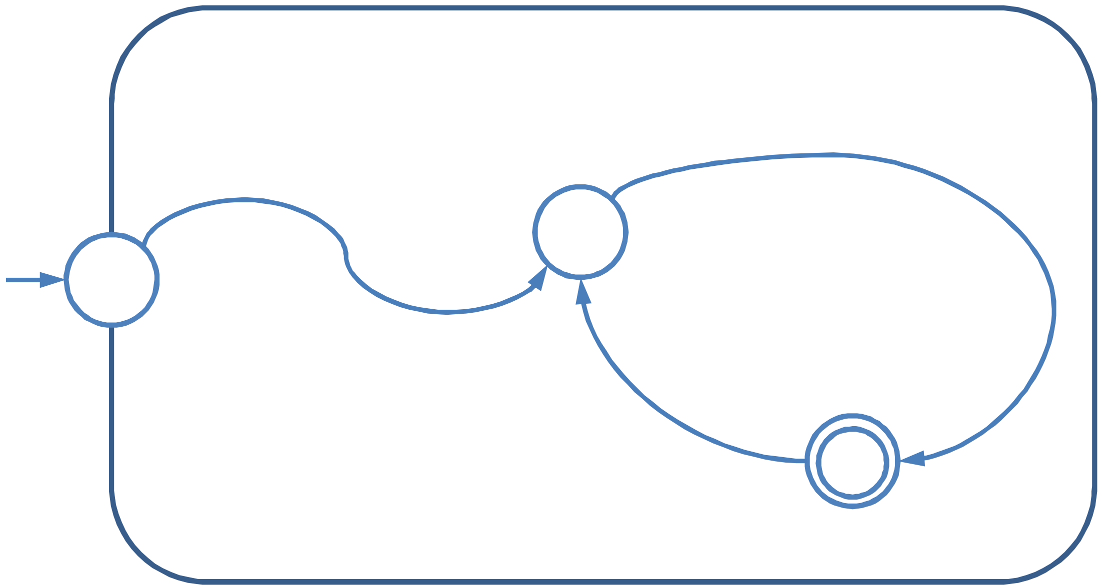
Checking emptiness of Büchi automata

Accepting lassos

- A NBA is nonempty iff it has an accepting lasso



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

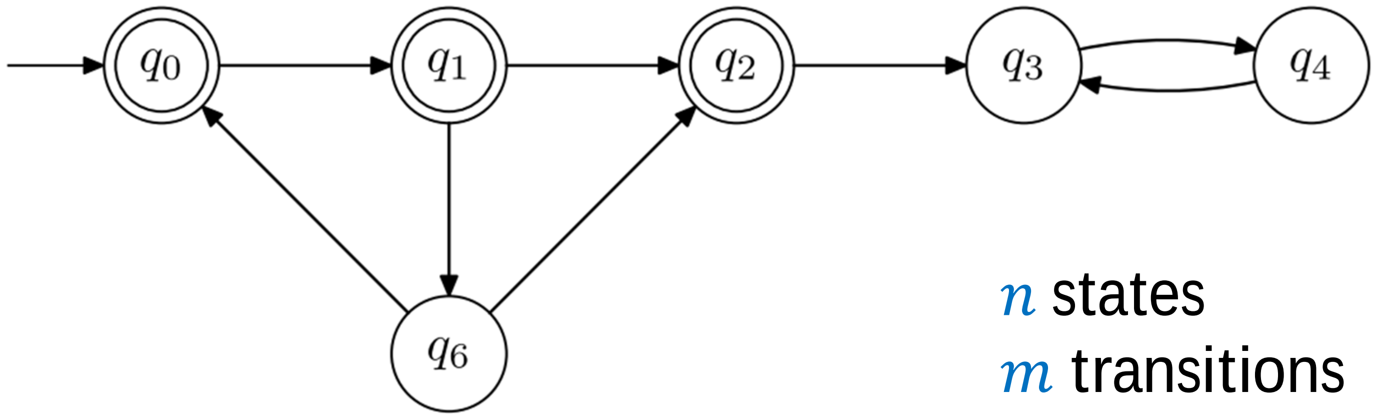
2. Compute the set of states that belong to some cycle, and for each of them, check if it is accepting.

   Two-stack algorithm

First approach: A naïve algorithm

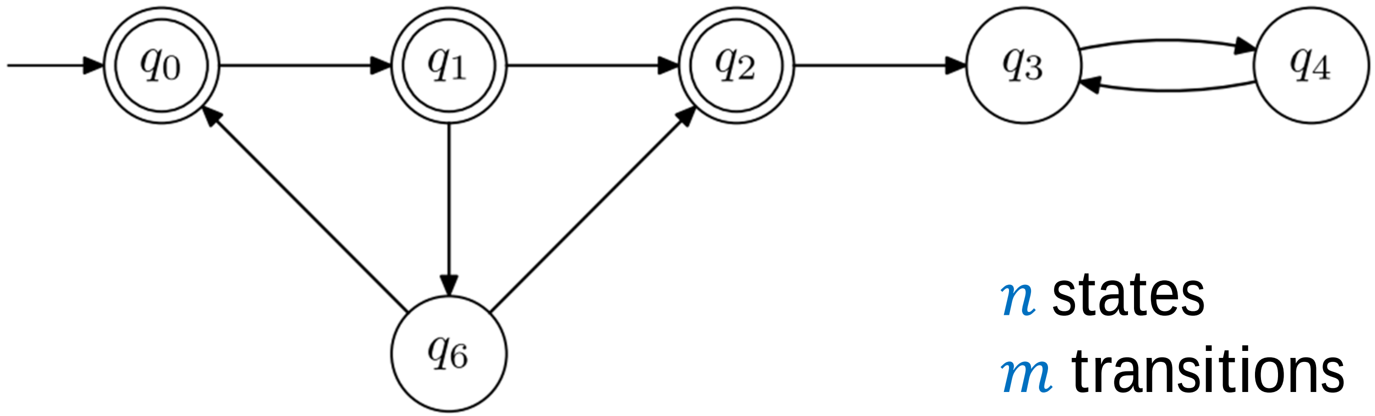
1. Compute the set of accepting states by means of a graph search (DFS, BFS, ...).
2. For each accepting state q, conduct a second search (DFS, BFS,...) starting at q to decide if q belongs to a cycle.

First approach: A naïve algorithm



n states
m transitions

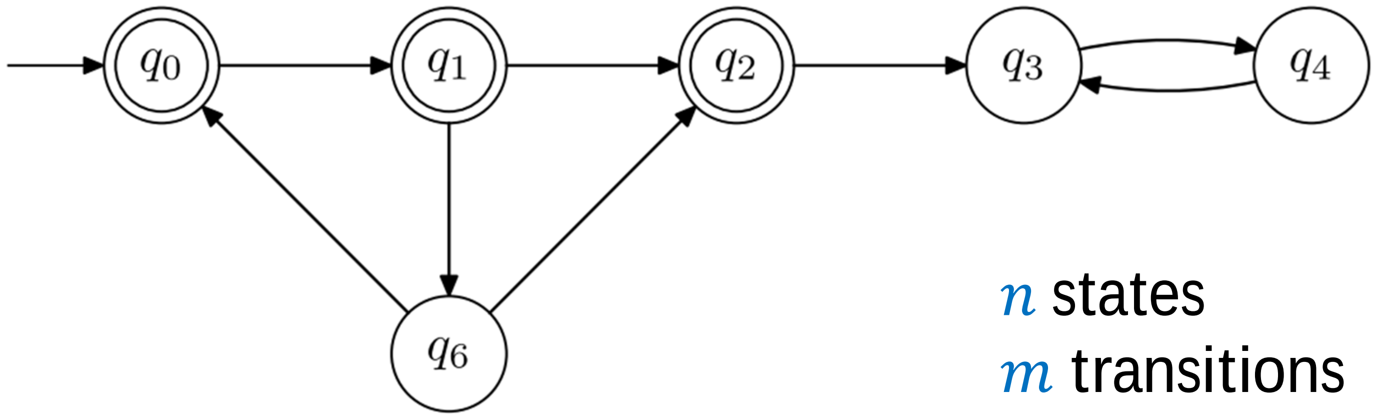
First approach: A naïve algorithm



n states
m transitions

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

First approach: A naïve algorithm

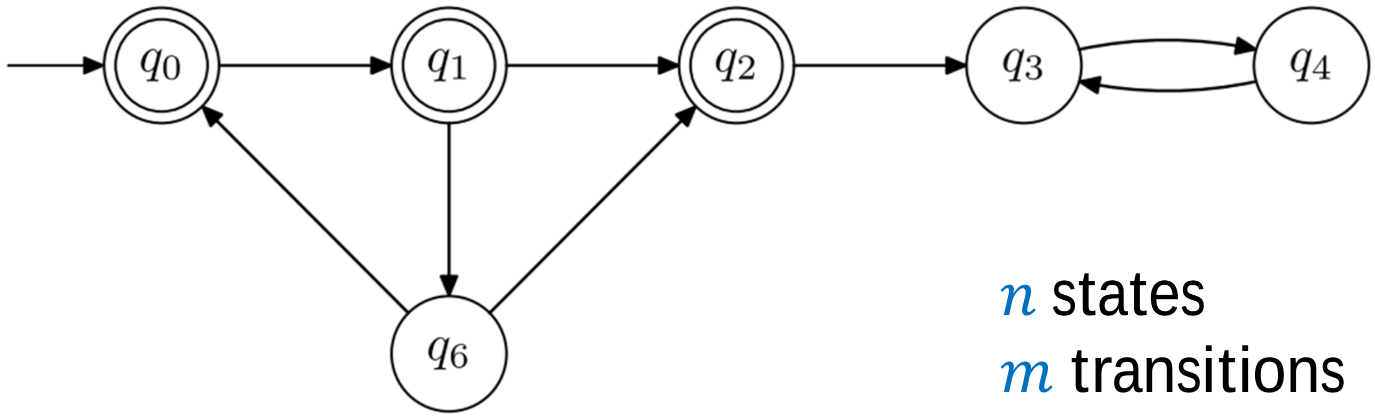


n states
m transitions

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

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

First approach: A naïve algorithm



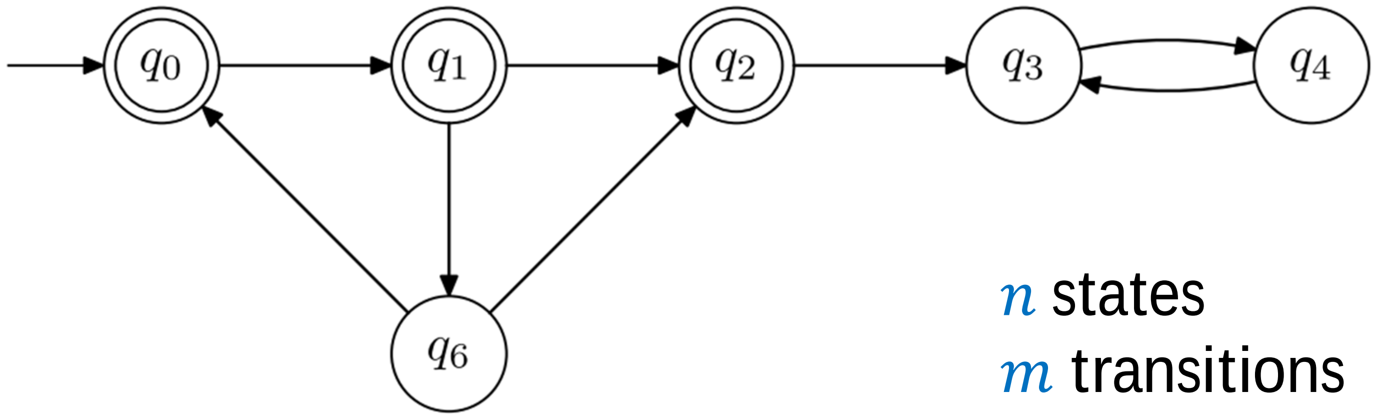
n states
m transitions

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

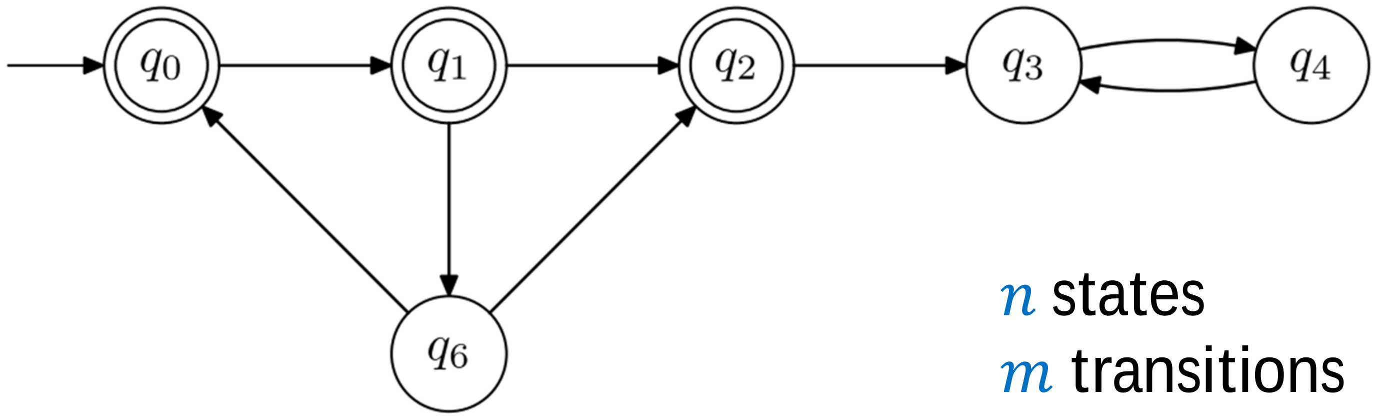
Runtime of the first search: $O(m)$

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.

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.
- Coloring scheme: at time t state q is either
  - white: not yet discovered, $1 \leq t \leq d[q]$

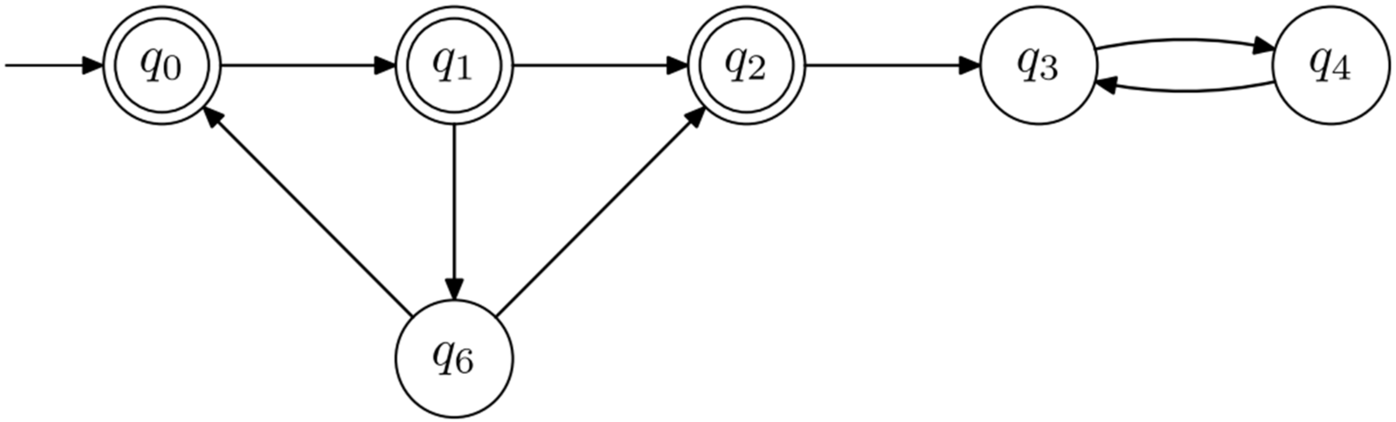
Depth-first search: Terminology

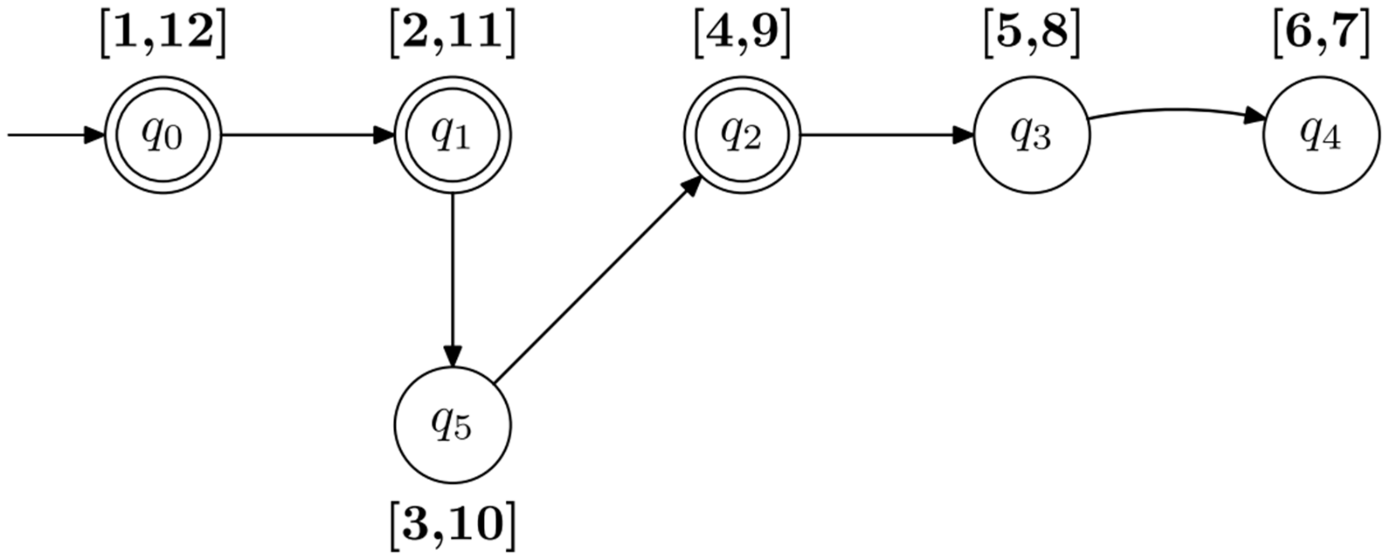
- States are discovered by the search.
- After recursively exploring all successors, the search backtracks from the state.
- The search assigns to a state q:
  - a discovery time $d[q]$;
  - a finishing time $f[q]$;
  - a DFS-predecessor, the state from which q is discovered (DFS-tree).
- Coloring scheme: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

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, $\mathrm{d}[q] < t \leq f[q]$
  - 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 $(d[q], f[q]]$.

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).

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

- $I(q)$ 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

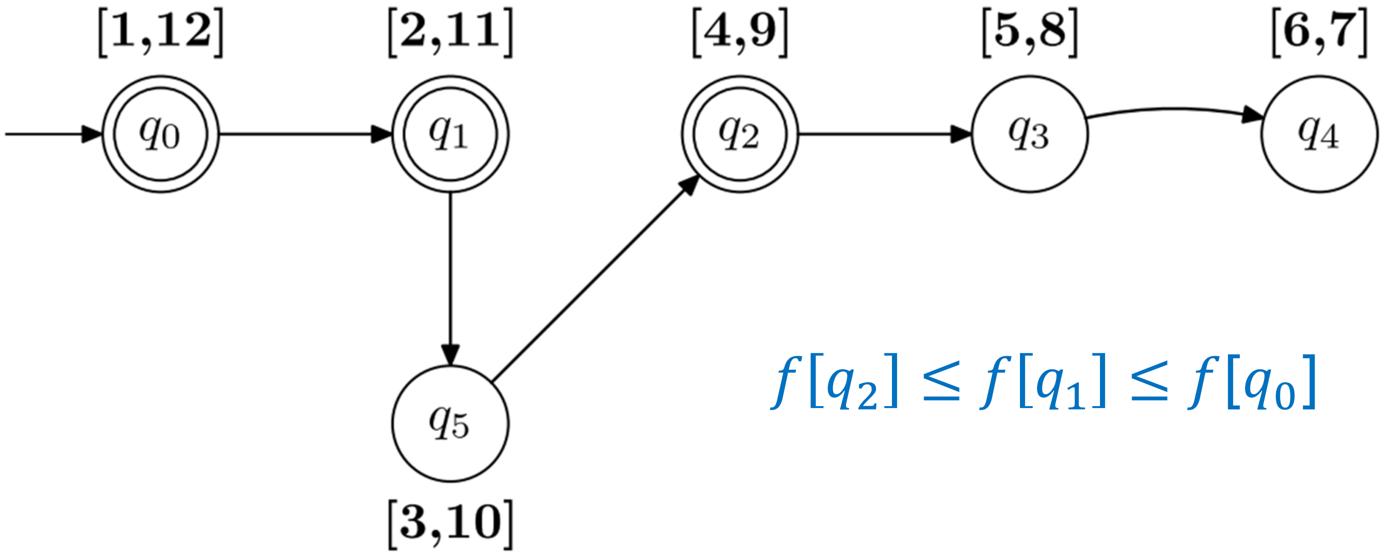
- White-path theorem. $q \Rightarrow r$ (and so $I(r) \subseteq I(q)$) iff at time $d[q]$ state r can be reached from q along a path of white states.
- Grey-path theorem. At every moment in time, all grey nodes form a simple path of the DFS tree (the grey path).

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.

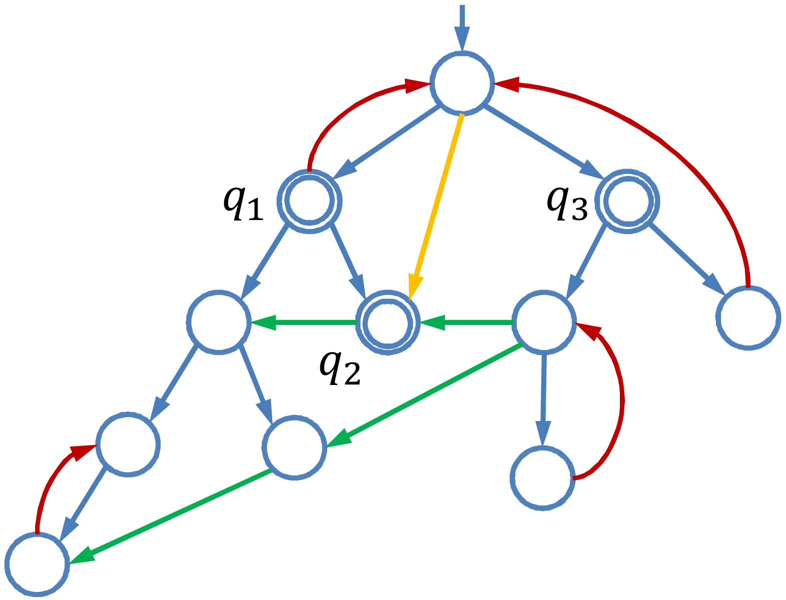
Nested-DFS algorithm

- 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?

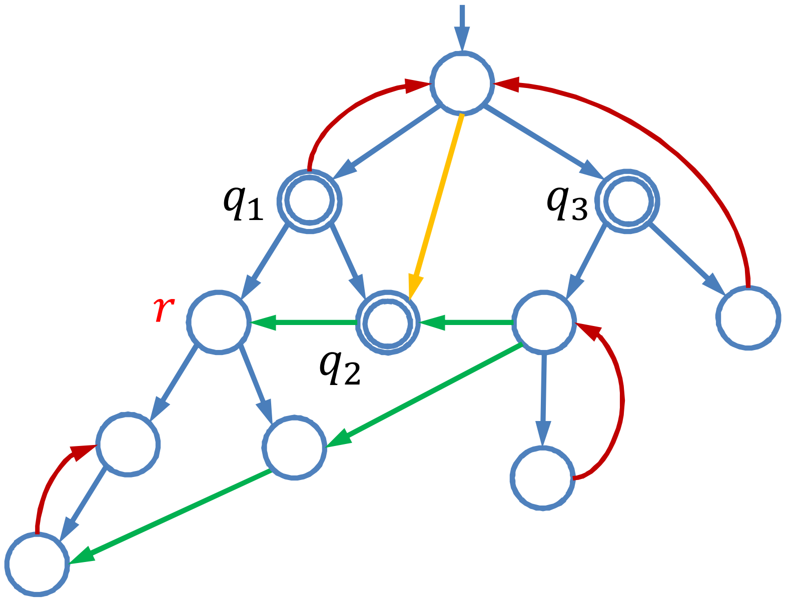


- Edges processed counterclockwise

DFS-tree
backedges
forward edges
crossedges

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

Why does it work?



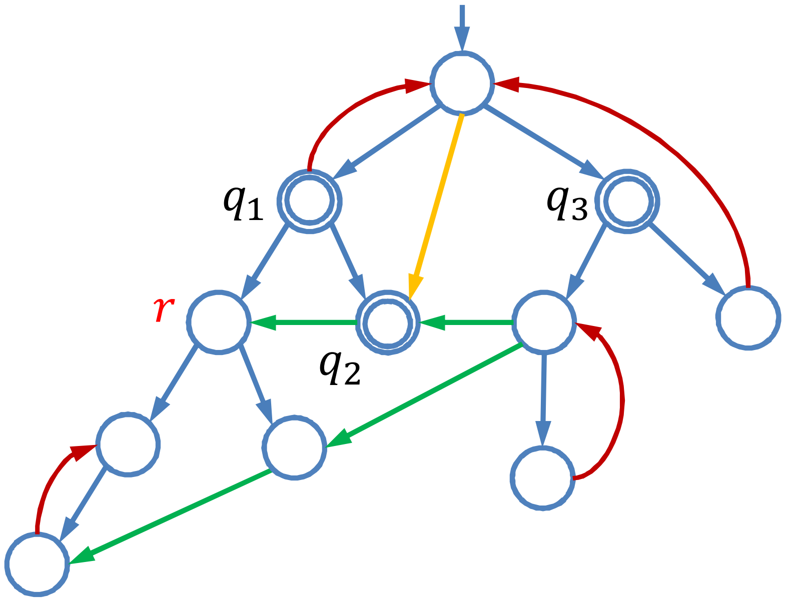
- Edges processed counterclockwise

DFS-tree
backedges
forward edges
crossedges

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

- State r discovered during the search from q_2

Why does it work?



- Edges processed counterclockwise

DFS-tree
backedges
forward edges
crossedges

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

- State r discovered during the search from q_2
- To prove: during the search from q_1, it is safe to backtrack from r, because we do not "miss any accepting lassos"
- Amounts to: proving that q_1 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 $\pi = q \to \cdots \to r$. 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 $s \rightsquigarrow q$.

Correctness proof

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

Lemma. If $q \leadsto r$ 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 $s \neq q$, $q \leadsto s$, and $s \leadsto q$.

- $s \neq q$. 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 $\mathrm{s} \Rightarrow q$, 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
```

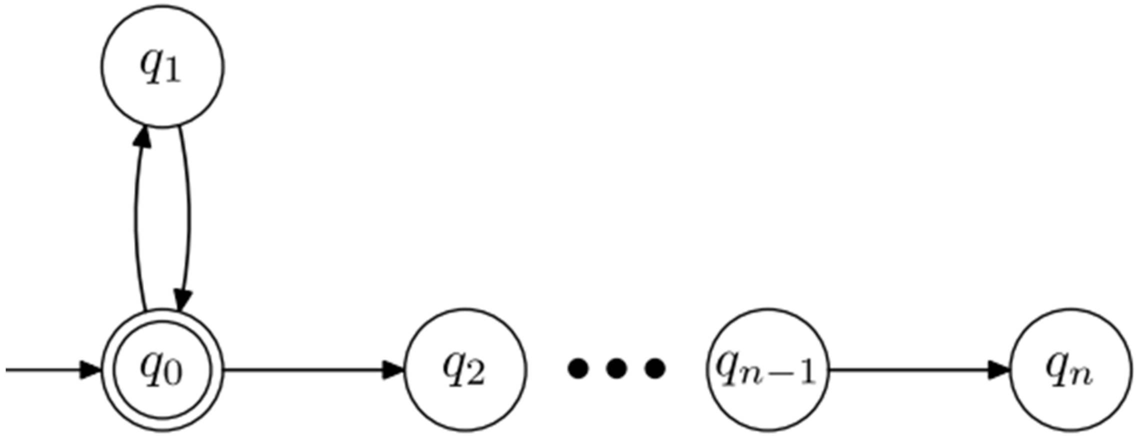
Evaluation

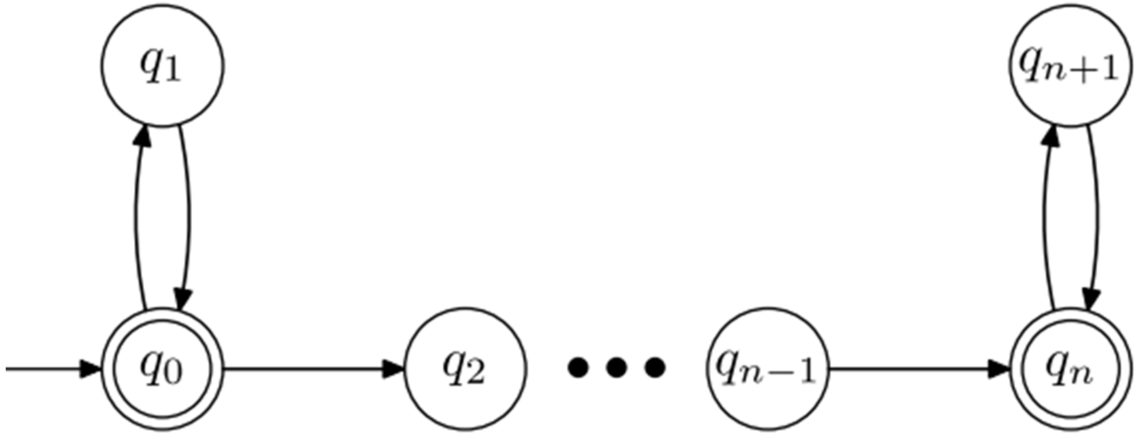
- Plus points:
  - Very low memory consumption: two extra bits per state.
  - Easy to understand and prove correct.

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.

   Two-stack 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

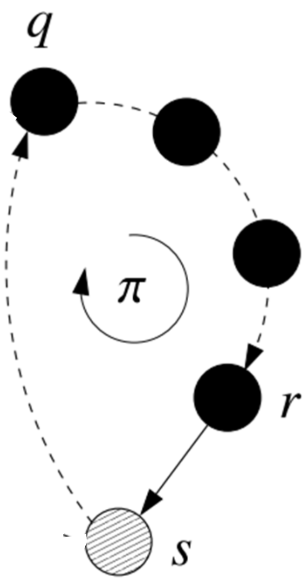
- Conduct a DFS, and for each discovered accepting state q start a new DFS from q to check if it belongs to a cycle.
- Problem: too expensive.
- Goal: conduct one single DFS which marks states in such a way that
  - every marked state belongs to a cycle, and
  - every state that belongs to a cycle is eventually marked.

There is hope ...

Lemma. At time $f[q]$, state q belongs to a cycle of the NBA iff it belongs to a cycle of the discovered part of the NBA.

There is hope ...

Lemma. At time $f[q]$, state q belongs to a cycle of the NBA iff it belongs to a cycle of the discovered part of the NBA.



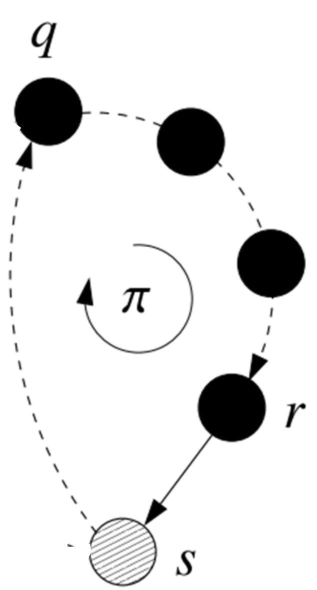
Proof.

π: cycle containing q.

r: last black state after q at $f[q]$.

There is hope ...

Lemma. At time $f[q]$, state q belongs to a cycle of the NBA iff it belongs to a cycle of the discovered part of the NBA.



Proof.

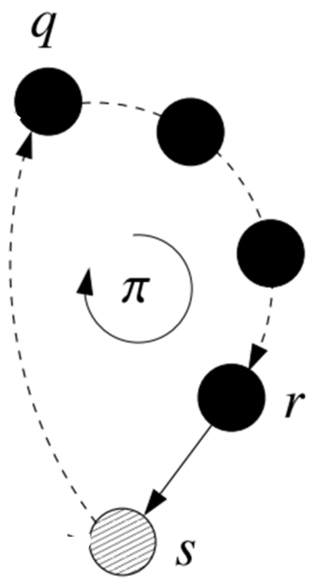
π: cycle containing q.

r: last black state after q at $f[q]$.

Case $r = q$. Then π has been discovered.

There is hope ...

Lemma. At time $f[q]$, state q belongs to a cycle of the NBA iff it belongs to a cycle of the discovered part of the NBA.



Proof.

π: cycle containing q.

r: last black state after q at $f[q]$.

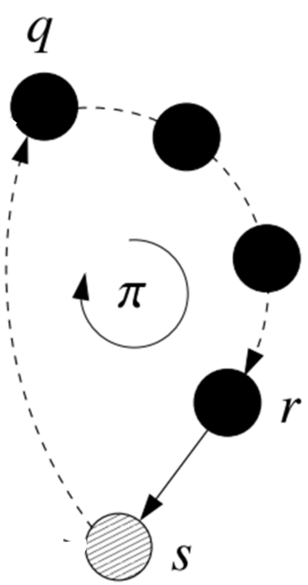
Case $r = q$. Then π has been discovered.

Case $r \neq q$.

s: successor of r in π.

There is hope ...

Lemma. At time $f[q]$, state q belongs to a cycle of the NBA iff it belongs to a cycle of the discovered part of the NBA.



Proof.

π: cycle containing q.

r: last black state after q at $f[q]$.

Case $r = q$. Then π has been discovered.

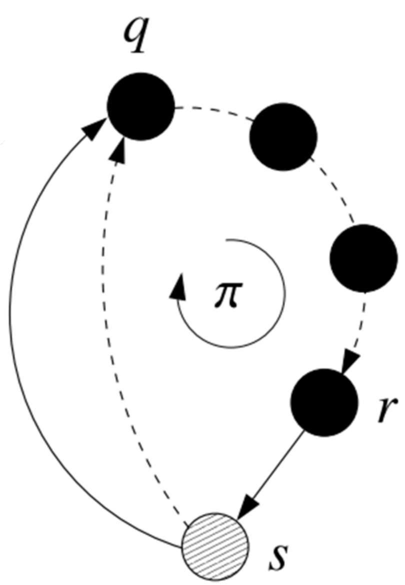
Case $r \neq q$.

s: successor of r in π.

We have $d[s] < f[r] < f[q] < f[s]$.

There is hope ...

Lemma. At time $f[q]$, state q belongs to a cycle of the NBA iff it belongs to a cycle of the discovered part of the NBA.



Proof.

π: cycle containing q.

r: last black state after q at $f[q]$.

Case $r = q$. Then π has been discovered.

Case $r \neq q$.

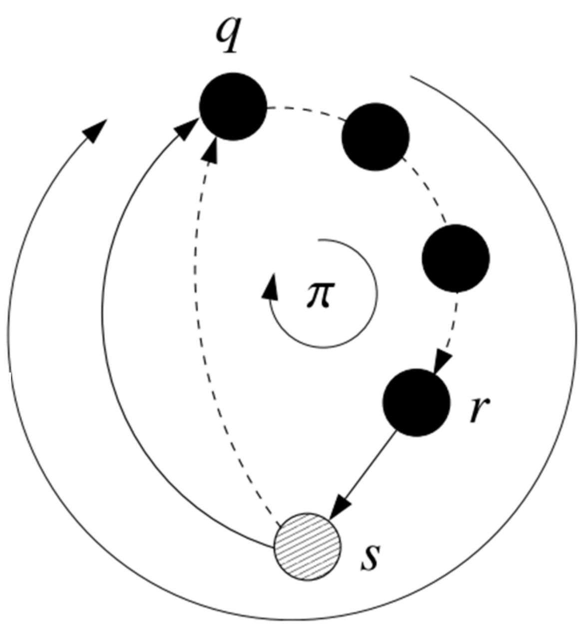
s: successor of r in π.

We have $d[s] < f[r] < f[q] < f[s]$.

So $s \Rightarrow q$, and every transition of $s \Rightarrow q$ has been discovered at time $f[q]$.

There is hope ...

Lemma. At time $f[q]$, state q belongs to a cycle of the NBA iff it belongs to a cycle of the discovered part of the NBA.



Proof.

π: cycle containing q.

r: last black state after q at $f[q]$.

Case $r = q$. Then π has been discovered.

Case $r \neq q$.

s: successor of r in π.

We have $d[s] < f[r] < f[q] < f[s]$.

So $s \Rightarrow q$, and every transition of $s \Rightarrow q$ has been discovered at time $f[q]$.

So cycle $q \xrightarrow{\pi} r \rightarrow s \Rightarrow q$ has been discovered at time $f[q]$.

First ideas

- Maintain a set C of candidates: states for which the search cannot yet decide if they belong to a cycle or not.
  - States are added to the set when they are greyed.
  - States are removed from the set when blackened, or before.
  - States are removed before they are blackened iff they belong to a cycle.

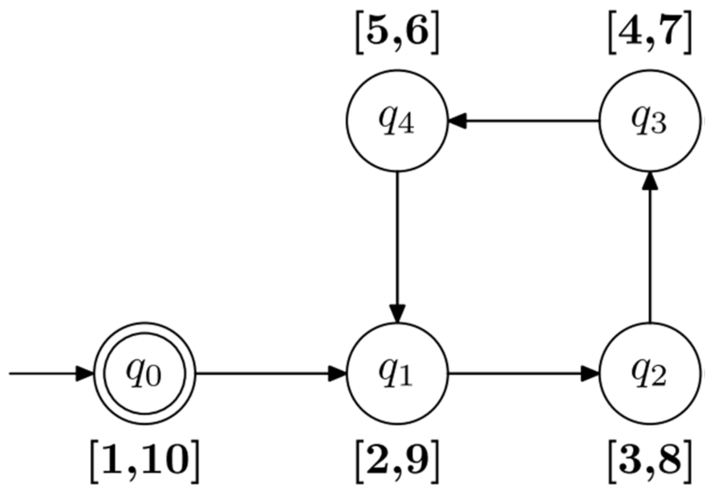
First ideas

- Maintain a set C of candidates: states for which the search cannot yet decide if they belong to a cycle or not.
  - States are added to the set when they are greyed.
  - States are removed from the set when blackened, or before.
  - States are removed before they are blackened iff they belong to a cycle.
- Updating C when the DFS explores a transition (q, r).
  - If r is a new state, add r to C.
  - If r has already been discovered, but q is not reachable from r, do nothing.
  - If r has already been discovered and $r \rightsquigarrow q$ then new cycles are created. Which states must be removed from C?

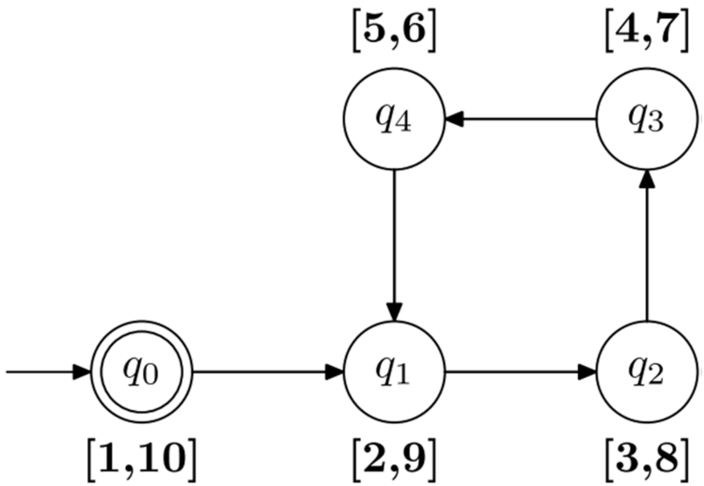
First ideas

- Maintain a set C of candidates: states for which the search cannot yet decide if they belong to a cycle or not.
  - States are added to the set when they are greyed.
  - States are removed from the set when blackened, or before.
  - States are removed before they are blackened iff they belong to a cycle.
- Updating C when the DFS explores a transition (q, r).
  - If r is a new state, add r to C.
  - If r has already been discovered, but q is not reachable from r, do nothing.
  - If r has already been discovered and $r \leadsto q$ then new cycles are created. Which states must be removed from C?
- For the moment we assume that an oracle determines if $r \leadsto q$ holds.

Updating C: first attempt

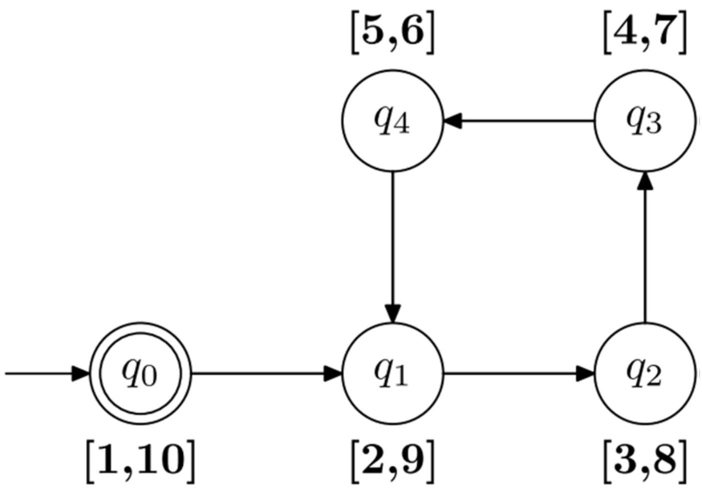


Updating C: first attempt



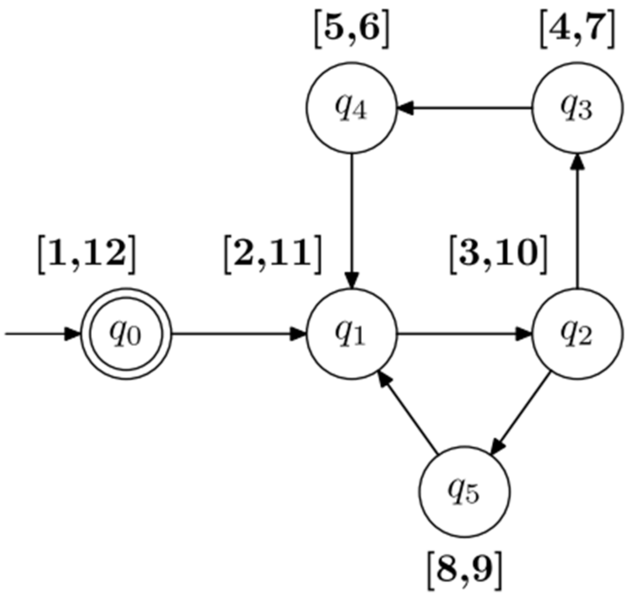
- After exploring (q_4, q_1) we have to remove $q_1, \ldots, q_4$ from C.
- Suggests implementing C as stack.

Updating C: first attempt



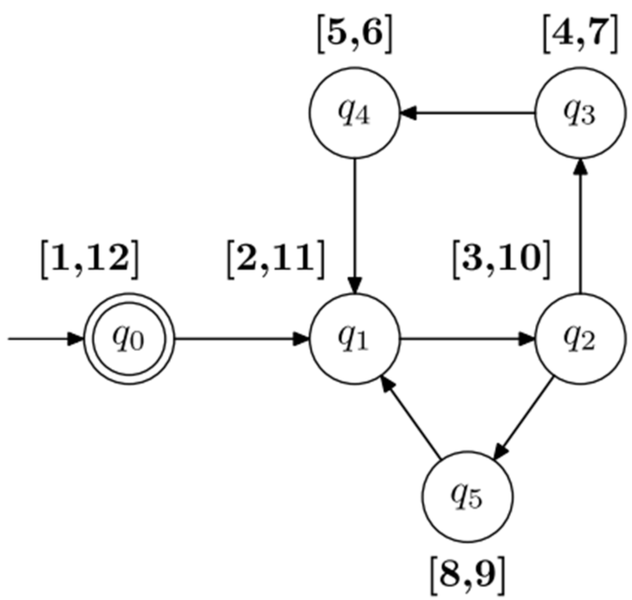
- After exploring (q_4, q_1) we have to remove $q_1, \ldots, q_4$ from C.
- Suggests implementing C as stack.
- First attempt:
  - push a state when it is discovered.
  - when exploring (q, r), if r has already been discovered and $r \rightsquigarrow q$, then pop until r is popped.

Problem and second attempt



After exploring (q_4, q_1) states $q_4, \dots, q_1$ are popped.
After exploring (q_5, q_1), since q_1 is not in the stack, q_0 is wrongly popped.

Problem and second attempt



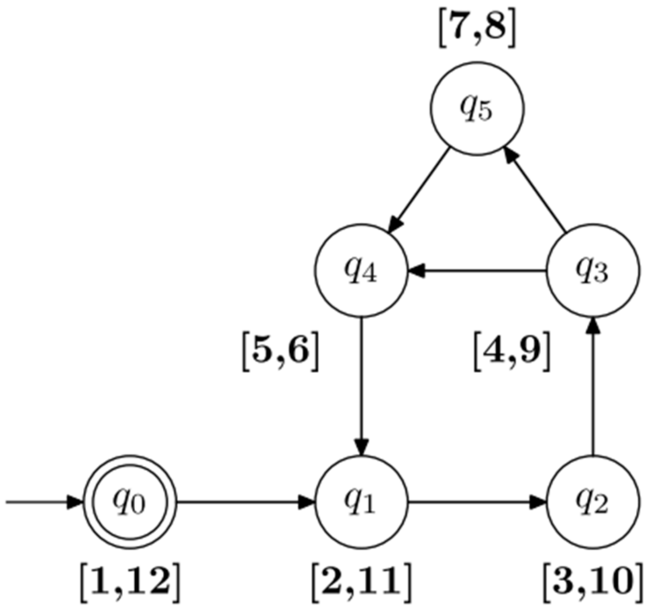
Second attempt:

– push a state when it is discovered.

– when exploring (q, r), if r has already been discovered and $r \rightsquigarrow q$, then pop until r is popped and then push r back.

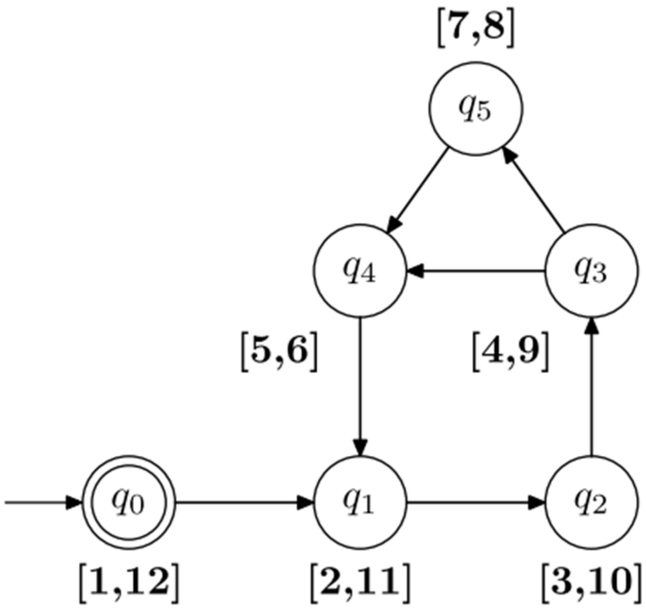
After exploring (q_4, q_1) states $q_4, \ldots, q_1$ are popped.
After exploring (q_5, q_1), since q_1 is not in the stack, q_0 is wrongly popped.

Problem and final attempt



After exploring (q_4, q_1) states $q_4, \ldots, q_1$ are popped and q_1 is pushed back again.
After exploring (q_5, q_4), since q_4 is not in the stack, q_0 is wrongly popped.

Problem and final attempt

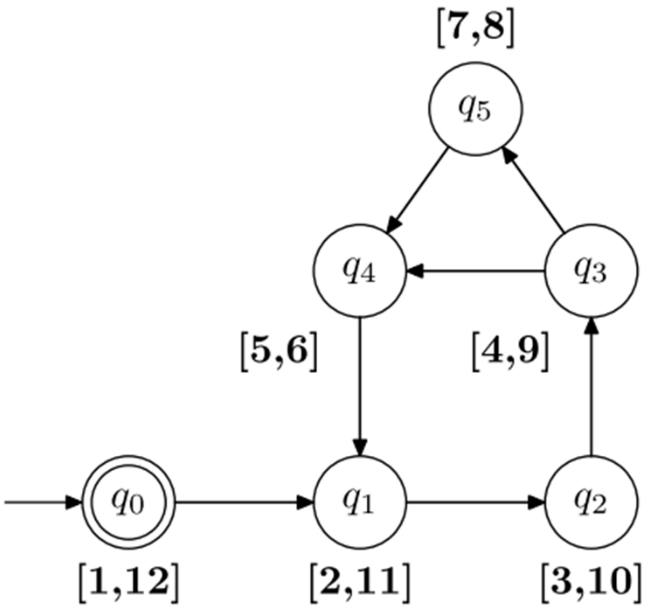


Final attempt:

- push a state when it is discovered.
- when exploring (q, r), if r has already been discovered and $r \rightsquigarrow q$, then pop until r or some state discovered before r is popped, and then push this state back.

After exploring (q_4, q_1) states $q_4, \dots, q_1$ are popped and q_1 is pushed back again.
After exploring (q_5, q_4), since q_4 is not in the stack, q_0 is wrongly popped.

Problem and final attempt



After exploring (q_4, q_1) states $q_4, \ldots, q_1$ are popped and q_1 is pushed back again.
After exploring (q_5, q_4), since q_4 is not in the stack, q_0 is wrongly popped.

Final attempt:

- push a state when it is discovered.
- when exploring (q, r), if r has already been discovered and $r \leadsto q$, then pop until r or some state discovered before r is popped, and then push this state back.

We will show: a state belongs to a cycle iff it is popped at least once before it is blackened.

The OneStack algorithm

OneStack(A)

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

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

$S, C \leftarrow \emptyset$;
dfs(q_0)
report EMP

dfs(q)
    add q **to** S ; **push**(q, C)
    for all $r \in \delta(q)$ **do**
        if $r \notin S$ **then** *dfs*(r)
        else if $r \rightsquigarrow q$ **then**
            repeat
                $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP
            until $d[s] \leq d[r]$
            push(s, C)
    if top$(C) = q$ **then pop**(C)

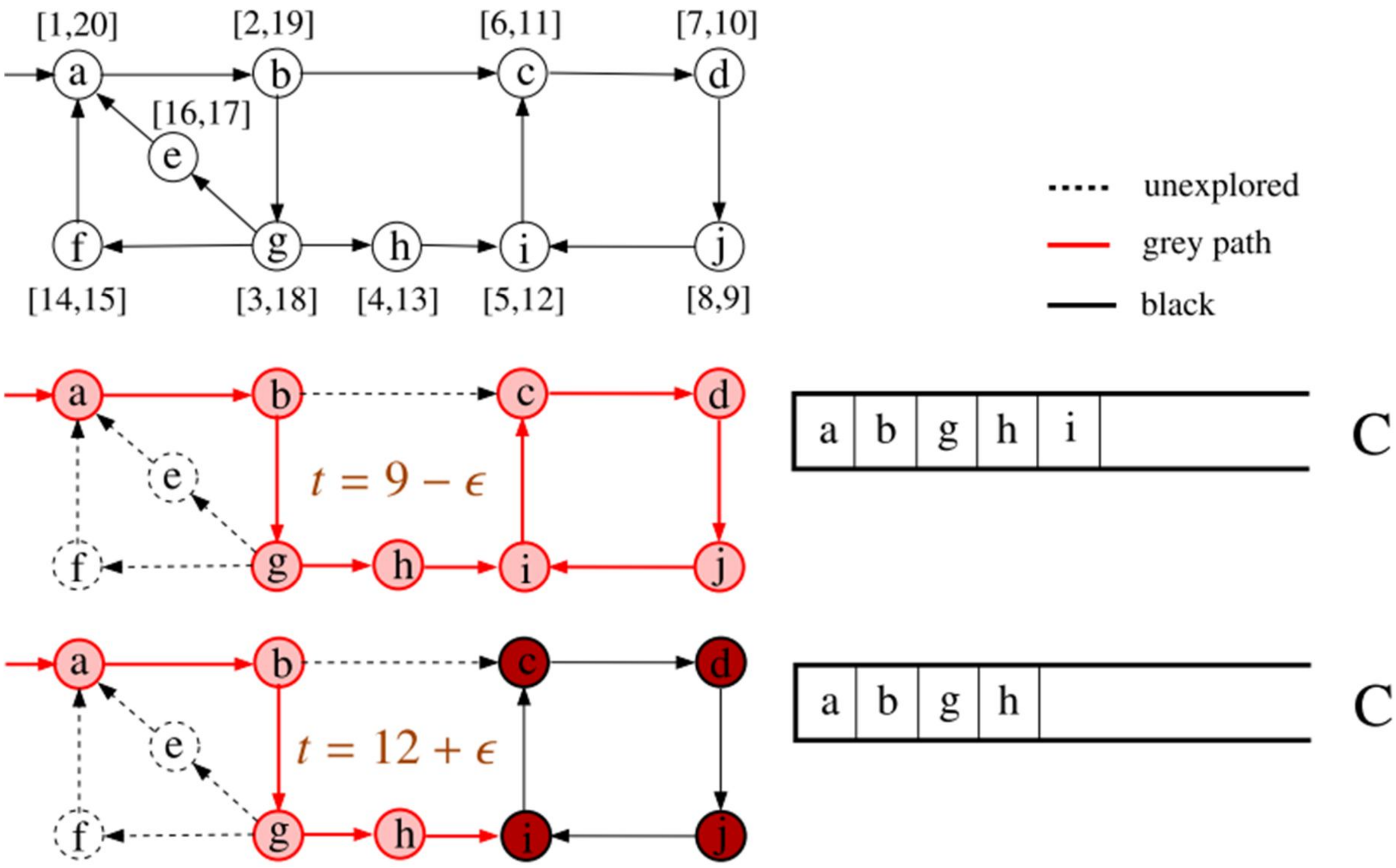
The OneStack algorithm

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

```
S, C ← ∅;
dfs(q0)
report EMP

dfs(q)
    add q to S ; p
    for all r ∈ δ(q) d
        if r ∉ S then dfs(r)
        else if r ⇝ q then
            repeat
                s ← pop(C); if s ∈ F then report NEMP
            until d[s] ≤ d[r]
            push(s, C)
    if top(C) = q then pop(C)
```

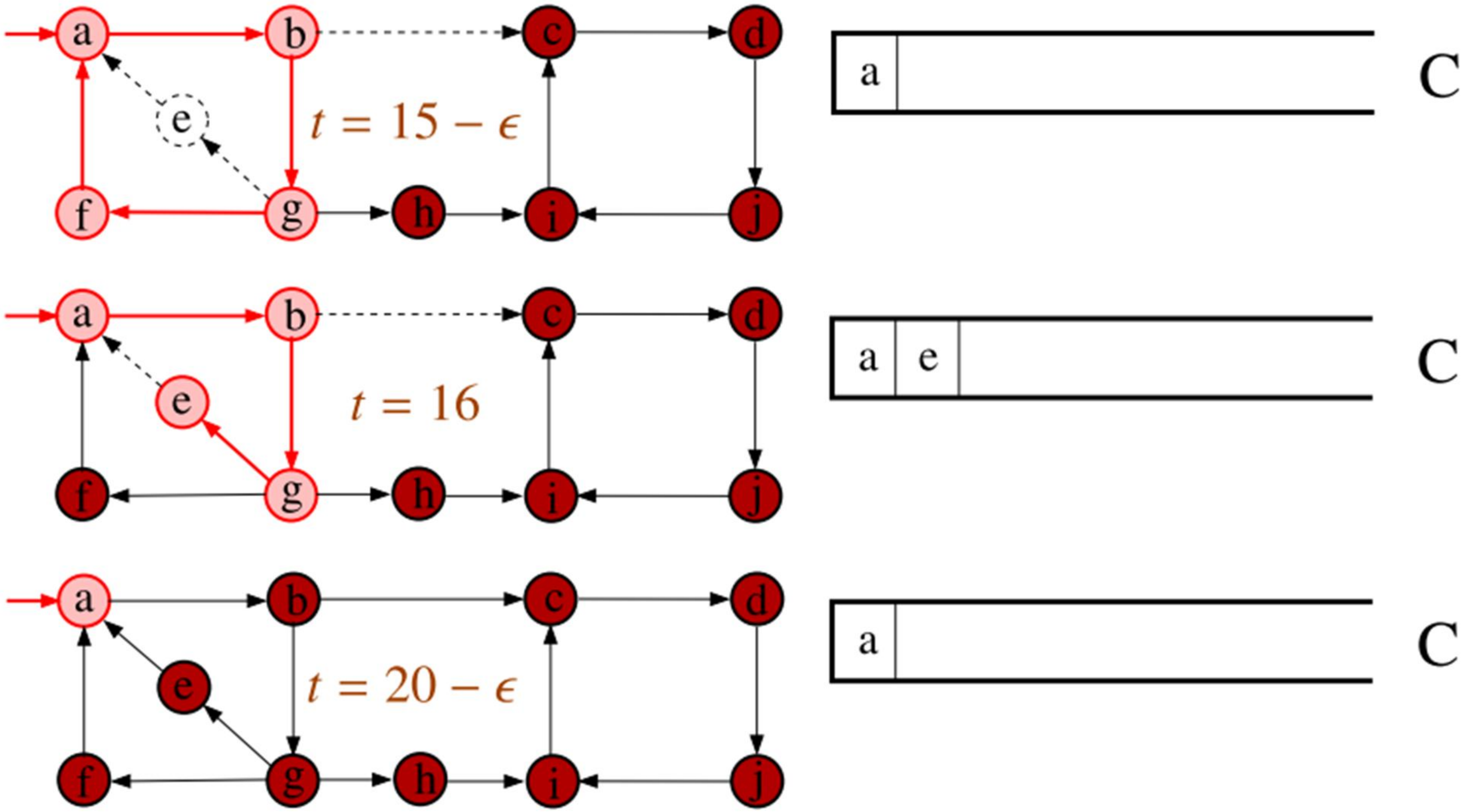
Oracle

An example



[1,20]
[2,19]
[6,11]
[7,10]
a
b
c
d
[16,17]
e
f
g
h
i
j
[14,15]
[3,18]
[4,13]
[5,12]
[8,9]
unexplored
grey path
black
t = 9 − ϵ
a b g h i
C
t = 12 + ϵ
a b g h
C

An example



Questions

- Is *OneStack* correct ?

  Proof obligations:

  1) Every node that belongs to some cycle is eventually popped by the repeat loop.
  2) Every node that is popped by the repeat loop belongs to a cycle.

- Is *OneStack* optimal ?

All nodes in cycles are eventually popped

Proposition. If q belongs to a cycle, then q is eventually popped by the repeat loop.

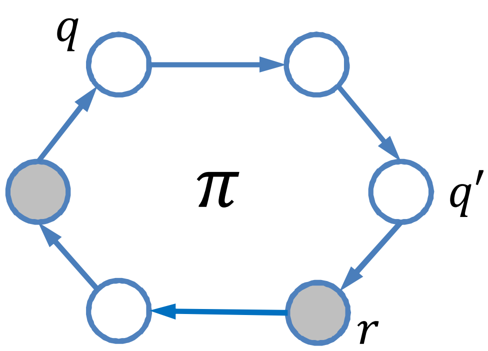
```
OneStack(A)
Input: NBA A = (Q, Σ, δ, Q0, F)
Output: EMP if Lω(A) = ∅, NEMP otherwise
S, C ← ∅;
dfs(q0)
report EMP

dfs(q)
  add q to S ; push(q, C)
  for all r ∈ δ(q) do
    if r ∉ S then dfs(r)
    else if r ⇝ q then
      repeat
        s ← pop(C); if s ∈ F then report NEMP
      until d[s] ≤ d[r]
      push(s, C)
  if top(C) = q then pop(C)
```

All nodes in cycles are eventually popped

Proposition. If q belongs to a cycle, then q is eventually popped by the repeat loop.

Proof.

- π: cycle containing q
- q': last successor of q in π such that at time $d[q]$ there is white path from q to q'
- r: successor of q' in π



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

```
S, C ← ∅;
dfs(q0)
report EMP

dfs(q)
   add q to S ; push(q, C)
   for all r ∈ δ(q) do
      if r ∉ S then dfs(r)
      else if r ⇝ q then
         repeat
            s ← pop(C); if s ∈ F then report NEMP
         until d[s] ≤ d[r]
         push(s, C)
   if top(C) = q then pop(C)
```

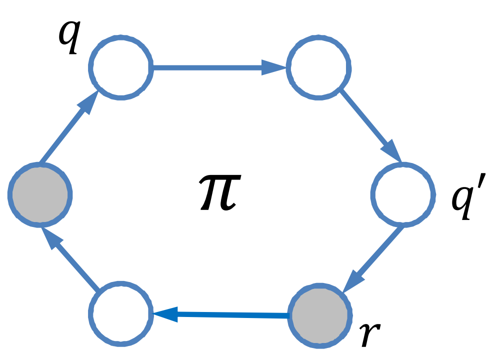
All nodes in cycles are eventually popped

Proposition. If q belongs to a cycle, then q is eventually popped by the repeat loop.

Proof.

- π: cycle containing q
- q': last successor of q in π such that at time $d[q]$ there is white path from q to q'
- r: successor of q' in π

At time $d[q]$ we have $d[r] \leq d[q] \leq d[q']$.



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

$S, C \leftarrow \emptyset$;
dfs(q_0)
report EMP

dfs(q)
  add q **to** S; **push**(q, C)
  for all $r \in \delta(q)$ **do**
    if $r \notin S$ **then** *dfs*(r)
    else if $r \rightsquigarrow q$ **then**
      repeat
        $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP
      until $d[s] \leq d[r]$
      push(s, C)
  if top$(C) = q$ **then pop**(C)

All nodes in cycles are eventually popped

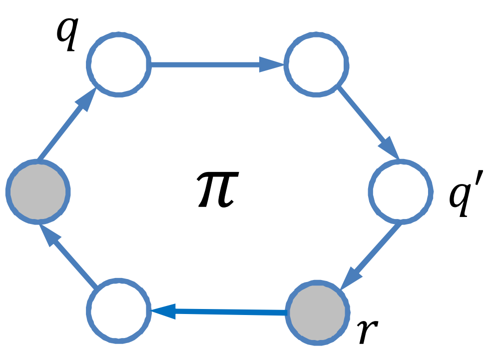
Proposition. If q belongs to a cycle, then q is eventually popped by the repeat loop.

Proof.

- π: cycle containing q
- q': last successor of q in π such that at time $d[q]$ there is white path from q to q'
- r: successor of q' in π

At time $d[q]$ we have $d[r] \leq d[q] \leq d[q']$.

By the White-Path Theorem q' is a descendant of q, and so (q', r) is explored before q is blackened.



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

```
S, C ← ∅;
dfs(q0)
report EMP

dfs(q)
  add q to S; push(q, C)
  for all r ∈ δ(q) do
    if r ∉ S then dfs(r)
    else if r ⇝ q then
      repeat
        s ← pop(C); if s ∈ F then report NEMP
      until d[s] ≤ d[r]
      push(s, C)
  if top(C) = q then pop(C)
```

All nodes in cycles are eventually popped

Proposition. If q belongs to a cycle, then q is eventually popped by the repeat loop.

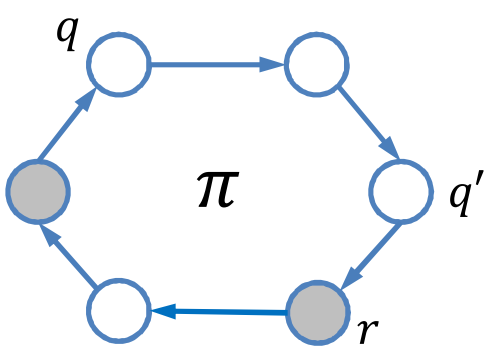
Proof.

- π: cycle containing q
- q': last successor of q in π such that at time $d[q]$ there is white path from q to q'
- r: successor of q' in π

At time $d[q]$ we have $d[r] \leq d[q] \leq d[q']$.

By the White-Path Theorem q' is a descendant of q, and so (q', r) is explored before q is blackened.

So when (q', r) is explored, q has not been popped at line 13.



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

1 $S, C \leftarrow \emptyset$;
2 dfs(q_0)
3 **report** EMP

4 *dfs*(q)
5 **add** q **to** S ; **push**(q, C)
6 **for all** $r \in \delta(q)$ **do**
7 **if** $r \notin S$ **then** *dfs*(r)
8 **else if** $r \rightsquigarrow q$ **then**
9 **repeat**
10 $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP
11 **until** $d[s] \leq d[r]$
12 **push**(s, C)
13 **if top**$(C) = q$ **then pop**(C)

All nodes in cycles are eventually popped

Proposition. If q belongs to a cycle, then q is eventually popped by the repeat loop.

Proof.

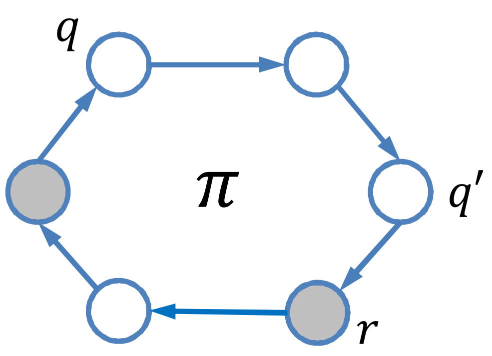
- π: cycle containing q
- q': last successor of q in π such that at time $d[q]$ there is white path form q to q'
- r: successor of q' in π

At time $d[q]$ we have $d[r] \leq d[q] \leq d[q']$.

By the White-Path Theorem q' is a descendant of q, and so (q', r) is explored before q is blackened.

So when (q', r) is explored, q has not been popped at line 13.

Since $r \rightsquigarrow q'$, either q has already been popped before or it is popped now because $d[r] \leq d[q']$.



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

1 $S, C \leftarrow \emptyset$;
2 dfs(q_0)
3 **report** EMP

4 *dfs*(q)
5   **add** q **to** S; **push**(q, C)
6   **for all** $r \in \delta(q)$ **do**
7   **if** $r \notin S$ **then** *dfs*(r)
8   **else if** $r \rightsquigarrow q$ **then**
9   **repeat**
10         $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP
11   **until** $d[s] \leq d[r]$
12   **push**(s, C)
13   **if top**$(C) = q$ **then pop**(C)

All nodes in cycles are eventually popped

Proposition. If q belongs to a cycle, then q is eventually popped by the repeat loop.

Proof.

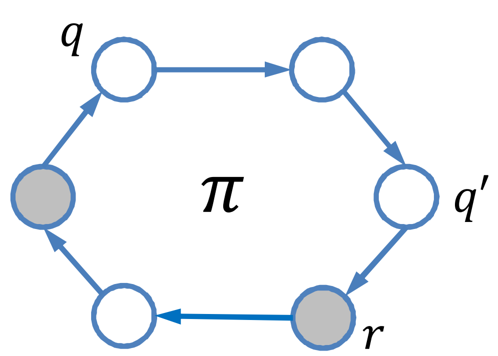
- π: cycle containing q
- q': last successor of q in π such that at time $d[q]$ there is white path form q to q'
- r: successor of q' in π

At time $d[q]$ we have $d[r] \leq d[q] \leq d[q']$.

By the White-Path Theorem q' is a descendant of q, and so (q', r) is explored before q is blackened.

So when (q', r) is explored, q has not been popped at line 13.

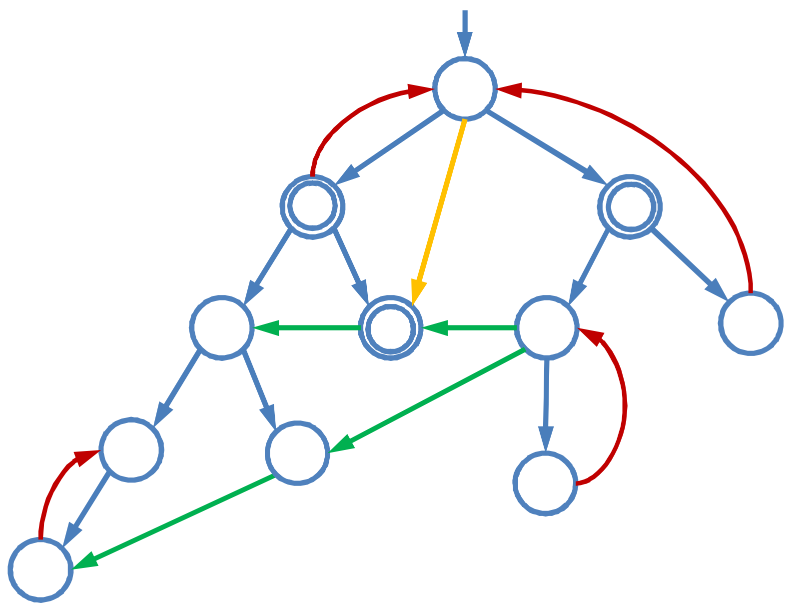
Since $r \rightsquigarrow q'$, either q has already been popped before or it is popped now because $d[r] \leq d[q']$.



This proof also shows optimality:
q is popped immediately after the DFS explores all transitions of π, or earlier.
Since π is an arbitrary cycle, $OneStack$ is optimal.

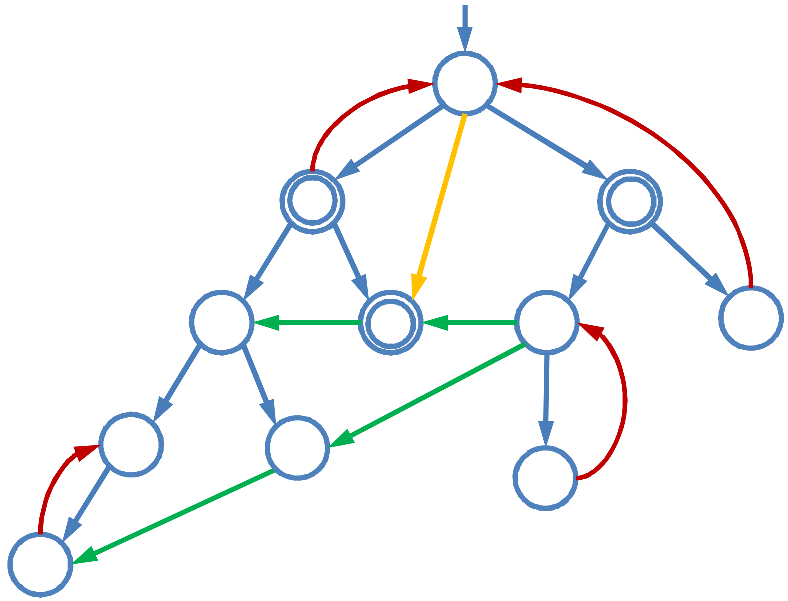
All popped nodes belong to cycles

- To show that every node popped by the repeat loop belongs to a cycle we need some concepts:
  - strongly connected component (scc) of a graph



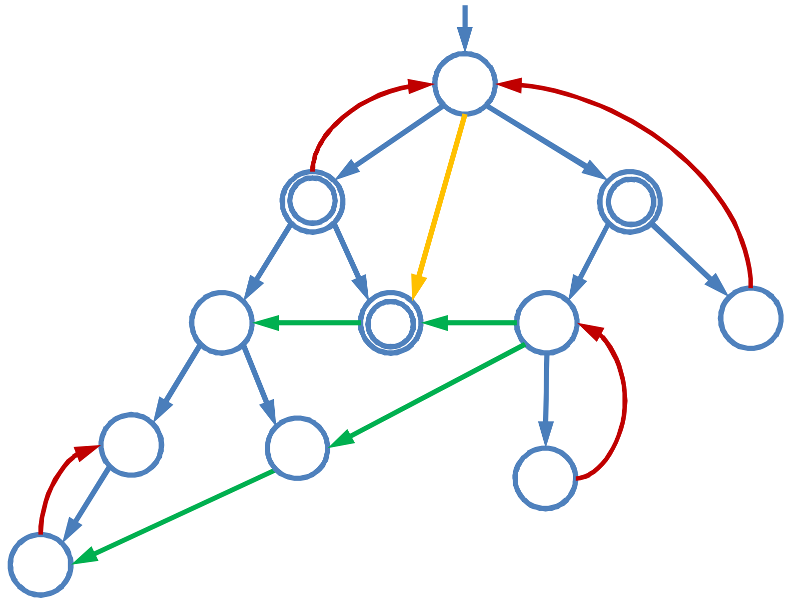
All popped nodes belong to cycles

- To show that every node popped by the repeat loop belongs to a cycle we need some concepts:
  - strongly connected component (scc) of a graph
  - dag of sccs of a graph



All popped nodes belong to cycles

- To show that every node popped by the repeat loop belongs to a cycle we need some concepts:
  - strongly connected component (scc) of a graph
  - dag of sccs of a graph
  - root of an scc in a DFS.



All popped nodes belong to cycles

Invariant of *OneStack*: The repeat loop cannot remove a grey root ρ from the stack (remove = pop and don't push back), and can only pop states s such that $d[s] \geq d[\rho]$.

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

$S, C \leftarrow \emptyset$;
dfs(q_0)
report EMP

dfs(q)
  add q **to** S; **push**(q, C)
  for all $r \in \delta(q)$ **do**
    if $r \notin S$ **then** *dfs*(r)
    else if $r \rightsquigarrow q$ **then**
      repeat
        $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP
      until $d[s] \leq d[r]$
      push(s, C)
  if top$(C) = q$ **then pop**(C)

All popped nodes belong to cycles

Invariant of *OneStack*: The repeat loop cannot remove a grey root ρ from the stack (remove = pop and don't push back), and can only pop states s such that $d[s] \geq d[\rho]$.

Proof (sketch):

t: time at which repeat loop starts because $r \rightsquigarrow q$ for some (q, r).

ρ: grey root at time t.

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

```
S, C ← ∅;
dfs(q0)
report EMP

dfs(q)
  add q to S; push(q, C)
  for all r ∈ δ(q) do
    if r ∉ S then dfs(r)
    else if r ⇝ q then
      repeat
        s ← pop(C); if s ∈ F then report NEMP
      until d[s] ≤ d[r]
      push(s, C)
  if top(C) = q then pop(C)
```

All popped nodes belong to cycles

Invariant of *OneStack*: The repeat loop cannot remove a grey root ρ from the stack (remove = pop and don't push back), and can only pop states s such that $d[s] \geq d[\rho]$.

Proof (sketch):

t: time at which repeat loop starts because $r \rightsquigarrow q$ for some (q, r).

ρ: grey root at time t.

r and q belong to the same scc.

ρ': root of this scc.

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

$S, C \leftarrow \emptyset$;
dfs(q_0)
report EMP

dfs(q)
  add q **to** S; **push**(q, C)
  for all $r \in \delta(q)$ **do**
    if $r \notin S$ **then** *dfs*(r)
    else if $r \rightsquigarrow q$ **then**
      repeat
        $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP
      until $d[s] \leq d[r]$
      push(s, C)
  if top$(C) = q$ **then pop**(C)

All popped nodes belong to cycles

Invariant of *OneStack*: The repeat loop cannot remove a grey root ρ from the stack (remove = pop and don't push back), and can only pop states s such that $d[s] \geq d[\rho]$.

Proof (sketch):

- t: time at which repeat loop starts because $r \rightsquigarrow q$ for some (q, r).
- ρ: grey root at time t.

r and q belong to the same scc.

- ρ': root of this scc.

q, ρ, and ρ' are grey at time t, and $q \rightsquigarrow \rho' \rightsquigarrow q$.

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

```
S, C ← ∅;
dfs(q0)
report EMP

dfs(q)
  add q to S; push(q, C)
  for all r ∈ δ(q) do
    if r ∉ S then dfs(r)
    else if r ⇝ q then
      repeat
        s ← pop(C); if s ∈ F then report NEMP
      until d[s] ≤ d[r]
      push(s, C)
  if top(C) = q then pop(C)
```

All popped nodes belong to cycles

Invariant of *OneStack*: The repeat loop cannot remove a grey root ρ from the stack (remove = pop and don't push back), and can only pop states s such that $d[s] \geq d[\rho]$.

Proof (sketch):

> t: time at which repeat loop starts because $r \rightsquigarrow q$ for some (q, r).
>
> ρ: grey root at time t.

r and q belong to the same scc.

> ρ': root of this scc.

q, ρ, and ρ' are grey at time t, and $q \rightsquigarrow \rho' \rightsquigarrow q$.

By the grey-path theorem and since ρ is root, we have $\rho \Rightarrow q \Rightarrow \rho'$ and so $d[\rho] \leq d[\rho'] \leq d[r]$.

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

```
S, C ← ∅;
dfs(q0)
report EMP

dfs(q)
  add q to S ; push(q, C)
  for all r ∈ δ(q) do
    if r ∉ S then dfs(r)
    else if r ⇝ q then
      repeat
        s ← pop(C); if s ∈ F then report NEMP
      until d[s] ≤ d[r]
      push(s, C)
  if top(C) = q then pop(C)
```

All popped nodes belong to cycles

Invariant of *OneStack*: The repeat loop cannot remove a grey root ρ from the stack (remove = pop and don't push back), and can only pop states s such that $d[s] \geq d[\rho]$.

Proof (sketch):

> t: time at which repeat loop starts because $r \rightsquigarrow q$ for some (q, r).
>
> ρ: grey root at time t.

r and q belong to the same scc.

> ρ': root of this scc.

q, ρ, and ρ' are grey at time t, and $q \rightsquigarrow \rho' \rightsquigarrow q$.

By the grey-path theorem and since ρ is root, we have $\rho \Rightarrow q \Rightarrow \rho'$ and so $d[\rho] \leq d[\rho'] \leq d[r]$.

So every state s popped by the repeat loop satisfies $d[s] \geq d[q]$.

Further, if ρ is popped, then it is pushed immediately after at line 12.

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

1 $S, C \leftarrow \emptyset$;
2 dfs(q_0)
3 **report** EMP

4 *dfs*(q)
5   **add** q **to** S ; **push**(q, C)
6   **for all** $r \in \delta(q)$ **do**
7   **if** $r \notin S$ **then** *dfs*(r)
8   **else if** $r \rightsquigarrow q$ **then**
9   **repeat**
10         $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP
11   **until** $d[s] \leq d[r]$
12   **push**(s, C)
13   **if top**$(C) = q$ **then pop**(C)

All popped nodes belong to cycles

Proposition: Any state popped by the repeat loop belongs to a cycle.

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

```
S, C ← ∅;
dfs(q0)
report EMP

dfs(q)
  add q to S ; push(q, C)
  for all r ∈ δ(q) do
    if r ∉ S then dfs(r)
    else if r ⇝ q then
      repeat
        s ← pop(C); if s ∈ F then report NEMP
      until d[s] ≤ d[r]
      push(s, C)
  if top(C) = q then pop(C)
```

All popped nodes belong to cycles

Proposition: Any state popped by the repeat loop belongs to a cycle.

Proof (sketch):

s: state popped by the repeat loop

t: time at which the repeat loop starts popping

(q, r): transition being currently explored ($r \rightsquigarrow q$).

ρ: root of the scc of r and q

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

$S, C \leftarrow \emptyset$;
dfs(q_0)
report EMP

dfs(q)
    add q **to** S; **push**(q, C)
    for all $r \in \delta(q)$ **do**
        if $r \notin S$ **then** *dfs*(r)
        else if $r \rightsquigarrow q$ **then**
            repeat
                $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP
            until $d[s] \leq d[r]$
            push(s, C)
    if top(C) $= q$ **then pop**(C)

All popped nodes belong to cycles

Proposition: Any state popped by the repeat loop belongs to a cycle.

Proof (sketch):

s: state popped by the repeat loop

t: time at which the repeat loop starts popping

(q, r): transition being currently explored ($r \rightsquigarrow q$).

ρ: root of the scc of r and q

Observe: q, s, ρ are grey at time t

1. $s \Rightarrow q$. Because s, q grey at time t and dfs(q) is being currently executed.

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

```
S, C ← ∅;
dfs(q0)
report EMP

dfs(q)
  add q to S ; push(q, C)
  for all r ∈ δ(q) do
    if r ∉ S then dfs(r)
    else if r ⇝ q then
      repeat
        s ← pop(C); if s ∈ F then report NEMP
      until d[s] ≤ d[r]
      push(s, C)
  if top(C) = q then pop(C)
```

All popped nodes belong to cycles

Proposition: Any state popped by the repeat loop belongs to a cycle.

Proof (sketch):

s: state popped by the repeat loop

t: time at which the repeat loop starts popping

(q, r): transition being currently explored ($r \rightsquigarrow q$).

ρ: root of the scc of r and q

Observe: q, s, ρ are grey at time t

1. $s \Rightarrow q$. Because s, q grey at time t and dfs(q) is being currently executed.
2. $\rho \Rightarrow s$. Since ρ, q grey at time t and ρ is root we have $\rho \Rightarrow q$, By 1) either $\rho \Rightarrow s$ or $s \Rightarrow \rho$. By the invariant $d[\rho] \leq d[s]$ and so $\rho \Rightarrow s$.

```
OneStack(A)
Input: NBA A = (Q, Σ, δ, Q0, F)
Output: EMP if Lω(A) = ∅, NEMP otherwise
  S, C ← ∅;
  dfs(q0)
  report EMP

  dfs(q)
    add q to S ; push(q, C)
    for all r ∈ δ(q) do
      if r ∉ S then dfs(r)
      else if r ⇝ q then
        repeat
          s ← pop(C); if s ∈ F then report NEMP
        until d[s] ≤ d[r]
        push(s, C)
    if top(C) = q then pop(C)
```

All popped nodes belong to cycles

Proposition: Any state popped by the repeat loop belongs to a cycle.

Proof (sketch):

s: state popped by the repeat loop

t: time at which the repeat loop starts popping

(q, r): transition being currently explored ($r \rightsquigarrow q$).

ρ: root of the scc of r and q

Observe: q, s, ρ are grey at time t

1. $s \Rightarrow q$. Because s, q grey at time t and dfs(q) is being currently executed.
2. $\rho \Rightarrow s$. Since ρ, q grey at time t and ρ is root we have $\rho \Rightarrow q$, By 1) either $\rho \Rightarrow s$ or $s \Rightarrow \rho$. By the invariant $d[\rho] \leq d[s]$ and so $\rho \Rightarrow s$.

By 1) and 2) we have $\rho \rightsquigarrow s \rightsquigarrow q \rightsquigarrow r \rightsquigarrow \rho$, and so s belongs to a cycle.

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

```
S, C ← ∅;
dfs(q0)
report EMP

dfs(q)
  add q to S ; push(q, C)
  for all r ∈ δ(q) do
    if r ∉ S then dfs(r)
    else if r ⇝ q then
      repeat
        s ← pop(C); if s ∈ F then report NEMP
      until d[s] ≤ d[r]
      push(s, C)
  if top(C) = q then pop(C)
```

Implementing the oracle

Assume *OneStack* calls the oracle for $r \rightsquigarrow q$. We look for a condition that holds at that moment iff $r \rightsquigarrow q$ holds, and is easy to check.

Implementing the oracle

Assume *OneStack* calls the oracle for $r \rightsquigarrow q$. We look for a condition that holds at that moment iff $r \rightsquigarrow q$ holds, and is easy to check.

Lemma. Assume *OneStack* is exploring (q, r) and r is already discovered. Let R be the scc of r. Then $r \rightsquigarrow q$ iff some state of R is not black.

Implementing the oracle

Assume *OneStack* calls the oracle for $r \rightsquigarrow q$. We look for a condition that holds at that moment iff $r \rightsquigarrow q$ holds, and is easy to check.

Lemma. Assume *OneStack* is exploring (q, r) and r is already discovered. Let R be the scc of r. Then $r \rightsquigarrow q$ iff some state of R is not black.

Proof. (⇒) Then $r, q \in R$ and q is not black.

(⇐) At least one $s \in R$ is grey. By the grey-path theorem there is a grey path $s \Rightarrow q$. So $r \rightsquigarrow s \Rightarrow q$.

Implementing the oracle

- Idea: maintain a set V of active states whose sccs have not yet been completely explored (not yet black)

Implementing the oracle

- Idea: maintain a set V of active states whose sccs have not yet been completely explored (not yet black)
- Since the root is the first state of an scc to be greyed and the last to be blackened, we can proceed as follows:
  - Add states to V when they are discovered.
  - Remove states from V when the root of their sccs is blackened.

Implementing the oracle

- Idea: maintain a set V of active states whose sccs have not yet been completely explored (not yet black)
- Since the root is the first state of an scc to be greyed and the last to be blackened, we can proceed as follows:
  - Add states to V when they are discovered.
  - Remove states from V when the root of their sccs is blackened.
- So V can be implemented as a stack: when a root ρ is blackened, pop from V until ρ is popped.

Implementing the oracle

- Idea: maintain a set V of active states whose sccs have not yet been completely explored (not yet black)
- Since the root is the first state of an scc to be greyed and the last to be blackened, we can proceed as follows:
  - Add states to V when they are discovered.
  - Remove states from V when the root of their sccs is blackened.
- So V can be implemented as a stack: when a root ρ is blackened, pop from V until ρ is popped.
- Problem to solve: when blackening a node, decide if it is a root.

Implementing the oracle

Lemma. At line 13, q is a root iff $\mathrm{top}(C) = q$.

OneStack(A)

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

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

1 $S, C \leftarrow \emptyset$;

2 dfs(q_0)

3 **report** EMP

4 *dfs*(q)

5   **add** q **to** S ; **push**(q, C)

6   **for all** $r \in \delta(q)$ **do**

7   **if** $r \notin S$ **then** *dfs*(r)

8   **else if** $r \rightsquigarrow q$ **then**

9   **repeat**

10         $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP

11   **until** $d[s] \leq d[r]$

12   **push**(s, C)

13   **if top**$(C) = q$ **then pop**(C)

Implementing the oracle

Lemma. At line 13, q is a root iff $\mathrm{top}(C) = q$.

Proof. (⇒) If q is root, by the invariant it still belongs to C after the for-loop, and so $\mathrm{top}(C) = q$.

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

4   dfs(q)
5     add q to S; push(q, C)
6     for all r ∈ δ(q) do
7       if r ∉ S then dfs(r)
8       else if r ⇝ q then
9         repeat
10          s ← pop(C); if s ∈ F then report NEMP
11        until d[s] ≤ d[r]
12        push(s, C)
13    if top(C) = q then pop(C)
```

Implementing the oracle

Lemma. At line 13, q is a root iff $\text{top}(C) = q$.

Proof. ($\Rightarrow$) If q is root, by the invariant it still belongs to C after the for-loop, and so $\text{top}(C) = q$.

($\Leftarrow$)

- ρ: root of scc of q, different from q
- π: path from ρ to q
- r: first state of π s.t. $d[r] < d[q]$
- q': successor of r in π

```
OneStack(A)
Input: NBA A = (Q, Σ, δ, Q_0, F)
Output: EMP if L_ω(A) = ∅, NEMP otherwise
1   S, C ← ∅;
2   dfs(q_0)
3   report EMP

4   dfs(q)
5     add q to S; push(q, C)
6     for all r ∈ δ(q) do
7       if r ∉ S then dfs(r)
8       else if r ⇝ q then
9         repeat
10          s ← pop(C); if s ∈ F then report NEMP
11        until d[s] ≤ d[r]
12        push(s, C)
13    if top(C) = q then pop(C)
```

Implementing the oracle

Lemma. At line 13, q is a root iff $\mathrm{top}(C) = q$.

Proof. ($\Rightarrow$) If q is root, by the invariant it still belongs to C after the for-loop, and so $\mathrm{top}(C) = q$.

($\Leftarrow$)

ρ: root of scc of q, different from q

π: path from ρ to q

r: first state of π s.t. $d[r] < d[q]$

q': successor of r in π

The white-path theorem gives $q \Rightarrow q'$.

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

1 $S, C \leftarrow \emptyset$;
2 dfs(q_0)
3 **report** EMP

4 *dfs*(q)
5   **add** q **to** S; **push**(q, C)
6   **for all** $r \in \delta(q)$ **do**
7   **if** $r \notin S$ **then** *dfs*(r)
8   **else if** $r \rightsquigarrow q$ **then**
9   **repeat**
10         $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP
11   **until** $d[s] \leq d[r]$
12   **push**(s, C)
13   **if top**$(C) = q$ **then pop**(C)

Implementing the oracle

Lemma. At line 13, q is a root iff $\mathrm{top}(C) = q$.

Proof. (⇒) If q is root, by the invariant it still belongs to C after the for-loop, and so $\mathrm{top}(C) = q$.

(⇐) ρ: root of scc of q, different from q

π: path from ρ to q

r: first state of π s.t. $d[r] < d[q]$

q': successor of r in π

The white-path theorem gives $q \Rightarrow q'$.

So when (q', r) is explored q is not yet black, and all s s.t. $d[s] > d[r]$ are popped from C and not pushed back.

So either q has already been popped, or it is popped now.

```
OneStack(A)
Input: NBA A = (Q, Σ, δ, Q₀, F)
Output: EMP if L_ω(A) = ∅, NEMP otherwise
 1  S, C ← ∅;
 2  dfs(q₀)
 3  report EMP

 4  dfs(q)
 5    add q to S; push(q, C)
 6    for all r ∈ δ(q) do
 7      if r ∉ S then dfs(r)
 8      else if r ⇝ q then
 9        repeat
10          s ← pop(C); if s ∈ F then report NEMP
11        until d[s] ≤ d[r]
12        push(s, C)
13    if top(C) = q then pop(C)
```

Implementing the oracle

Lemma. At line 13, q is a root iff $\text{top}(C) = q$.

Proof. ($\Rightarrow$) If q is root, by the invariant it still belongs to C after the for-loop, and so $\text{top}(C) = q$.

($\Leftarrow$) ρ: root of scc of q, different from q

π: path from ρ to q

r: first state of π s.t. $d[r] < d[q]$

q': successor of r in π

The white-path theorem gives $q \Rightarrow q'$.

So when (q', r) is explored q is not yet black, and all s s.t. $d[s] > d[r]$ are popped from C and not pushed back.

So either q has already been popped, or it is popped now.

Since q not yet black, at line 13 q is not in C, and so $\text{top}(C) \neq q$.

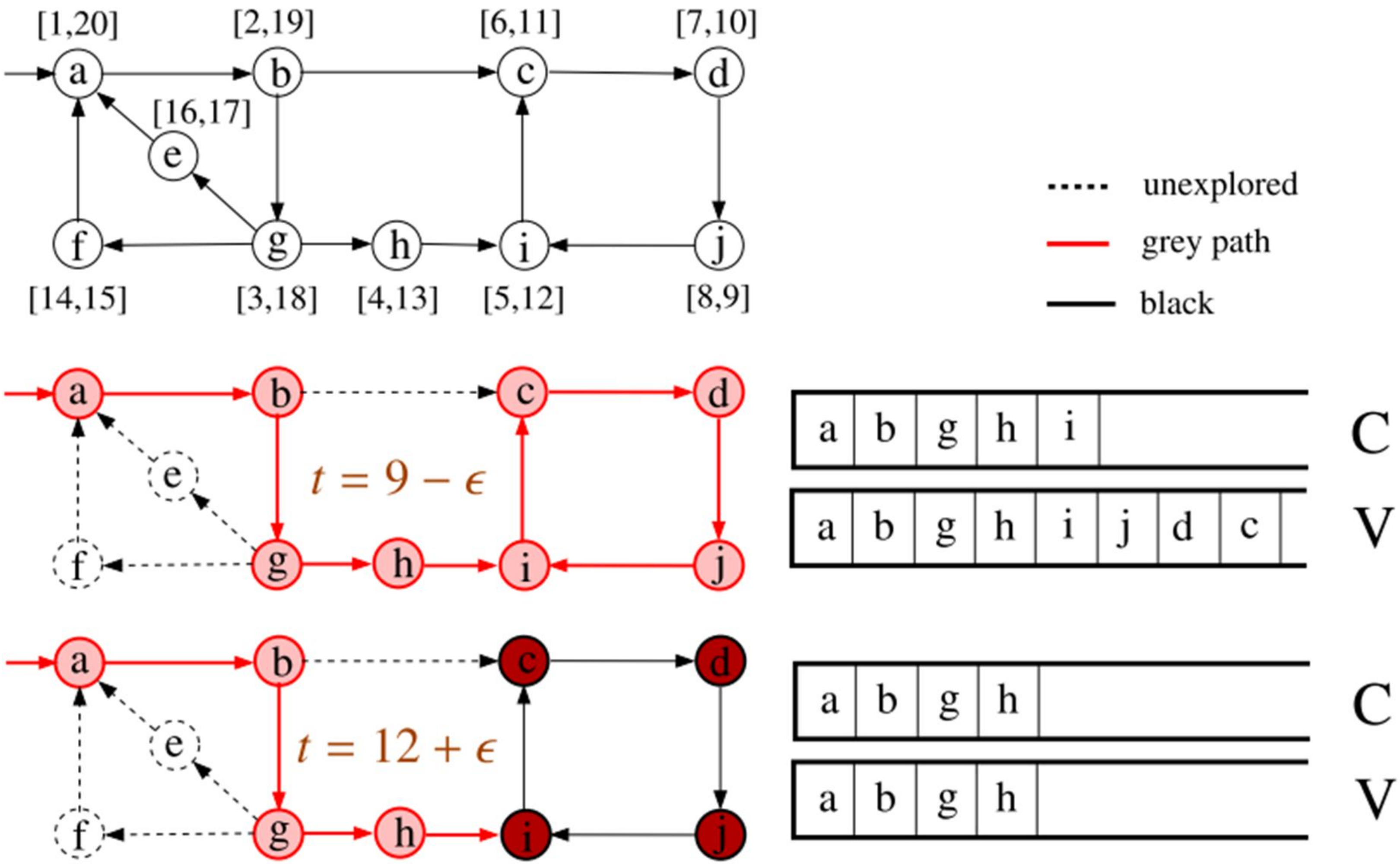
```
OneStack(A)
Input: NBA A = (Q, Σ, δ, Q0, F)
Output: EMP if Lω(A) = ∅, NEMP otherwise
1   S, C ← ∅;
2   dfs(q0)
3   report EMP

4   dfs(q)
5     add q to S ; push(q, C)
6     for all r ∈ δ(q) do
7       if r ∉ S then dfs(r)
8       else if r ⇝ q then
9         repeat
10          s ← pop(C); if s ∈ F then report NEMP
11        until d[s] ≤ d[r]
12        push(s, C)
13    if top(C) = q then pop(C)
```

Implementing the oracle

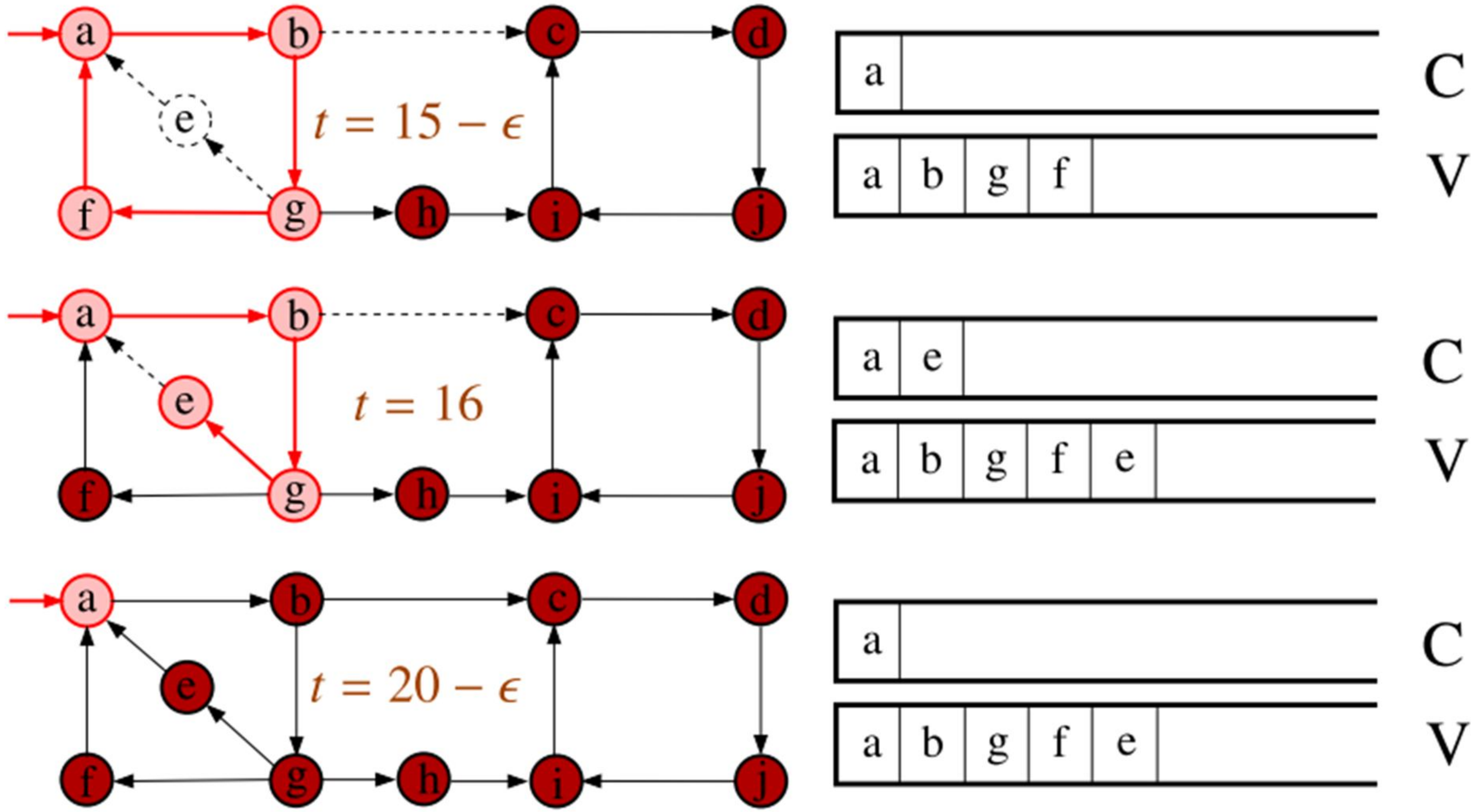
So V can be implemented as a second stack maintained as follows:

- when a state is greyed, it is pushed into V;
- when a root is blackened, all states of V above it (including the root) are popped.

Implementing the oracle



Implementing the oracle



Implementing the oracle

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

$S, C \leftarrow \emptyset$;
dfs(q_0)
report EMP

dfs(q)
  add q **to** S; **push**(q, C)
  for all $r \in \delta(q)$ **do**
    if $r \notin S$ **then** *dfs*(r)
    else if $r \rightsquigarrow q$ **then**
      repeat
        $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP
      until $d[s] \leq d[r]$
      push(s, C)
  if top$(C) = q$ **then pop**(C)

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

$S, C, V \leftarrow \emptyset$;
dfs(q_0)
report EMP

proc *dfs*(q)
  add q **to** S; **push**(q, C); **push**(q, V)
  for all $r \in \delta(q)$ **do**
    if $r \notin S$ **then** *dfs*(r)
    else if $r \in V$ **then**
      repeat
        $s \leftarrow$ **pop**(C); **if** $s \in F$ **then report** NEMP
      until $d[s] \leq d[r]$
      push(s, C)
  if top$(C) = q$ **then**
    pop(C)
    repeat $s \leftarrow$ **pop**(V) **until** $s = q$

Extension to NGAs

TwoStack(A)

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

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

```
S, C, V ← ∅;
dfs(q0)
report EMP

proc dfs(q)
   add q to S; push(q, C);  push(q, V)
   for all r ∈ δ(q) do
      if r ∉ S then dfs(r)
      else if r ∈ V then
         repeat
            s ← pop(C); if s ∈ F then report NEMP
         until d[s] ≤ d[r]
         push(s, C)
   if top(C) = q then
      pop(C)
      repeat s ← pop(V) until s = q
```

TwoStackNGA(A)

Input: NGA $A = (Q, \Sigma, \delta, q_0, \{F_0, \ldots, F_{k-1}\})$

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

```
S, C, V ← ∅;
dfs(q0)
report EMP

proc dfs(q)
   add [q, F(q)] to S; push([q, F(q)], C); push(q, V)
   for all r ∈ δ(q) do
      if r ∉ S then dfs(r)
      else if r ∈ V then
         I ← ∅
         repeat
            [s, J] ← pop(C);
            I ← I ∪ J; if I = K then report NEMP
         until d[s] ≤ d[r]
         push([s, I], C)
   if top(C) = (q, I)for some I then
      pop(C)
      repeat s ← pop(V) until s = q
```