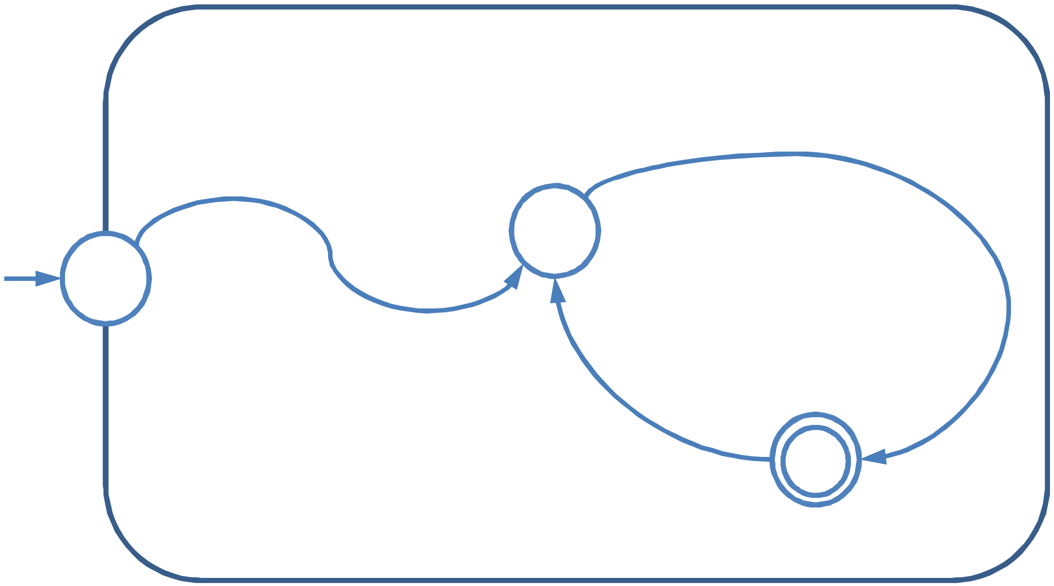
Checking emptiness of generalized Büchi automata

Accepting lassos

- A NBA is nonempty iff it has an accepting lasso



- For NGA: the ``loop part'' must visit all sets of accepting states.

Setting

- We want on-the-fly algorithms that search for an accepting lasso of a given NBA while constructing it.
- The algorithms know the initial state, and have access to an oracle that, called with a state q returns all successors of q (and for each successor whether it is accepting or not).
- We think big: the NBA may have tens of millions of states.

Two approaches

1. Compute the set of accepting states, and for each accepting state, check if it belongs to some cycle.

   Nested-depth-first-search algorithm

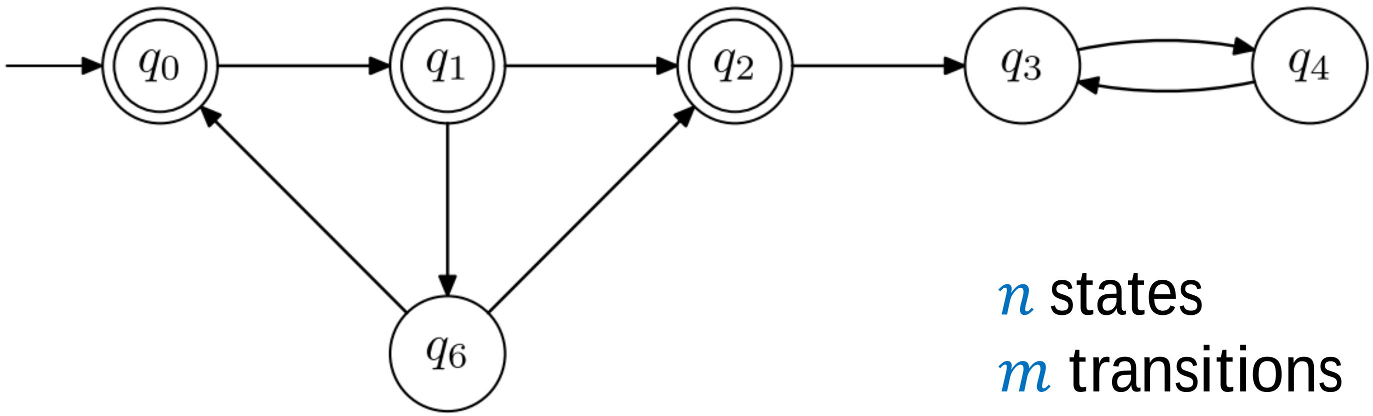
2. Compute the set of states that belong to some cycle, and for each such set, check if is accepting.

   SCC-based algorithm

First approach: A naïve algorithm

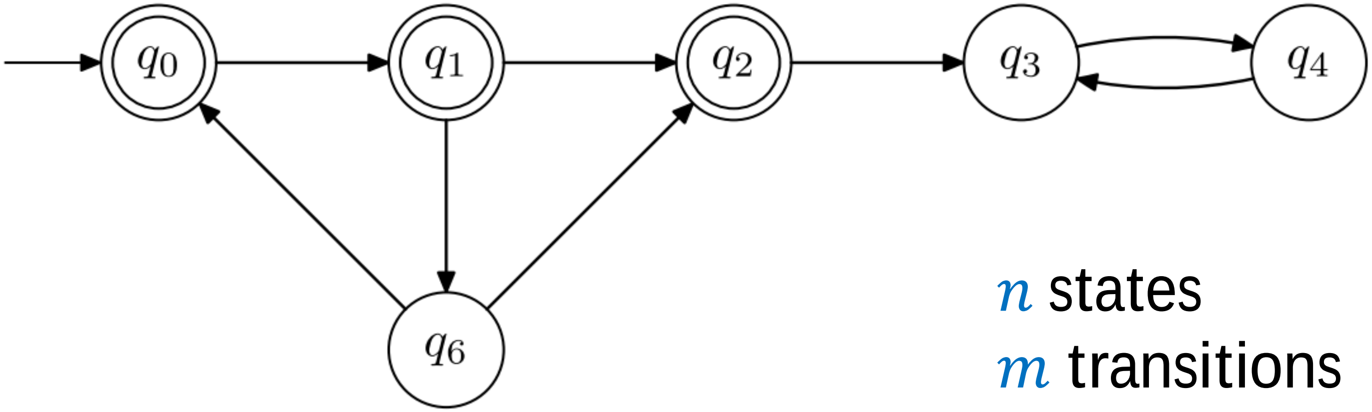
1. Compute the set of accepting states by means of a graph search (DFS, BFS, ...).
2. For each accepting state q, conduct a second search (DFS, BFS,...) starting at q to decide if q belongs to a cycle.

First approach: A naïve algorithm



n states
m transitions

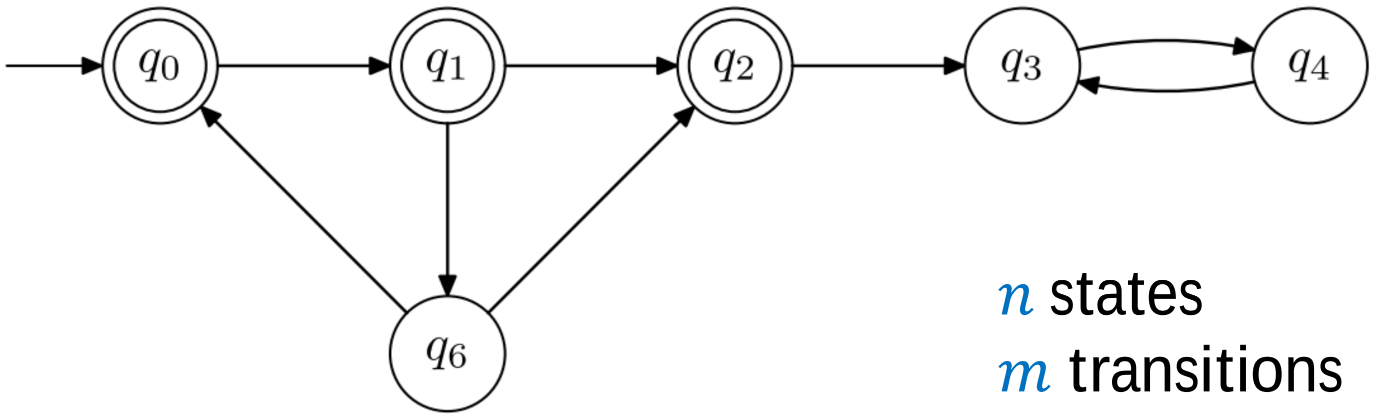
First approach: A naïve algorithm



n states
m transitions

Runtime of the first search: $O(m)$

First approach: A naïve algorithm

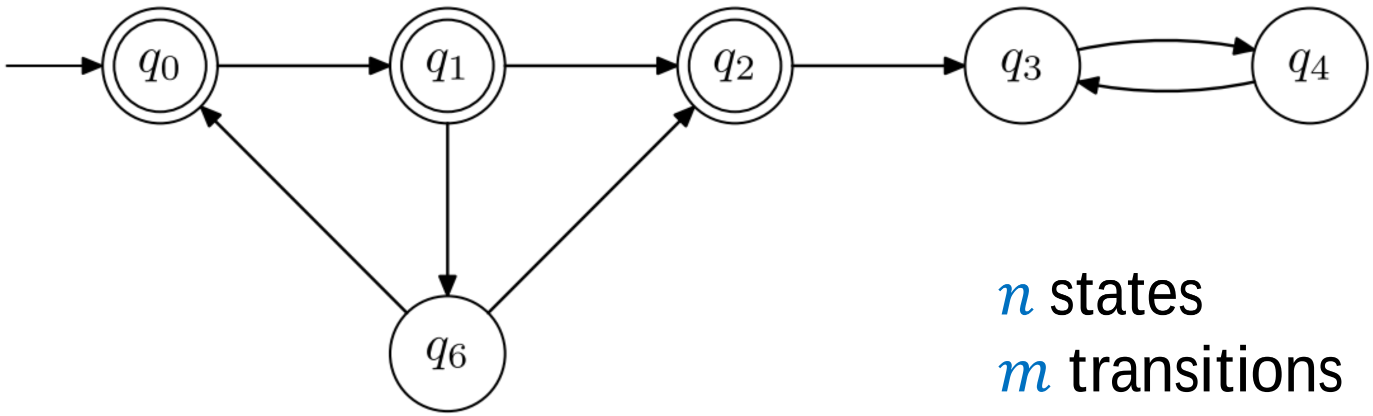


n states
m transitions

Runtime of the first search: $O(m)$

Number of searches in the second step: $O(n)$

First approach: A naïve algorithm



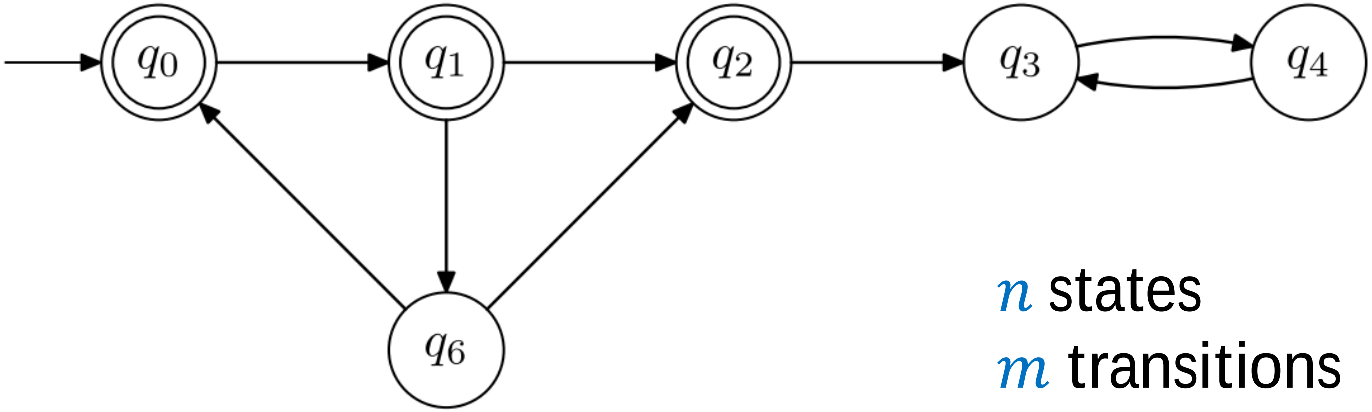
n states
m transitions

Runtime of the first search: $O(m)$

Number of searches in the second step: $O(n)$

Overall runtime of the second step: $O(nm)$

First approach: A naïve algorithm



n states
m transitions

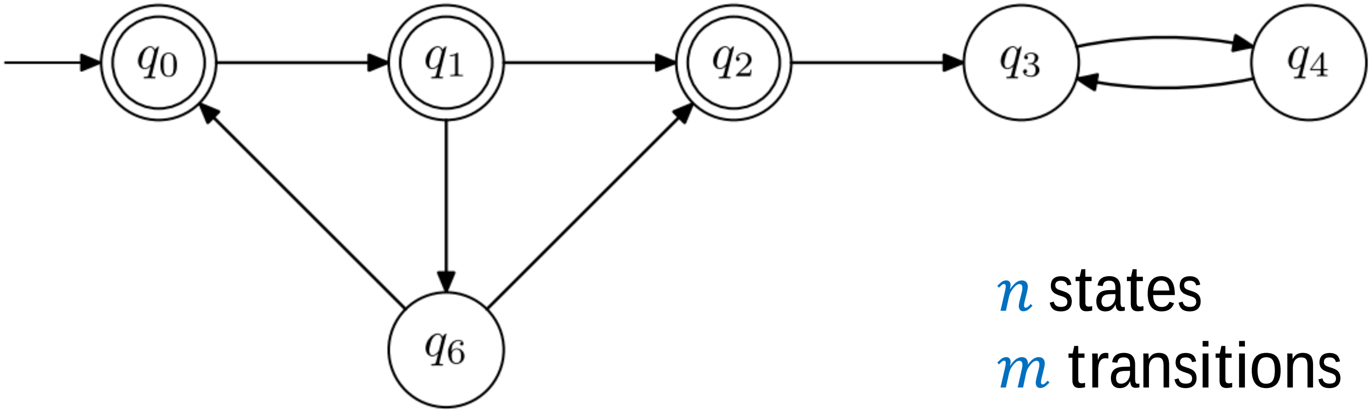
Runtime of the first search: $O(m)$

Number of searches in the second step: $O(n)$

Overall runtime of the second step: $O(nm)$

Overall runtime: $O(nm)$. Too high!

First approach: A naïve algorithm



n states
m transitions

Runtime of the first search: $O(m)$

Number of searches in the second step: $O(n)$

Overall runtime of the second step: $O(nm)$

Overall runtime: $O(nm)$. Too high!

We want an $O(m)$ algorithm.

Generic search in graphs

- Similar to a workset algorithm

Generic search in graphs

- Similar to a workset algorithm
- Initially the workset contains only the initial state. At every iteration:

Generic search in graphs

- Similar to a workset algorithm
- Initially the workset contains only the initial state. At every iteration:
  - Choose a state from the workset and mark it as discovered (but don’t remove it yet).

Generic search in graphs

- Similar to a workset algorithm
- Initially the workset contains only the initial state. At every iteration:
  – Choose a state from the workset and mark it as discovered (but don't remove it yet).
  – If all successors of the state have already been discovered, then remove the state from the workset.

Generic search in graphs

- Similar to a workset algorithm
- Initially the workset contains only the initial state. At every iteration:
  - Choose a state from the workset and mark it as discovered (but don't remove it yet).
  - If all successors of the state have already been discovered, then remove the state from the workset.
  - Otherwise, choose a not-yet-discovered successor and add it to the workset.

Generic search in graphs

- Similar to a workset algorithm
- Initially the workset contains only the initial state. At every iteration:
  - Choose a state from the workset and mark it as discovered (but don't remove it yet).
  - If all successors of the state have already been discovered, then remove the state from the workset.
  - Otherwise, choose a not-yet-discovered successor and add it to the workset.
- Depth-first search: workset is implemented as a stack (first in last out)

Generic search in graphs

- Similar to a workset algorithm
- Initially the workset contains only the initial state. At every iteration:
  - Choose a state from the workset and mark it as discovered (but don't remove it yet).
  - If all successors of the state have already been discovered, then remove the state from the workset.
  - Otherwise, choose a not-yet-discovered successor and add it to the workset.
- Depth-first search: workset is implemented as a stack (first in last out)
- Breadth-first search: workset is implemented as a queue (first in first out)

Depth-first search: Terminology

- States are discovered by the search.

Depth-first search: Terminology

- States are discovered by the search.
- After recursively exploring all successors, the search backtracks from the state.

Depth-first search: Terminology

- States are discovered by the search.
- After recursively exploring all successors, the search backtracks from the state.
- The search assigns to a state q:
  - a discovery time $d[q]$;

Depth-first search: Terminology

- States are discovered by the search.
- After recursively exploring all successors, the search backtracks from the state.
- The search assigns to a state q:
  - a discovery time $d[q]$;
  - a finishing time $f[q]$;

Depth-first search: Terminology

- States are discovered by the search.
- After recursively exploring all successors, the search backtracks from the state.
- The search assigns to a state q:
  - a discovery time $d[q]$;
  - a finishing time $f[q]$;
  - a DFS-predecessor, the state from which q is discovered (DFS-tree).

Depth-first search: Terminology

- States are discovered by the search.
- After recursively exploring all successors, the search backtracks from the state.
- The search assigns to a state q:
  - a discovery time $d[q]$;
  - a finishing time $f[q]$;
  - a DFS-predecessor, the state from which q is discovered (DFS-tree).
- Coloring scheme: at time t state q is either
  - white: not yet discovered, $1 \leq t \leq d[q]$

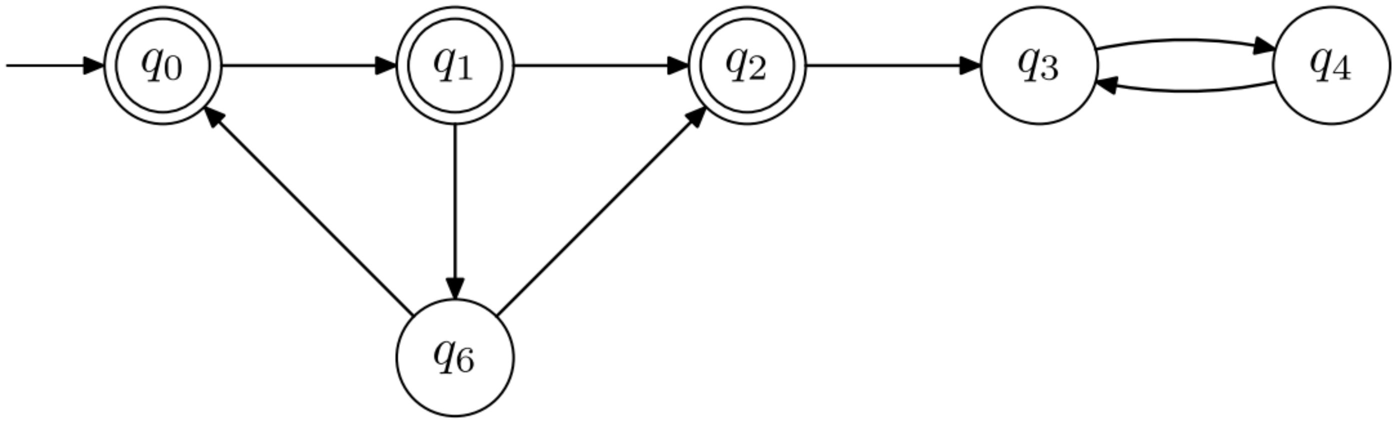
Depth-first search: Terminology

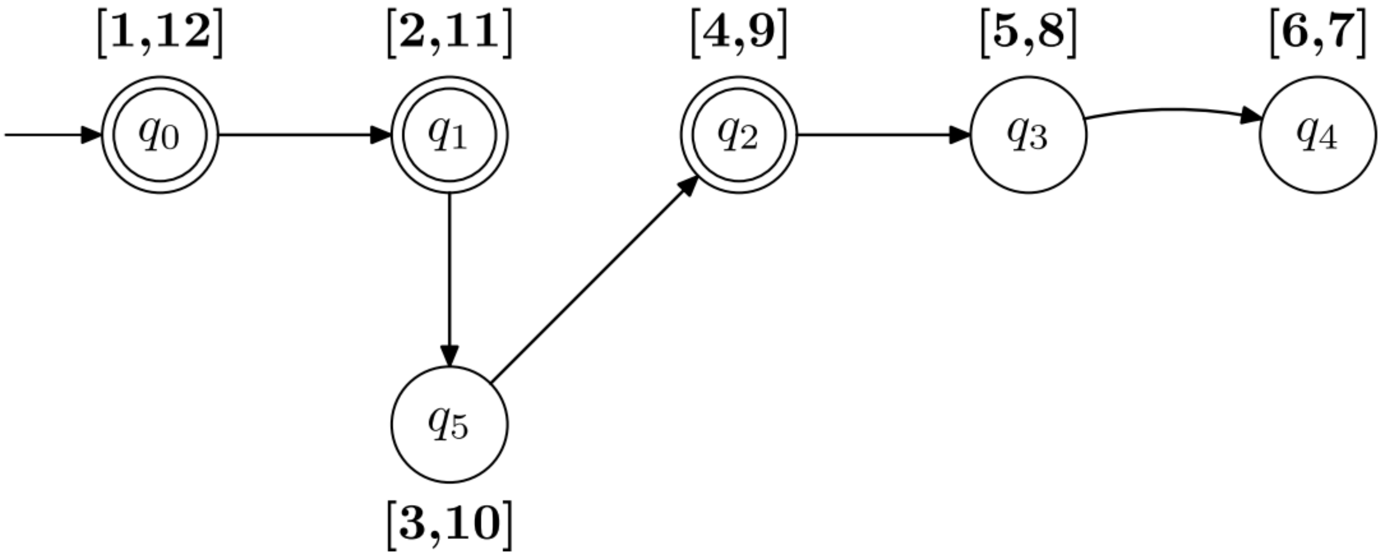
- States are discovered by the search.
- After recursively exploring all successors, the search backtracks from the state.
- The search assigns to a state q:
  - a discovery time $d[q]$;
  - a finishing time $f[q]$;
  - a DFS-predecessor, the state from which q is discovered (DFS-tree).
- Coloring scheme: at time t state q is either
  - white: not yet discovered, $1 \leq t \leq d[q]$
  - grey: discovered, but at least one successor not yet fully explored, $d[q] < t \leq f[q]$

Depth-first search: Terminology

- States are discovered by the search.
- After recursively exploring all successors, the search backtracks from the state.
- The search assigns to a state q:
  - a discovery time $d[q]$;
  - a finishing time $f[q]$;
  - a DFS-predecessor, the state from which q is discovered (DFS-tree).
- Coloring scheme: at time t state q is either
  - white: not yet discovered, $1 \leq t \leq d[q]$
  - grey: discovered, but at least one successor not yet fully explored, $d[q] < t \leq f[q]$
  - black: search has already backtracked from q, $f(q) < t \leq 2n$

An example





Recursive implementation of DFS

DFS(*A*)

Input: NBA $A = (Q, \Sigma, \delta, Q_0, F)$

$S \leftarrow \emptyset$
$dfs(q_0)$

proc $dfs(q)$
    add q **to** S
    for all $r \in \delta(q)$ **do**
        if $r \notin S$ **then** $dfs(r)$
    return

DFS_Tree(*A*)

Input: NBA $A = (Q, \Sigma, \delta, Q_0, F)$

Output: Time-stamped tree (S, T, d, f)

$S \leftarrow \emptyset$
$T \leftarrow \emptyset; t \leftarrow 0$
$dfs(q_0)$

proc $dfs(q)$
    $t \leftarrow t + 1; d[q] \leftarrow t$
    add q **to** S
    for all $r \in \delta(q)$ **do**
        if $r \notin S$ **then**
            add (q, r) **to** T; $dfs(r)$
    $t \leftarrow t + 1; f[q] \leftarrow t$
    return

Parenthesis theorem

- $I(q)$ denotes the interval $\left(d[q], f[q]\right]$.

Parenthesis theorem

- $I(q)$ denotes the interval $\left(d[q], f[q]\right]$.
- $I(q) \prec I(r)$ denotes that $f[q] < d[r]$ holds (i.e., $I(q)$ is to the left of $I(r)$ and does not overlap with it).

Parenthesis theorem

- $I(q)$ denotes the interval $\left(d[q], f[q]\right]$.
- $I(q) \prec I(r)$ denotes that $f[q] < d[r]$ holds (i.e., $I(q)$ is to the left of $I(r)$ and does not overlap with it).
- $q \Rightarrow r$ denotes that r is a DFS-descendant of q in the DFS-tree.

Parenthesis theorem

- $I(q)$ denotes the interval $(d[q], f[q]]$.
- $I(q) \prec I(r)$ denotes that $f[q] < d[r]$ holds (i.e., $I(q)$ is to the left of $I(r)$ and does not overlap with it).
- $q \Rightarrow r$ denotes that r is a DFS-descendant of q in the DFS-tree.
- Parenthesis theorem. In a DFS-tree, for any two states q and r, exactly one of the following conditions hold:
  - $I(q) \subseteq I(r)$ and $r \Rightarrow q$.
  - $I(r) \subseteq I(q)$ and $q \Rightarrow r$.
  - $I(q) \prec I(r)$, and none of q, r is a descendant of the other
  - $I(r) \prec I(q)$, and none of q, r is a descendant of the other

White-path and grey-path theorems

- White-path theorem. $q \Rightarrow r$ (and so $I(r) \subseteq I(q)$) iff at time $d[q]$ state r can be reached from q along a path of white states.

White-path and grey-path theorems

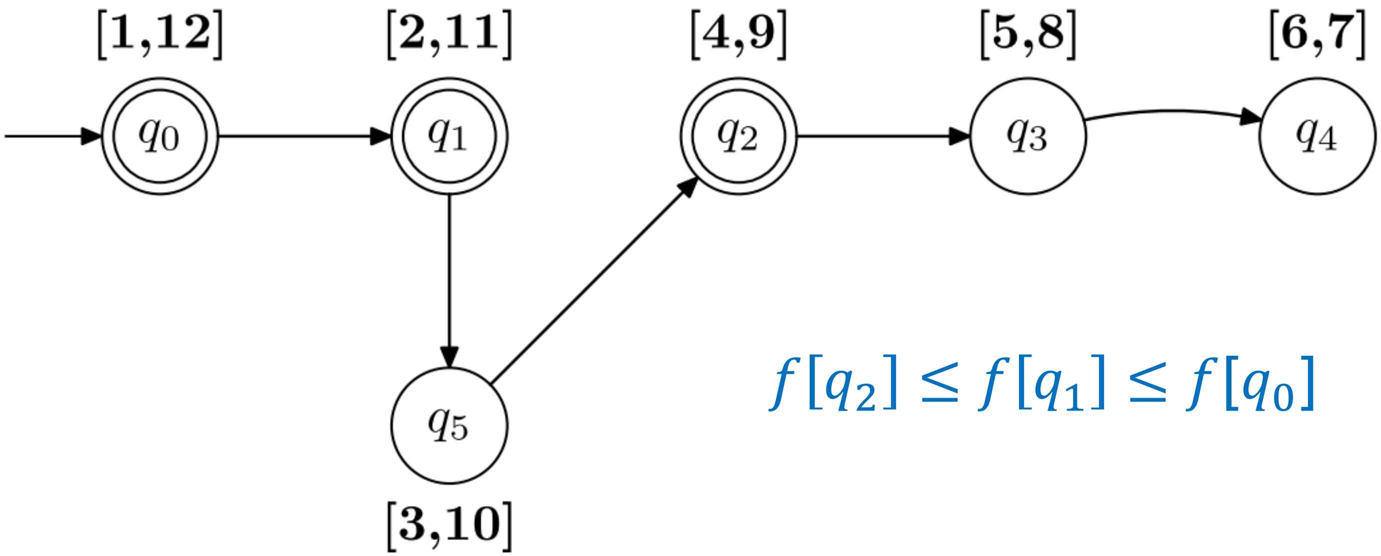
- White-path theorem. $q \Rightarrow r$ (and so $I(r) \subseteq I(q)$) iff at time $d[q]$ state r can be reached from q along a path of white states.
- Grey-path theorem. At every moment in time, all grey nodes form a simple path of the DFS tree (the grey path).

Nested-DFS algorithm

- Modification of the naïve algorithm:
  - Use a DFS to discover the accepting states and sort them in a certain order $q_1, q_2, \dots, q_k$;
  - conduct a DFS from each accepting state in the order $q_1, q_2, \dots, q_k$.
- The order will guarantee that if the search from q_j hits a state already discovered during the search from q_i, for some $i < j$, then the search can backtrack.
- Runtime: $O(m)$, because every transition is explored at most twice, once in each phase.

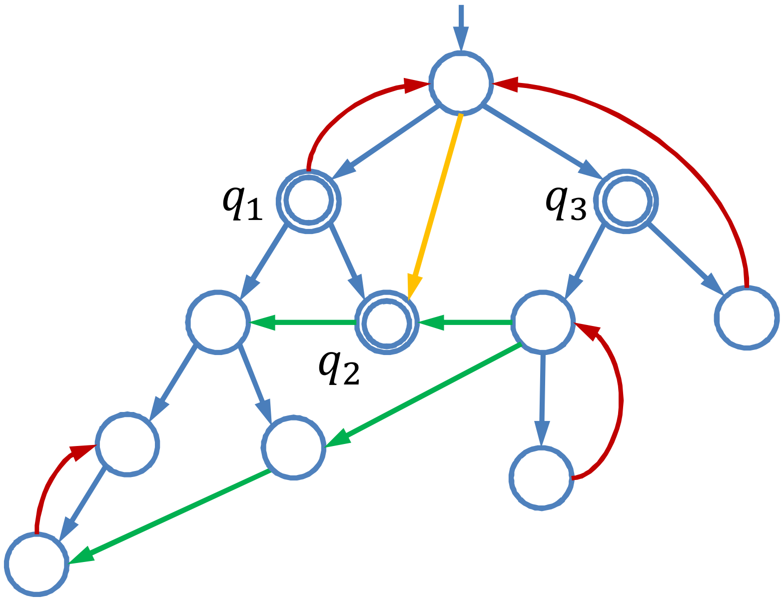
Nested-DFS algorithm

- Suitable order: postorder
- The postorder sorts the states according to increasing finishing time.

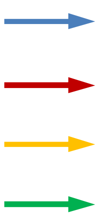


$f[q_2] \leq f[q_1] \leq f[q_0]$

Why does it work?

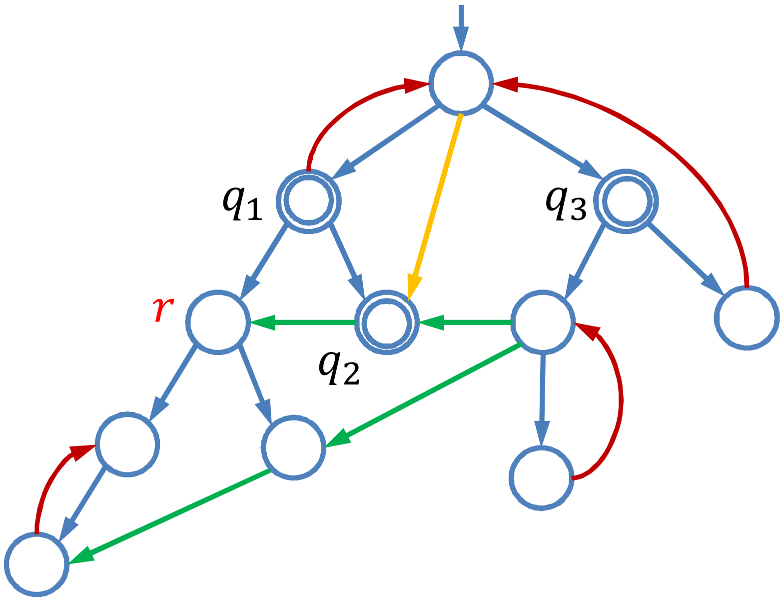


- Edges processed counterclockwise

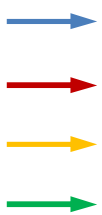


- $f[q_2] \leq f[q_1] \leq f[q_3]$

What do we have to prove?



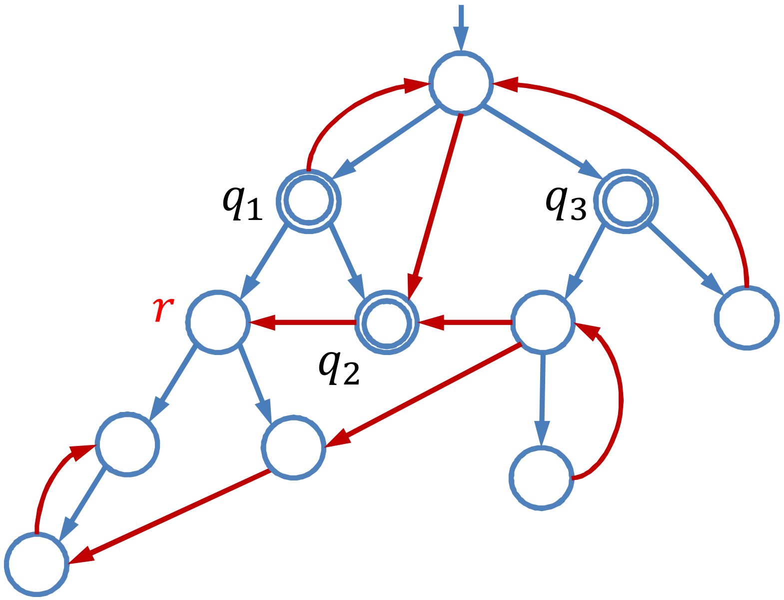
- Edges processed counterclockwise



- $f[q_2] \leq f[q_1] \leq f[q_3]$

- State r discovered during the search from q_2

What do we have to prove?



- Edges processed counterclockwise

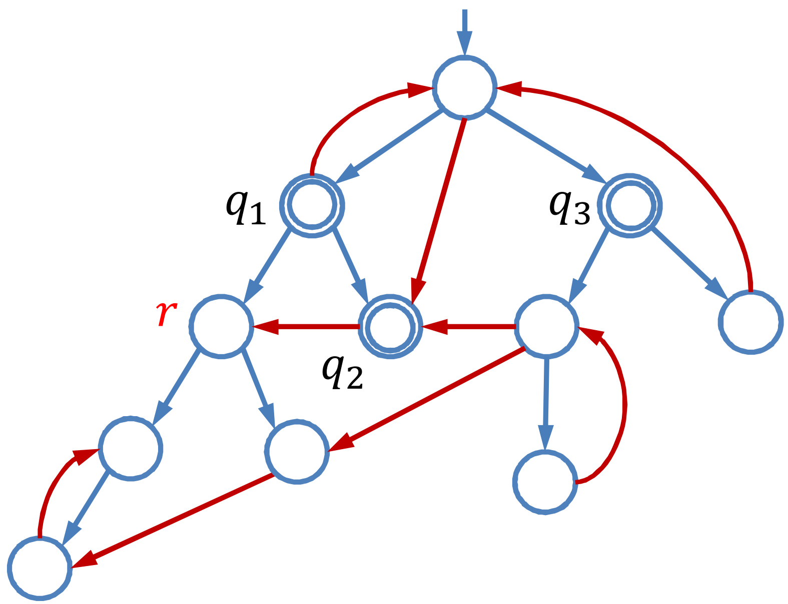
DFS-tree

Other edges

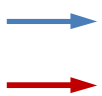
- $f[q_2] \leq f[q_1] \leq f[q_3]$

- State r discovered during the search from q_2

What do we have to prove?



- Edges processed counterclockwise



- $f[q_2] \leq f[q_1] \leq f[q_3]$

- State r discovered during the search from q_2
- To prove: during the search from q_1 (or q_3), it is safe to backtrack from r, because we do not “miss any accepting lassos”
- Amounts to: proving that q_1 (or q_3) is not reachable from r.

Correctness proof

Notation. $q \rightsquigarrow r$ denotes "q is reachable from r"

Correctness proof

Notation. $q \rightsquigarrow r$ denotes "q is reachable from r"

Lemma. If $q \rightsquigarrow r$ and $f[q] < f[r]$, then some cycle contains q.

Correctness proof

Notation. $q \rightsquigarrow r$ denotes "q is reachable from r"

Lemma. If $q \rightsquigarrow r$ and $f[q] < f[r]$, then some cycle contains q.

Proof: Let $\pi = q \rightarrow \cdots \rightarrow r$. Let s be the first node of π that is discovered (so $d[s] \leq d[q]$). We show $s \neq q$, $q \rightsquigarrow s$, and $s \rightsquigarrow q$.

Correctness proof

Notation. $q \rightsquigarrow r$ denotes "q is reachable from r"

Lemma. If $q \rightsquigarrow r$ and $f[q] < f[r]$, then some cycle contains q.

Proof: Let $\pi = q \to \cdots \to r$. Let s be the first node of π that is discovered (so $d[s] \le d[q]$). We show $s \neq q$, $q \rightsquigarrow s$, and $s \rightsquigarrow q$.

- $s \neq q$. Otherwise at time $d[q]$ the path π is white and so $I(r) \subseteq I(q)$, which contradicts $f[q] < f[r]$.

Correctness proof

Notation. $q \rightsquigarrow r$ denotes "q is reachable from r"

Lemma. If $q \rightsquigarrow r$ and $f[q] < f[r]$, then some cycle contains q.

Proof: Let $\pi = q \to \cdots \to r$. Let s be the first node of π that is discovered (so $d[s] \leq d[q]$). We show $s \neq q$, $q \rightsquigarrow s$, and $s \rightsquigarrow q$.

- $s \neq q$. Otherwise at time $d[q]$ the path π is white and so $I(r) \subseteq I(q)$, which contradicts $f[q] < f[r]$.
- $q \rightsquigarrow s$. Obvious, because s in π.

Correctness proof

Notation. $q \rightsquigarrow r$ denotes "q is reachable from r"

Lemma. If $q \rightsquigarrow r$ and $f[q] < f[r]$, then some cycle contains q.

Proof: Let $\pi = q \to \cdots \to r$. Let s be the first node of π that is discovered (so $d[s] \leq d[q]$). We show $s \neq q$, $q \rightsquigarrow s$, and $s \rightsquigarrow q$.

- $s \neq q$. Otherwise at time $d[q]$ the path π is white and so $I(r) \subseteq I(q)$, which contradicts $f[q] < f[r]$.
- $q \rightsquigarrow s$. Obvious, because s in π.
- $s \rightsquigarrow q$. Since $d[s] < d[q]$ either $I(q) \subset I(s)$ or $I(s) \prec I(q)$. Since at time $d[s]$ the subpath of π from s to r is white, we have $I(r) \subseteq I(s)$. If $I(s) \prec I(q)$ then $f[q] > f[r]$. So $I(q) \subset I(s)$, and so s $\Rightarrow q$, which implies $s \rightsquigarrow q$.

Correctness proof

Theorem. Assume:

- q and r are accepting states such that $f[q] < f[r]$;
- the search from q has finished without an accepting lasso; and
- the search from r has just discovered a state s that was also discovered in the search from q.

Then r is not reachable from s (and so it is safe to backtrack from s).

Correctness proof

Theorem. Assume:

- q and r are accepting states such that $f[q] < f[r]$;
- the search from q has finished without an accepting lasso; and
- the search from r has just discovered a state s that was also discovered in the search from q.

Then r is not reachable from s (and so it is safe to backtrack from s).

Proof: Assume $s \rightsquigarrow r$. Since $q \rightsquigarrow s$ we have $q \rightsquigarrow r$. By the lemma some cycle contains q, contradicting that the search from q was unsuccessful.

Nesting the searches

- Two problems:
  - The algorithm always examines all states and transitions at least once.
  - If the algorithm must return a witness of non-emptiness, then it requires a lot of memory.

Nesting the searches

- Two problems:
  - The algorithm always examines all states and transitions at least once.
  - If the algorithm must return a witness of non-emptiness, then it requires a lot of memory.
- Solution: nest the searches.

Nesting the searches

- Two problems:
  - The algorithm always examines all states and transitions at least once.
  - If the algorithm must return a witness of non-emptiness, then it requires a lot of memory.
- Solution: nest the searches.
  - Perform a DFS from the initial state q_0.

Nesting the searches

- Two problems:
  - The algorithm always examines all states and transitions at least once.
  - If the algorithm must return a witness of non-emptiness, then it requires a lot of memory.
- Solution: nest the searches.
  - Perform a DFS from the initial state q_0.
  - Whenever the search blackens an accepting state q, launch a new (modified) DFS from q. If this DFS visits q again, report NONEMPTY. Otherwise, after termination continue with the first DFS.

Nesting the searches

- Two problems:
  - The algorithm always examines all states and transitions at least once.
  - If the algorithm must return a witness of non-emptiness, then it requires a lot of memory.
- Solution: nest the searches.
  - Perform a DFS from the initial state q_0.
  - Whenever the search blackens an accepting state q, launch a new (modified) DFS from q. If this DFS visits q again, report NONEMPTY. Otherwise, after termination continue with the first DFS.
  - If the first DFS terminates, report EMPTY.

NestedDFS(A)

Input: NBA $A = (Q, \Sigma, \delta, Q_0, F)$

Output: EMP if $L_\omega(A) = \emptyset$
NEMP otherwise

```
S ← ∅
dfs1(q0)
report EMP

proc dfs1(q)
   add [q, 1] to S
   for all r ∈ δ(q) do
      if [r, 1] ∉ S then dfs1(r)
   if q ∈ F then { seed ← q; dfs2(q) }
   return

proc dfs2(q)
   add [q, 2] to S
   for all r ∈ δ(q) do
      if r = seed then report NEMP
      if [r, 2] ∉ S then dfs2(r)
   return
```

NestedDFSwithWitness(A)

Input: NBA $A = (Q, \Sigma, \delta, Q_0, F)$

Output: EMP if $L_\omega(A) = \emptyset$
NEMP otherwise

```
S ← ∅; succ ← false
dfs1(q0)
report EMP

proc dfs1(q)
   add [q, 1] to S
   for all r ∈ δ(q) do
      if [r, 1] ∉ S then dfs1(r)
      if succ = true then return [q, 1]
   if q ∈ F then
      seed ← q; dfs2(q)
      if succ = true then return [q, 1]
   return

proc dfs2(q)
   add [q, 2] to S
   for all r ∈ δ(q) do
      if [r, 2] ∉ S then dfs2(r)
      if r = seed then
         report NEMP; succ ← true
      if succ = true then return [q, 2]
   return
```

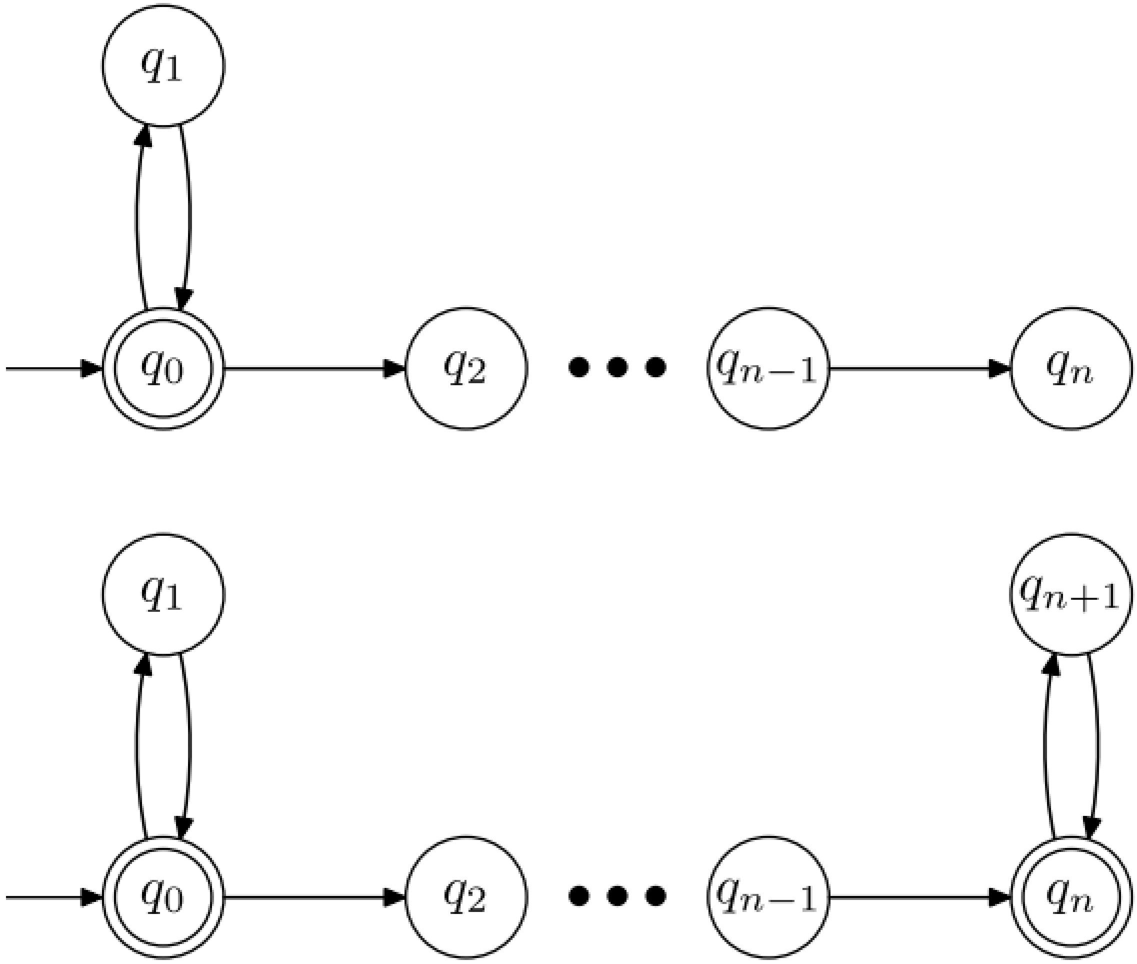
Evaluation

- Plus points:
  - Very low memory consumption: two extra bits per state.
  - Easy to understand and prove correct.

Evaluation

- Plus points:
  - Very low memory consumption: two extra bits per state.
  - Easy to understand and prove correct.
- Minus points:
  - Cannot be generalized to NGAs.
  - It may return unnecessarily long witnesses.
  - It is not optimal. An emptiness algorithm is optimal if it answers NONEMPTY immediately after the explored part of the NBA contains an accepting lasso.

Nested DFS is not optimal



Recall: Two approaches

1. Compute the set of accepting states, and for each accepting state, check if it belongs to a cycle.

   Nested depth first search algorithm

2. Compute the set of states that belong to some cycle, and for each of them, check if it is accepting.

   SCC-based algorithm

Second approach: a naïve algorithm

- Conduct a DFS, and for each discovered accepting state q start a new DFS from q to check if it belongs to a cycle.

Second approach: a naïve algorithm

- Conduct a DFS, and for each discovered accepting state q start a new DFS from q to check if it belongs to a cycle.
- Problem: too expensive.

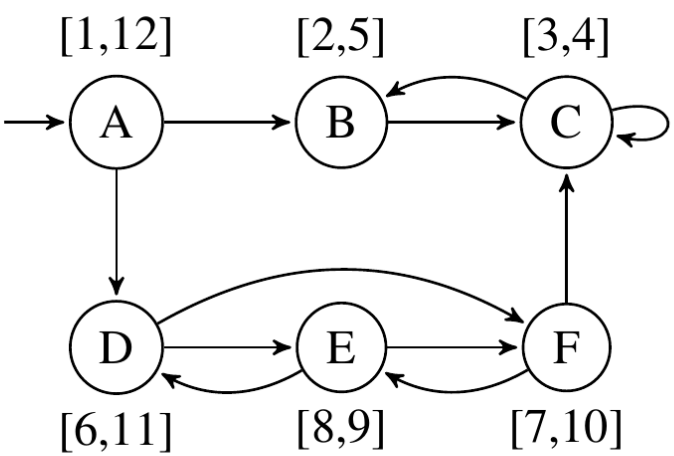
Second approach: a naïve algorithm

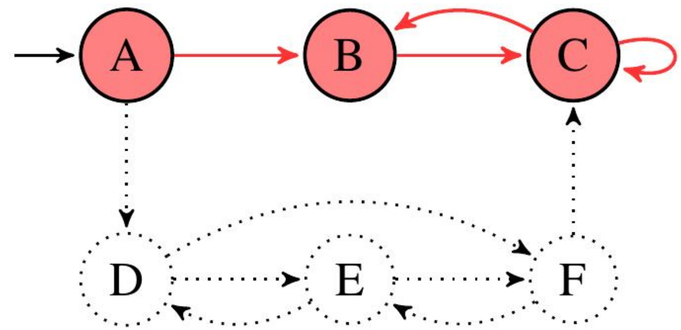
- Goal: conduct one single DFS which marks states in such a way that
  - every marked state belongs to a cycle, and
  - every state that belongs to a cycle is eventually marked.

The active graph

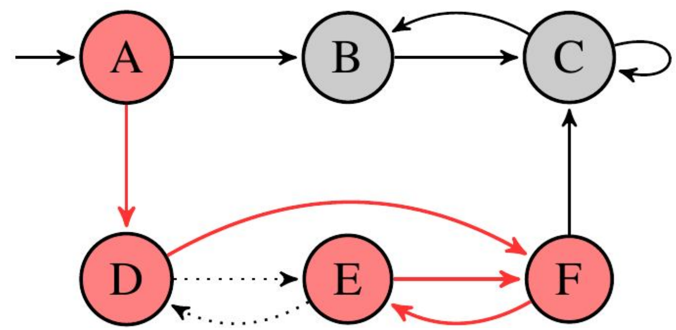
- Explored graph A_t at time t: subgraph of A containing the states and transitions explored by the DFS until time t.
- Strongly connected component (scc) of A_t: maximal set of states mutually reachable in A_t.
- A scc of A_t is active if some state appears in the grey path, and inactive otherwise. A state is active if its scc in A_t is active.
- Active graph at time t: subgraph of A_t containing the active states and the transitions between them.

The active graph

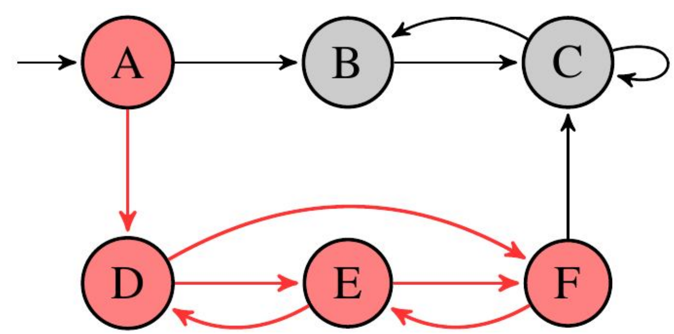




Time 5



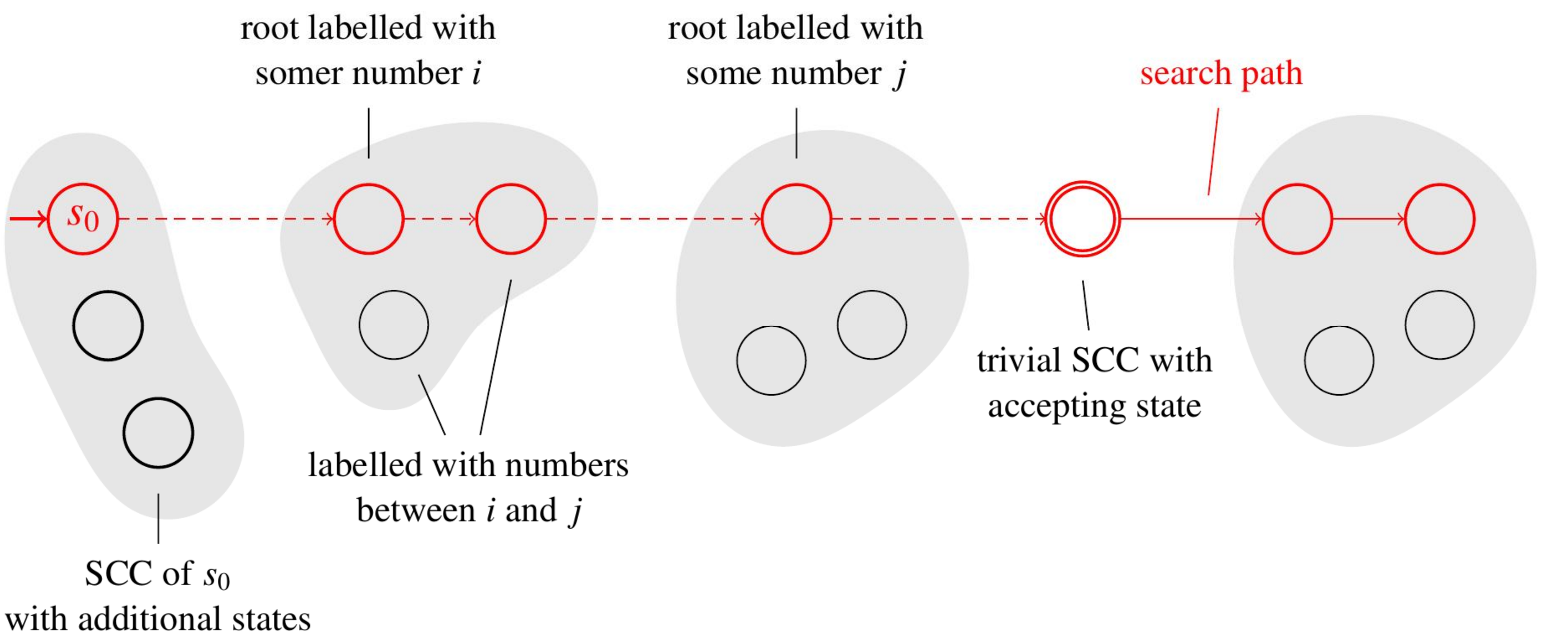
Time between 8 and 9



Time 11

Necklace structure of the active graph

- Def: The root of a scc of A_t is the first state of the scc visited by the DFS.
- The chain of the (open) necklace is the grey path. The beads are the active sccs.
- The chain contains all roots of the active sccs (and possibly other nodes).
- The scc of a root q contains all nodes s such that $d[q] \leq d[s] < d[r]$, where r is the next root.



Properties of the active graph

1) The root of a scc of A_t is the first state of the scc visited by the DFS.

2) The root of an scc of A_t is the last state of the scc from which the DFS backtracks.
   - Let r be the root of an scc. At time $d[r]$ there are white paths from r to all states of the scc.
   - By the White-path Theorem, all states of the scc are discovered before the DFS backtracks from r.
   - By the Parenthesis Theorem, the DFS backtracks from all states of the scc before it backtracks from r.

Properties of the active graph

3) An scc of A_t becomes inactive when the DFS backtracks from its root, i.e., when its root is blackened.
4) An inactive scc of A_t is also a scc of A.
   - When a scc of A_t becomes inactive, the DFS has already explored, and backtracked from, all states of A reachable from its root.
5) Roots of active sccs of A_t occur in the grey path.
   - If a scc is active then its root has already been discovered, and by (3) it is not yet black. So it is grey.

Properties of the active graph

6) Let q be an active state of A_t, and let r be the root of its scc. No state discovered between q and r, i.e., no state s satisfying $d[r] < d[s] < d[q]$, is an active root of A_t.

- Assume s is active root and $d[r] < d[s] < d[q]$
- Claim: r and s are in the same scc, contradicting that r is root.
  - $r \rightsquigarrow s$. By (5), r and s are in the grey path. Further, r precedes s because $d[r] < d[s]$.
  - $s \rightsquigarrow q$. Because, since s is active and $d[s] < d[q]$, state q is discovered during the execution of dfs(s).
  - $q \rightsquigarrow r$. Because q and r belong to the same scc.

Properties of the active graph

7) If q and r are active and $d[q] < d[r]$ then $q \rightsquigarrow r$.

Let q' and r' be the roots of the sccs of q and r.

Since $q \rightsquigarrow q'$ and $r' \rightsquigarrow r$ it suffices to prove $q' \rightsquigarrow r'$.

Since q' and r' are roots , they belong to the grey path by (5). So at least one of $q' \rightsquigarrow r'$ and $r' \rightsquigarrow q'$ holds.

We have $d[q'] < d[q]$ by the definition of root and $d[q] < d[r]$ by assumption.
So $d[q'] < d[q] < d[r]$.

By (6), neither $d[r'] < d[q'] < d[r]$ nor $d[q'] < d[r'] < d[q]$ hold. Further, $d[r'] < d[r]$ by the definition of root.
So $d[q'] < d[q] < d[r'] < d[r]$.

But then q' entered the grey path before r', and so $q' \rightsquigarrow r'$.

SCC-based algorithm

- The algorithm maintains the explored graph and the necklace structure of the active graph while the DFS is conducted.
- Data structures:
  - Set S of states visited by the DFS so far.
  - Mapping $rank: S \to \mathbb{N}$ assigning to each state a consecutive number in the order they are discovered.
  - Mapping $act: S \to \{\text{true}, \text{false}\}$ indicating which states are currently active.
  - Necklace stack $neck$, containing beads of the form (r, C), where C is the set of states of an active scc, and r its root. The oldest bead (i.e., the one with the oldest root) is at the bottom of the stack, and the newest at the top.

SCC-based algorithm

- After the initialization step, the DFS is always either
  - exploring a new edge (which may lead to a new state or to a state already visited), or
  - backtracking along an edge explored earlier.
- We show how to update S, $rank$, act, and $neck$ after an initialization, exploration, or backtracking step.
- Further, we show how to check after each step whether the explored graph contains an accepting lasso.

Initialization

Initially the explored and active graphs only contain the initial state q_0 and no edges. So:

- $S \coloneqq \{q_0\}$
- $rank(q_0) := 1$
- $act(q_0) := \text{true}$
- $neck := (q_0, \{q_0\})$

Exploration

Assume the DFS has just explored a transition $q \rightarrow r$.
We show how to update the data structures.
We consider five cases:

i. r is a new state.

ii. r has been visited by the DFS before, and is inactive.

iii. r has been visited by the DFS before, is active, and was discovered strictly after q.

iv. r has been visited by the DFS before, is active, and $r = q$.

v. r has been visited by the DFS before, is active, and was discovered strictly before q.

Exploration: Case i

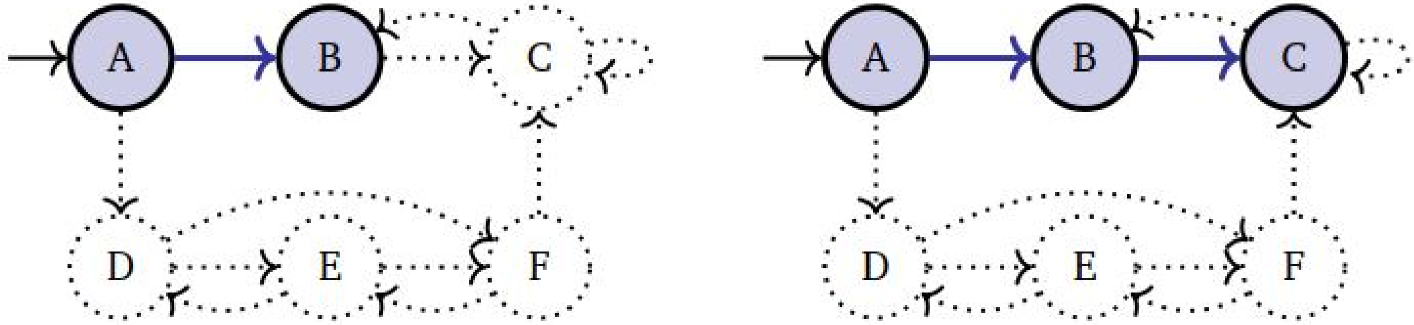
The DFS has just explored a transition $q \to r$.

Case i: r is a new state.

Then the explored graph is extended with r, which is active.

The updates are: $S \coloneqq S \cup \{r\}, rank(r) \coloneqq |S|, act(r) := \text{true}$, and $push(r, \{r\})$ to $neck$.

After that recursively call dfs(r)



Exploring B→C: before and after

Exploration: Case ii

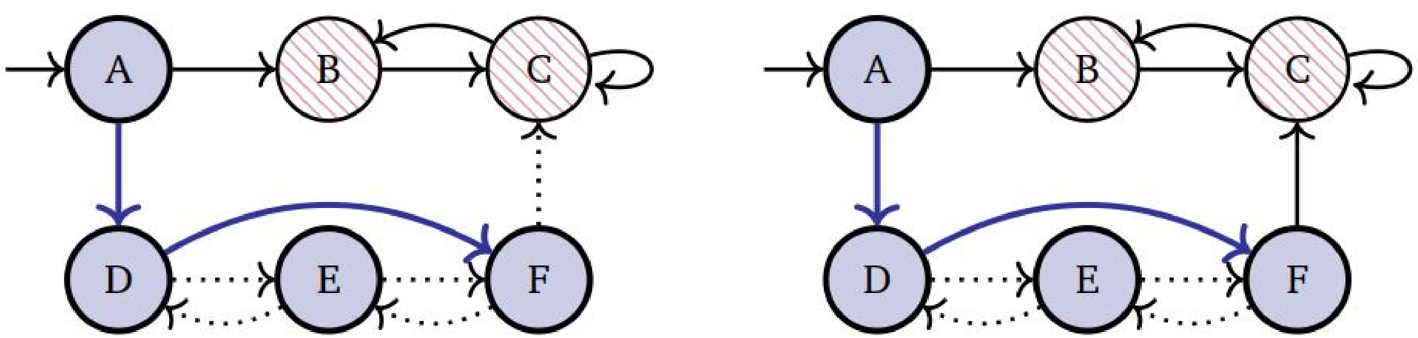
The DFS has just explored a transition $q \to r$.

Case ii: r has been visited by the DFS before, and is inactive.

Since r is inactive, its scc has already been completely explored by the DFS (see properties (2) and (3)).

So q and r belong to different sccs and $q \to r$ cannot create an accepting lasso.

So no update is needed, and no recursive DFS call is started.



Exploring F→C: before and after

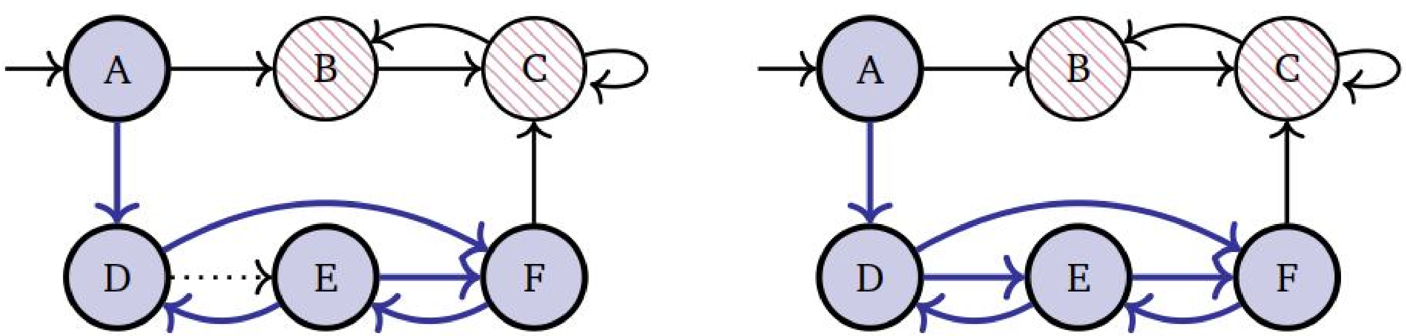
Exploration: Case iii

The DFS has just explored a transition $q \rightarrow r$.

Case iii: r has been visited by the DFS before, is active, and was discovered strictly after q.

In this case both q and r are active, and already belong to the necklace.

Since $rank(r) > rank(q)$, either q and r belong to the same scc, or the scc of q is before the scc of r in the necklace. No accepting lasso can be created. There is nothing to do, and no recursive DFS call is started.



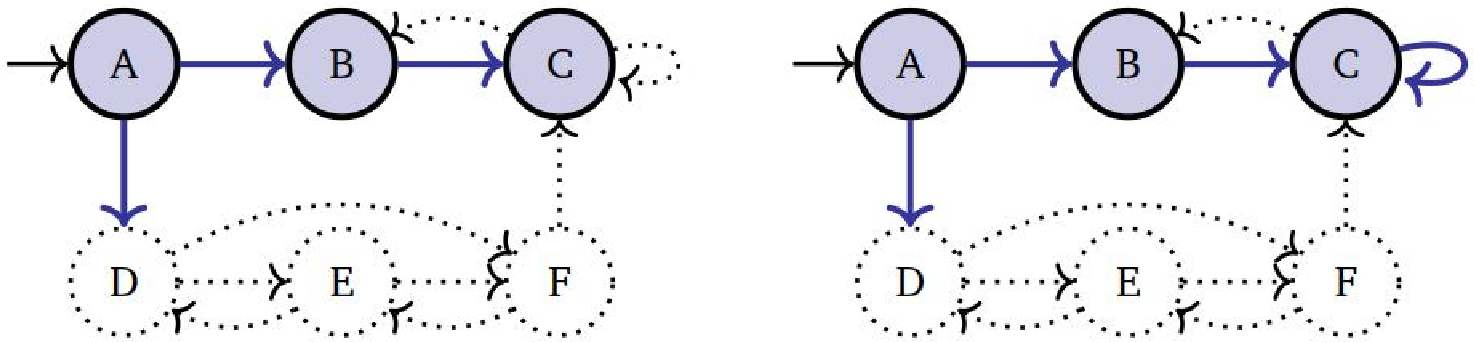
Exploring D→E: before and after

Exploration: Case iv

The DFS has just explored a transition $q \to r$.

Case iv: r has been visited by the DFS before, is active, and $r = q$.

Then $q \to r$ is a self-loop. If q is accepting state, then an accepting lasso has been discovered, and the algorithm reports it. Otherwise, there is nothing to do.



Exploring C→C: before and after

Exploration: Case v

The DFS has just explored a transition $q \to r$.

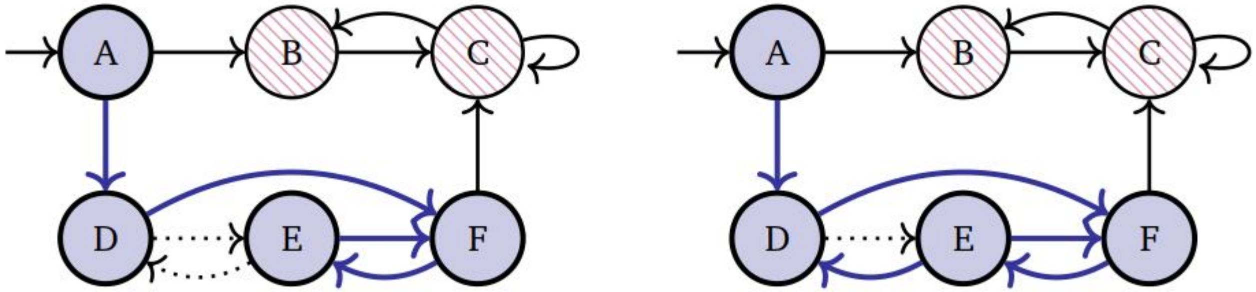
Case v: r has been visited by the DFS before, is active, and was discovered strictly before q.

By property (7) we have $r \rightsquigarrow q$. So q and r belong to the same scc.

All sccs of the necklace between the sccs of r and q must be merged.

For this, pop beads (s, C) from neck, merging the C's, and stopping when the popped bead satisfies $rank(s) \leq rank(r)$.

Then push a new bead (s, D), where D is the result of the merge.



Exploring E→D: before and after

Backtracking: Case vi

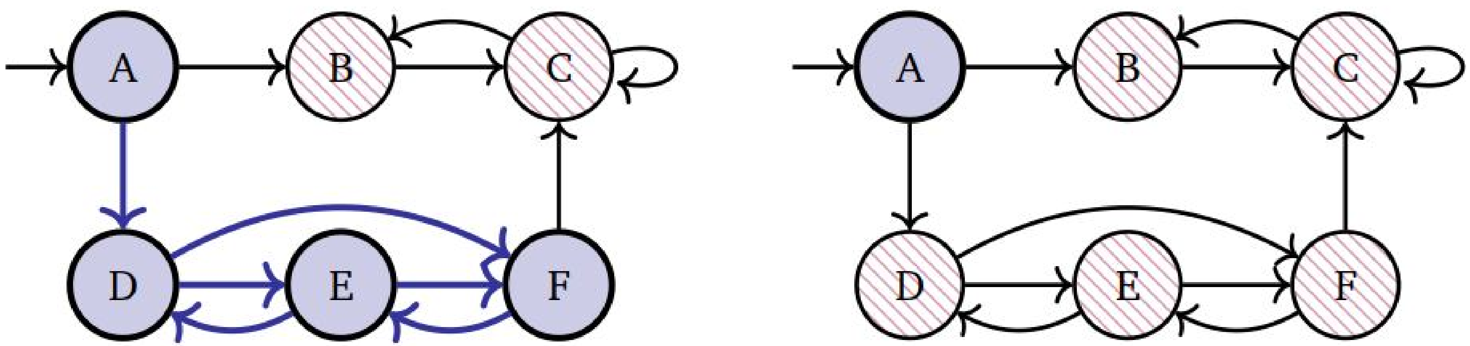
The DFS has already explored all edges leaving q, and now backtracks from q.

Case vi: q is a root of the active graph.

Then, before backtracking from q, the top bead of $neck$ is (q, C) for some set C

After backtracking, q and its entire scc become inactive by property (3), and they do not belong to the active graph anymore.

So we pop (q, C) from $neck$ and set $act(r)$ to false for every $r \in$ C



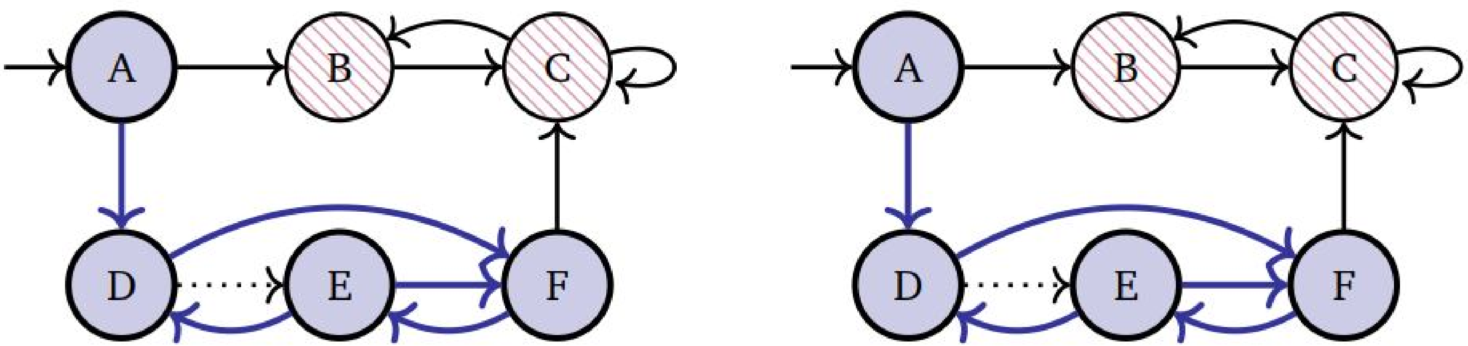
Backtracking from D

Backtracking: Case vii

The DFS has already explored all edges leaving q, and now backtracks from q.

Case vii: q is not a root of the active graph.

Then, by properties (2) and (3) the root of the scc of q is active and remains so after backtracking. The active graph does not change, and there is nothing to do.



Backtracking from E

Pseudocode

SCCsearch(*A*)
Input: NBA $A = (Q, \Sigma, \delta, Q_0, F)$
Output: EMP if $L_\omega(A) = \emptyset$, NEMP otherwise

```
1   S, N ← ∅; t ← 0
2   dfs(q0)
3   report EMP

4   proc dfs(q)
5       n ← n + 1; rank(q) ← n
6       add q to S; act(q) ← 1; push(q, {q}) onto N
7       for all r ∈ δ(q) do
8           if r ∉ S then dfs(r)
9           else if act(r) then
10              D ← ∅
11              repeat
12                  pop (s, C) from N; if s ∈ F then report NEMP
13                  D ← D ∪ C
14              until rank(s) ≤ rank(r)
15              push(s, D) onto N
16      if q is the top root in N then
17          pop (q, C) from N
18          for all r ∈ C do act(r) ← false
```

- Initialization and Case (i): Line 5
- Case (ii): conditions at 7,8 do not hold and nothing happens
- Cases (iii)-(v): repeat-until loop

Pseudocode: runtime

```
SCCsearch(A)
Input: NBA A = (Q, Σ, δ, Q0, F)
Output: EMP if Lω(A) = ∅, NEMP otherwise
1  S, N ← ∅; t ← 0
2  dfs(q0)
3  report EMP

4  proc dfs(q)
5     n ← n + 1; rank(q) ← n
6     add q to S; act(q) ← 1; push(q, {q}) onto N
7     for all r ∈ δ(q) do
8        if r ∉ S then dfs(r)
9        else if act(r) then
10          D ← ∅
11          repeat
12             pop (s, C) from N; if s ∈ F then report NEMP
13             D ← D ∪ C
14          until rank(s) ≤ rank(r)
15          push(s, D) onto N
16    if q is the top root in N then
17       pop (q, C) from N
18       for all r ∈ C do act(r) ← false
```

- 2m steps of type (i)-(vii)
- Each step of type (i)-(iv) or (vii) takes constant time
- Step of type (v):
  - At most *n* primary beads enter the necklace
  - Secondary beads are merges of primary beads, at most n enter the necklace.
  - So line 13 is executed $O(n)$ times
  - Implementing sets as linked lists with pointers to first and last elements: $O(n)$ time
- Step of type (vi): each state is deactivated exactly once at line 18, so $O(n)$ time.

Extension to NGAs

- A NGA A with accepting condition $\{F_0, \dots, F_{k-1}\}$ is nonempty iff some scc S satisfies $S \cap F_i \neq \emptyset$ for every $i \in [k]$
- Label each state q with the index set I_q of the acceptance sets it belongs to.
- Extend beads with a third component: (q, C, I), where I is an index set.

line	*SCCsearch* for NBA	*SCCsearch* for NGA
6	**push**$(q, \{q\})$	**push**$(q, \{q\}, I_q)$
10	$D \leftarrow \emptyset$	$D \leftarrow \emptyset$; $J \leftarrow \emptyset$
12	**pop**(s, C); **if** $s \in F$ **then report** NEMP	**pop**(s, C, I)
13	$D \leftarrow D \cup C$	$D \leftarrow D \cup C$; $J \leftarrow J \cup I$;
15	**push**(s, D)	**push**(s, D, J); **if** $J = K$ **then report** NEMP
17	**pop**(q, C)	**pop**(q, C, I)