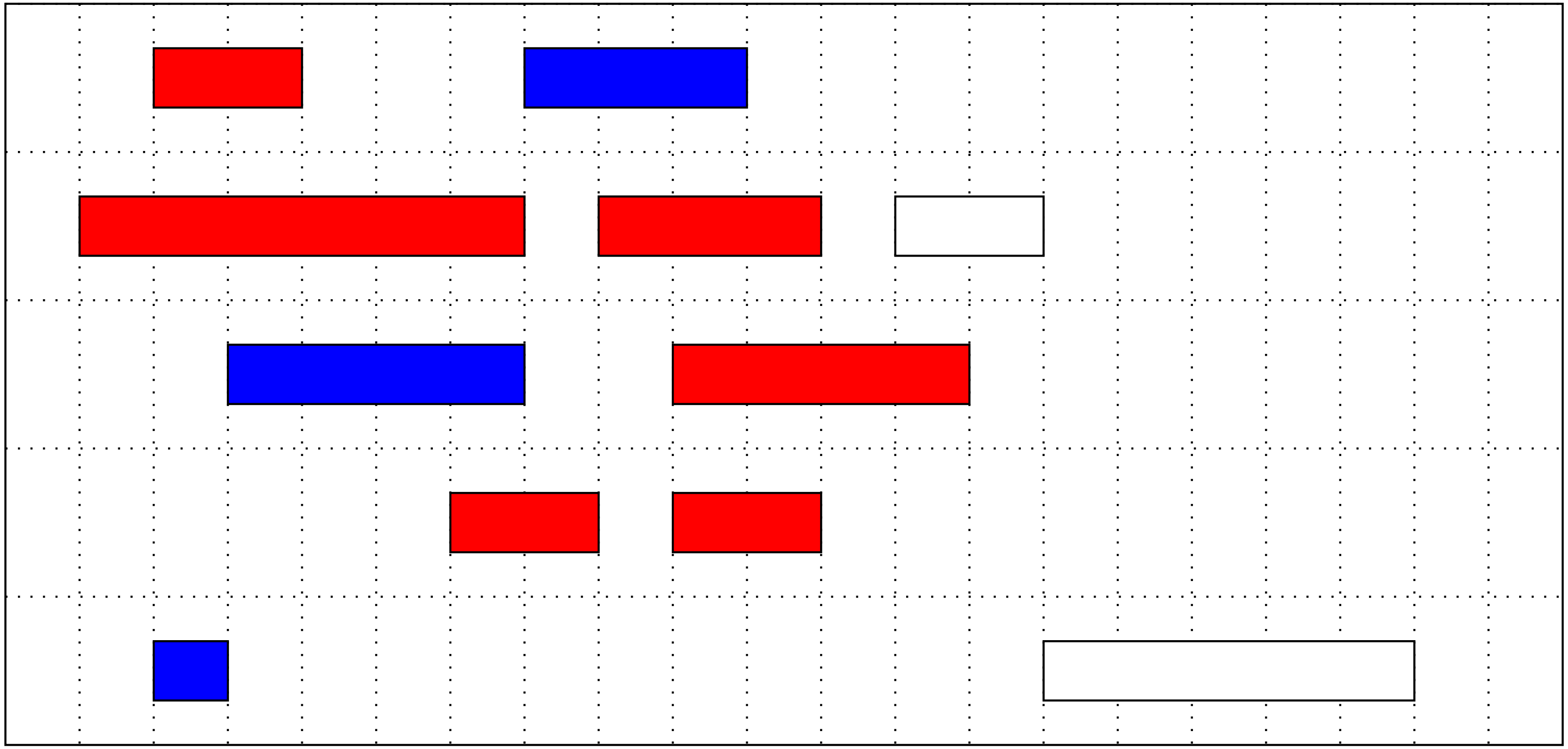
Select request which finishes first

- Select next request:



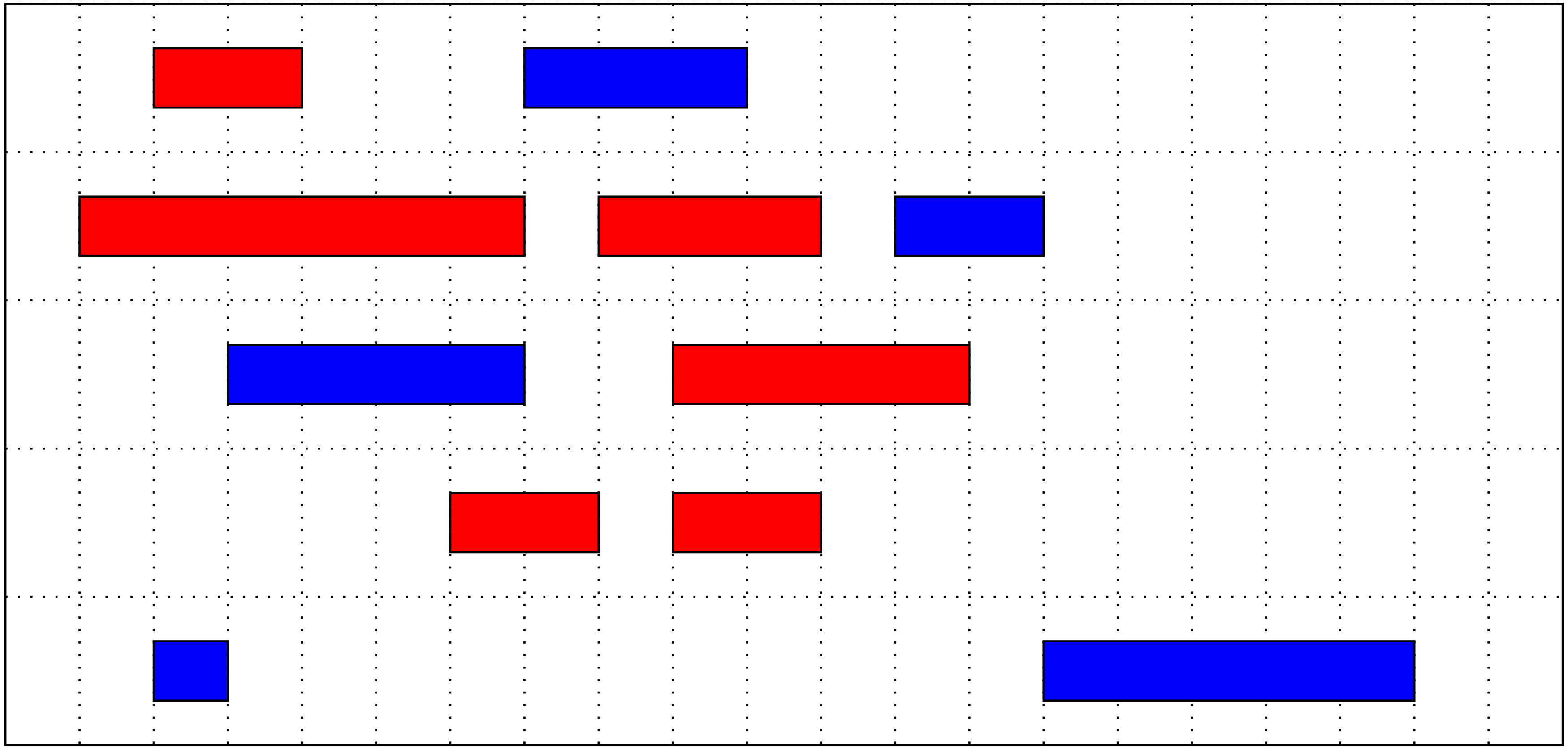
Select request which finishes first

- Remove incompatible requests:



Select request which finishes first

- End result:



Proving optimality of a greedy algorithm

- What should we prove?
- Can there exist several optimal schedules?
- Either show our algorithm is as good as an optimal (stays ahead), or output from an optimal algorithm can be transformed to the output of our algorithm.
- For our problem, assume there is an optimal schedule represented as a sequence T sorted in order of increasing finish time
- We should not try to prove $S = T$
- Instead we should prove $|S| = |T|$
- We will use the first proof technique:
  "our algorithm stays ahead of an optimal solution"
- That is, $|S| \geq |T|$

Two solutions

- Our: $S = (r_1, r_2, ..., r_n)$
- Optimal: $T = (t_1, t_2, ..., t_m)$
- We want to show $n = m$
- S and T are both sorted by increasing finish time
- It is clear that $f(r_1) \leq f(t_1)$ since we select the request with earliest finish time
- There are at least n requests in T so we can aim at proving:
- $f(r_k) \leq f(t_k) \quad 1 \leq k \leq n$

Comparing the solutions

Lemma

$f(r_k) \leq f(t_k) \quad 1 \leq k \leq n$

Proof.

- Proof by induction. $f(r_1) \leq f(t_1)$ is clear.
- For $k > 1$, assume (1): $f(r_{k-1}) \leq f(t_{k-1})$.
- Since T is compatible (2): $f(t_{k-1}) \leq s(t_k)$
- From (1) and (2) follows (3): $f(r_{k-1}) \leq s(t_k)$
- Our algorithm can select t_k as its r_k
- Our algorithm selects as r_k the request with earliest finish time, i.e. $f(r_k) \leq f(t_k)$

$\square$

$|S| = |T|$

- It remains to prove that $|S| = |T|$.
- Recall $n = |S|$ and $m = |T|$.

Theorem

$|S| = |T|$.

Proof.

- Assume in contradiction that $m > n$.
- We know $f(r_n) \leq f(t_n)$
- Since $m > n$, T contains a request t_{n+1}
- We must have $s(t_{n+1}) \geq f(t_n) \geq f(r_n)$
- But this request should have been scheduled by our algorithm which contradicts the assumption that $|S| = n$ so $m = n$ and optimality has been proved

$\square$

Time complexity

- Our greedy algorithm can be implemented in time $O(n \log n)$
- First all requests are sorted in order of increasing finish time
- Then a linear pass finds the schedule

Scheduling to minimize delays

- Again one resource
- Consider now requests with a soft deadline $d(r)$ and a time length $t(r)$
- It is not a disaster to fail a soft deadline compared with a hard deadline
- $s(r)$ and $f(r)$ are start and finish times and in this problem they are output and not input
- The delay of one request is $\max(0, f(r) - d(r))$
- Our problem is to schedule requests so that the maximum delay of any request is minimized
- What is a simple rule to do that optimally?

Algorithm: select next with earliest deadline

- Request are renamed so that $d(r_1) \leq d(r_2)... \leq d(r_n)$
- Our algorithm simply is to schedule the requests in this order, or in other words, **the earliest deadline first**
- In addition, we schedule requests so that $s(r_{i+1}) = f(r_i)$, i.e. without a gap between r_i and r_{i+1}
- Thus there is no **idle time** between any two requests
- Consider any optimal schedule T. Can it have idle time?
- Yes, e.g. if $t(r_1) = 1, d(r_1) = 2, t(r_2) = 3, d(r_2) = 10$
- With $s(r_1) = 1, f(r_1) = 2, s(r_2) = 7, d(r_2) = 10$, the maximum delay is zero
- But there obviously exists a different optimal schedule without any gap — just start the requests in the same order but as early as possible

Inversions

- Let $d(r_i) < d(r_j)$
- If r_j is scheduled before r_i, it is called an **inversion**
- Our algorithm creates no inversions
- But if $d(r_i) = d(r_{i+1})$ then it can schedule r_{i+1} before r_i

First part of our optimality proof

Lemma

All schedules with no idle time and no inversions have the same maximum delay

Proof.

- Consider two different schedules S and T without idle time and no inversions
- Then the only difference between S and T is the order in which requests with identical deadlines are scheduled
- In a sequence with such requests, the last has the maximal delay or no delay.
- The maximal delay is the same in both schedules.

$\square$

An example

- Let all requests take one time unit
- Let all requests have deadline $d(r_i) = 3$
- $S = (r_1, r_2, r_3, r_4, r_5)$
- $T = (r_4, r_5, r_1, r_2, r_3)$
- In S r_1, r_2, r_3 have no delay, r_4 is delayed 1 and r_5 is delayed 2
- In T r_4, r_5, r_1 have no delay, r_2 is delayed 1 and r_3 is delayed 2
- Same maximum delay but for different requests

Second part

- We will prove that an optimal schedule T can be transformed to S

Lemma

Assume an optimal schedule Q has an inversion of r_i and r_j i.e. $d_i < d_j$ and r_j is scheduled before r_i. Then Q has a pair r_a and r_b of inverted requests scheduled immediately after each other.

- In $(7, 8, 9, 1, 2, 3, 4, 5, 6)$ with $i = 1, j = 7$ we have $a = 1, b = 9$.

Proof.

- We have $(r_j, \ldots, r_i)$ with k requests scheduled between r_j and r_i, $k \geq 0$
- In this sequence of $k + 2$ requests, since $d_i < d_j$, there must be a first pair r_a and r_b of inverted requests with r_b scheduled immediately before r_a such that $d_a < d_b$.

$\square$

Third part

Lemma

Let r_a and r_b be a pair of inverted requests scheduled consecutively in Q Swapping r_a and r_b into R, the maximum delay does not increase.

Proof.

- We have $Q = (..., r_b, r_a, ...)$ and $R = (..., r_a, r_b, ...)$.
- The delay of r_a cannot have increased in R
- Let $f^X(r)$ be the finish time in schedule X for request r
- Let $t = f^Q(r_a) = f^R(r_b)$ (i.e. finish time of last of them)
- The delay of r_a in Q is $f^Q(r_a) - d(r_a) = t - d(r_a)$
- The delay of r_b in R is $f^R(r_b) - d(r_b) = t - d(r_b)$
- Can r_b now be more late than r_a was? I.e. can $t - d(r_b) > t - d(r_a)$?
- $t - d(r_b) > t - d(r_a) \iff t + d(r_a) > t + d(r_b)$
- No, since by assumption $d(r_a) < d(r_b)$

Last part

Theorem

Our algorithm produces a schedule with minimum maximum delay.

Proof.

- We have just shown that there exists an optimal schedule T with no idle time and no inversions.
- Since all schedules with no idle time and no inversions have the same maximum delay, our algorithm is optimal.

□

An exchange argument

- What did we do?
- We find an algorithm which seems to be optimal.
- We characterize optimal solutions.
- We exchange optimal solutions to produce an output which is identical to ours.
- Therefore our algorithm is optimal.
- Note that the schedules may be different but the minimum maximum delay is the same.