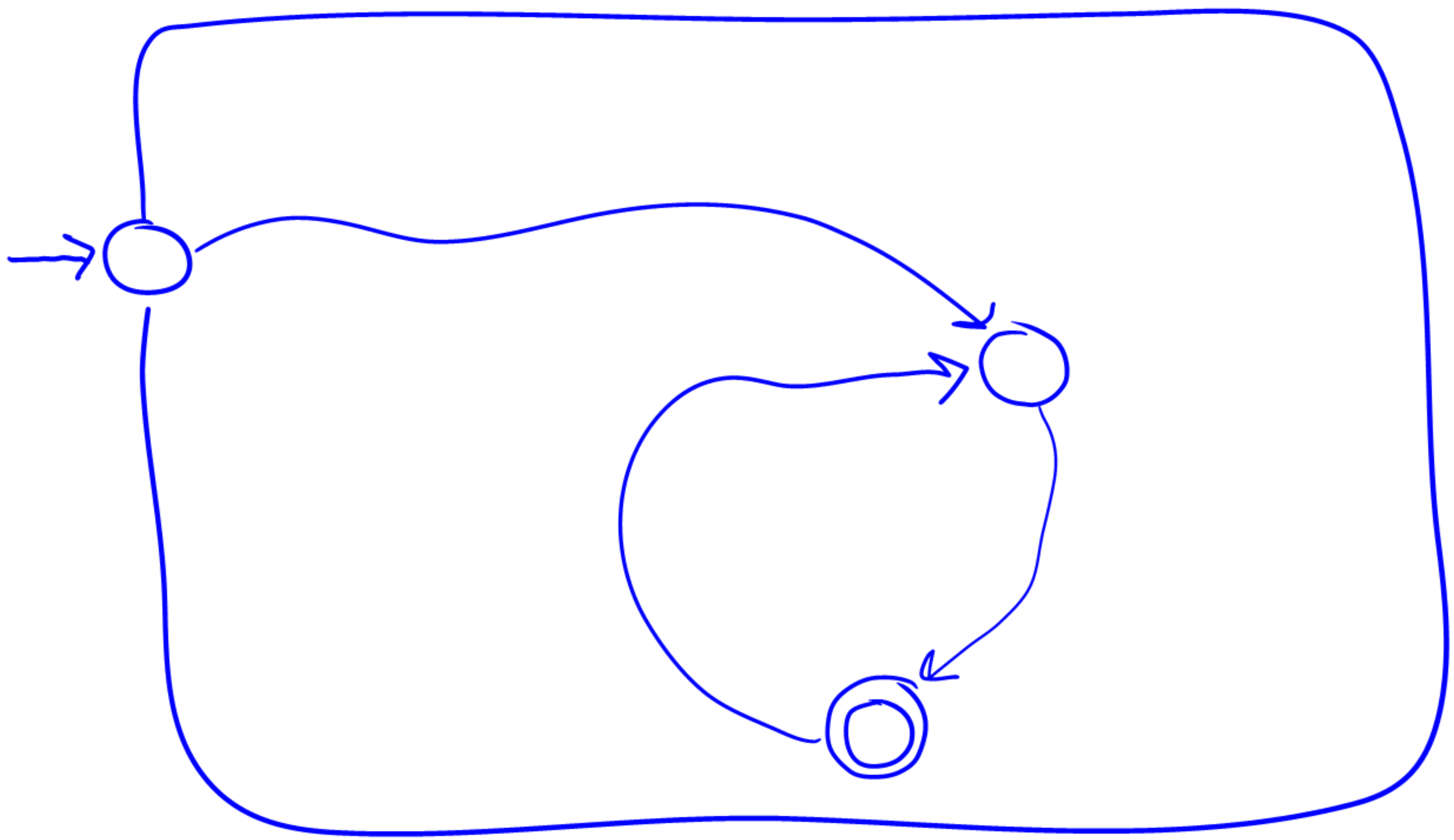
Emptiness-check: Implementations

A NBA is nonempty iff it has an accepting lasso:



Setting

NBA with n states and m transitions.

We are interested in "on the fly" algorithms that check for emptiness of the NBA while constructing it.

We are given:

- the name of the initial state
- an oracle which, supplied with a state of the NBA, returns its set of successors (and for each successor, the information whether they are accepting or not).

Two generic approaches

1. Compute the set of accepting states.
   For every accepting state, check if it belongs to a cycle.

   Nested-depth-first algorithm

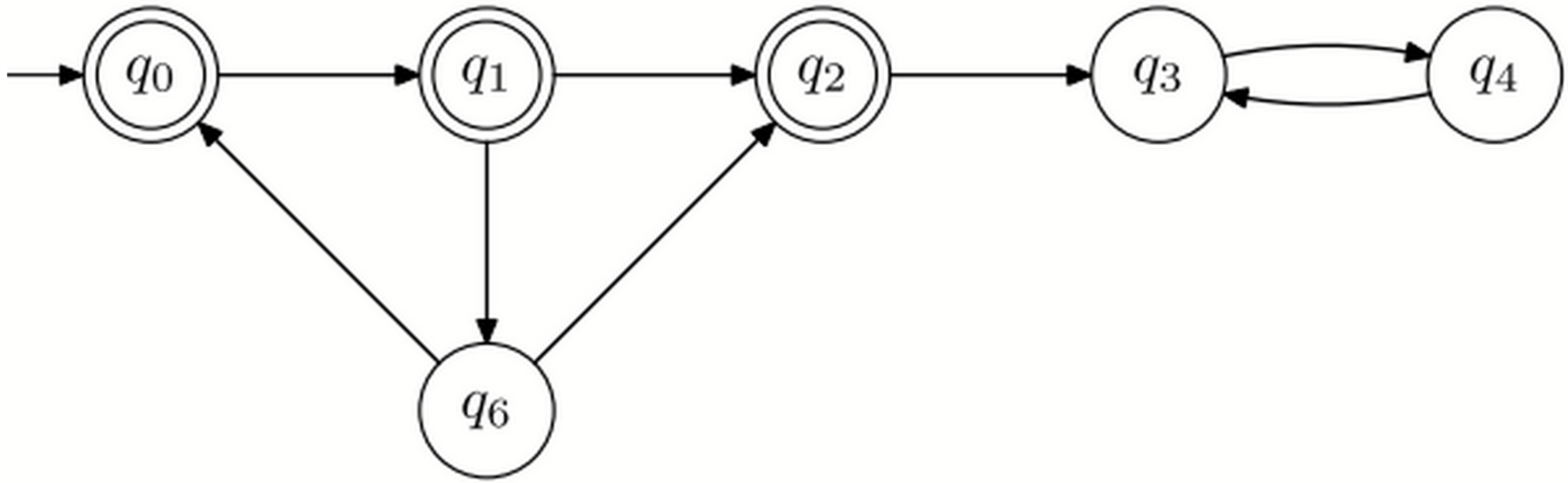
2. Compute the set of states that belong to a cycle
   For each of them, check if it is accepting.

   Two-stack algorithm

The first approach: A naive algorithm

1. Compute all accepting states by means of a search (DFS, BFS, ...)

2. For each accepting state q, conduct a second search (DFS, BFS, ...) to decide if q belongs to a cycle.



Complexity of the first search: O(m)
Number of searches in step 2: O(n)
Complexity of step 2: O(nm)

Overall complexity: O(nm)   <== Far too high!!

We look for a linear algorithm

Recalling some search concepts

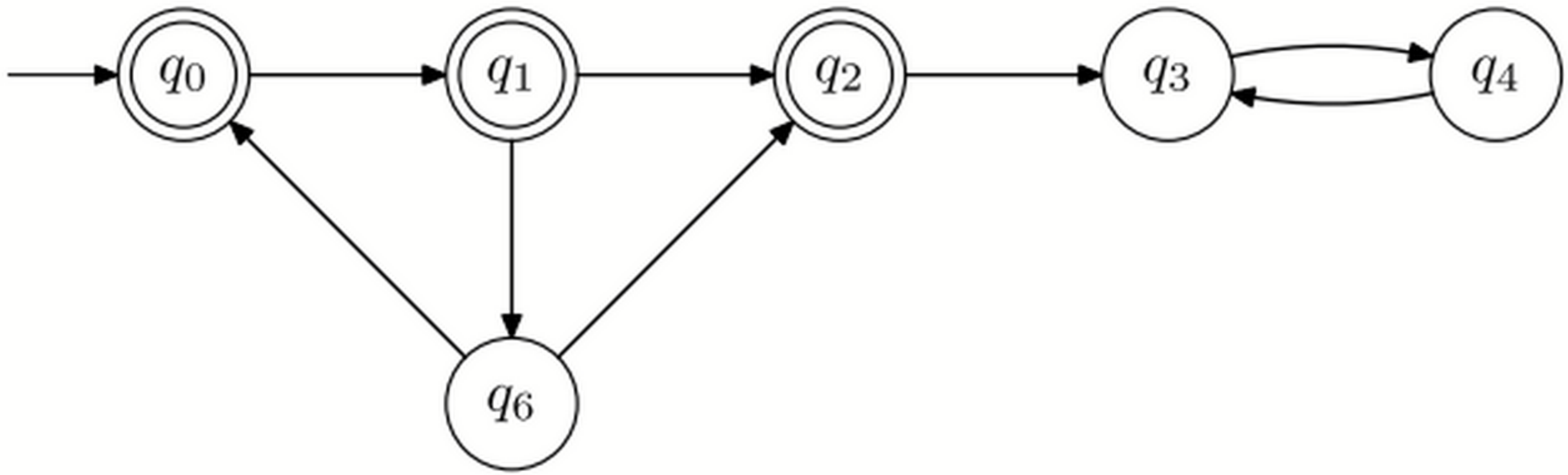
Generic search in graphs: Similar to a worklist algorithm

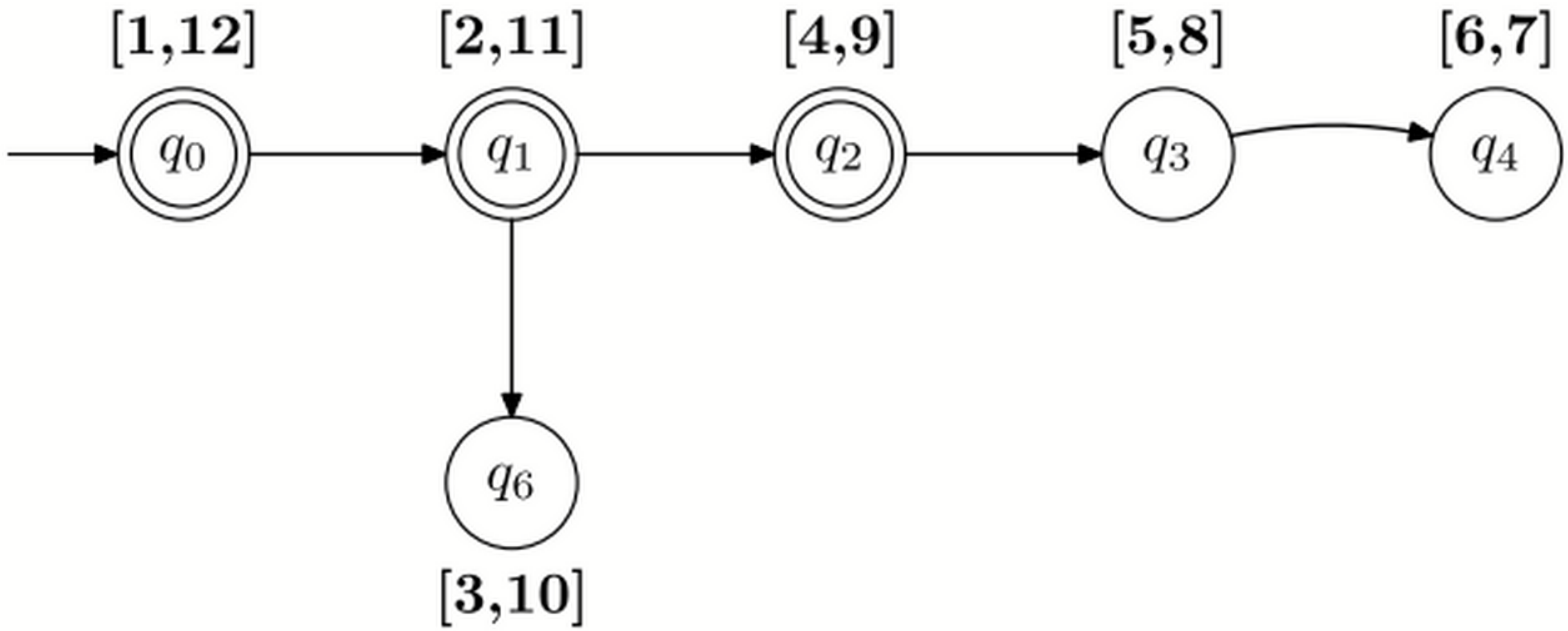
Initially, the worklist contains only the initial state.
At every iteration:

- choose state from the worklist and mark it as "discovered" (but don't remove it yet)
- If all successors have already been discovered, then remove the state from the worklist
- Otherwise, choose a not yet discovered successor and add it to the worklist

Depth-first-search: worklist implemented as a STACK
Breadth-first-search: worklist implemented as a QUEUE

Depth-first search





Some DFS-terminology

States are "discovered" by the search.
After recursively exploring all successors, the search "backtracks" from the state.
A state q is attached
- a "discovery time" d[q]
- a "finishing time" f[q].
- a "DFS-predecessor", the state from which it is discovered

Coloring scheme: At a given time moment a state is either
- white: not yet discovered [1, d[q]]
- grey: already discovered, successors not yet fully explored (d[q], f[q]]
- black: search has already backtracked from the state (f[q], 2n]

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

Theorem 13.1 (Parenthesis Theorem) *In a DFS-tree, for any two states q and r, exactly one of the following four conditions holds, where $I(q)$ denotes the interval $(\, d[q], f[q]\,]$, and $I(q) \prec I(r)$ denotes that $f[q] < d[r]$ holds.*

- *$I(q) \subseteq I(r)$ and q is a descendant of r, or*
- *$I(r) \subseteq I(q)$ and r is a descendant of q, or*
- *$I(q) \prec I(r)$, and neither q is a descendant of r, nor r is a descendant of q, or*
- *$I(r) \prec I(q)$, and neither q is a descendant of r, nor r is a descendant of q.*

Theorem 13.2 (White-path Theorem) *In a DFS-tree, r is a descendant of q (and so $I(r) \subseteq I(q)$) if and only if at time $d[q]$ state r can be reached from q in A along a path of white states.*

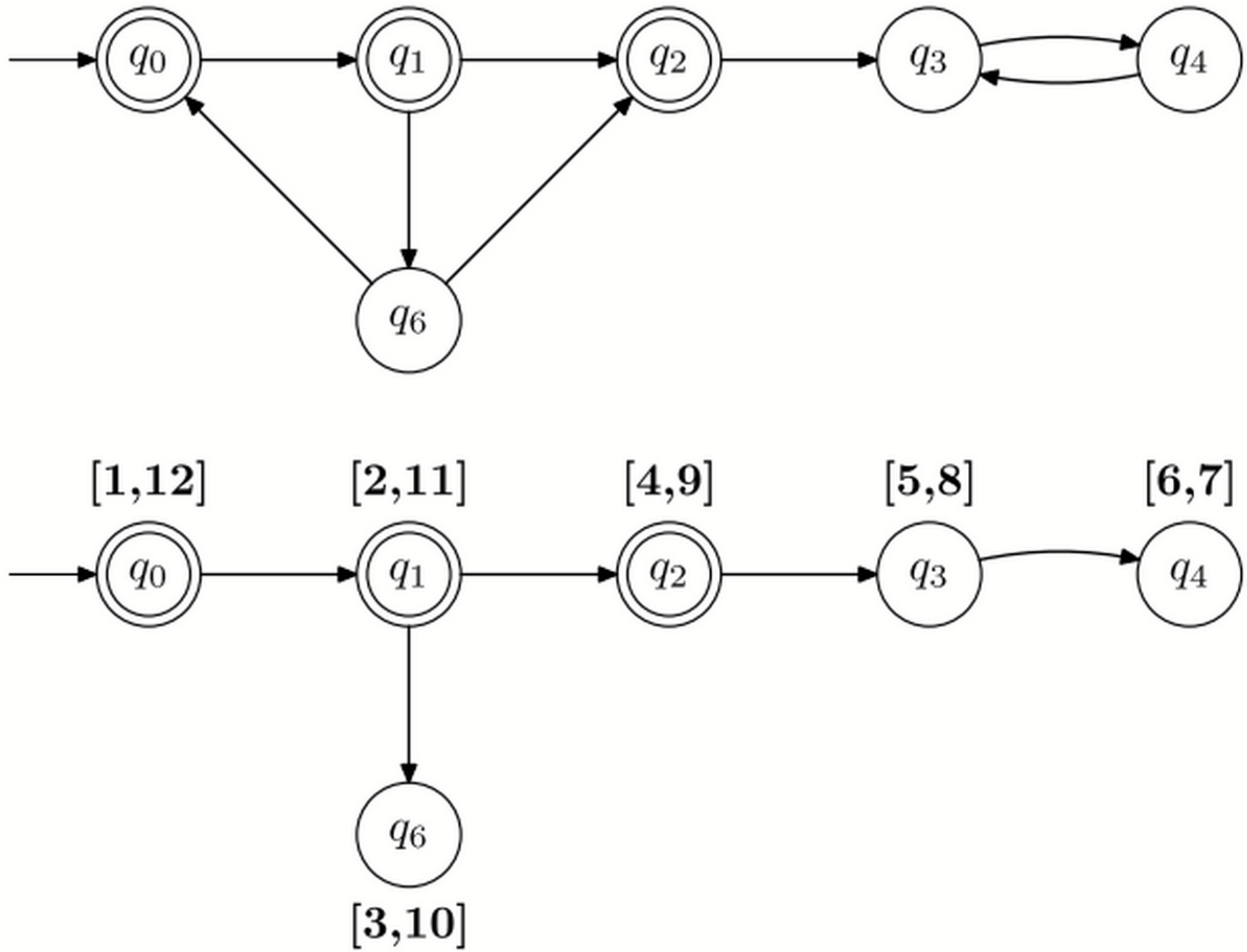
The nested DFS-algorithm

Modification of the naive algorithm:

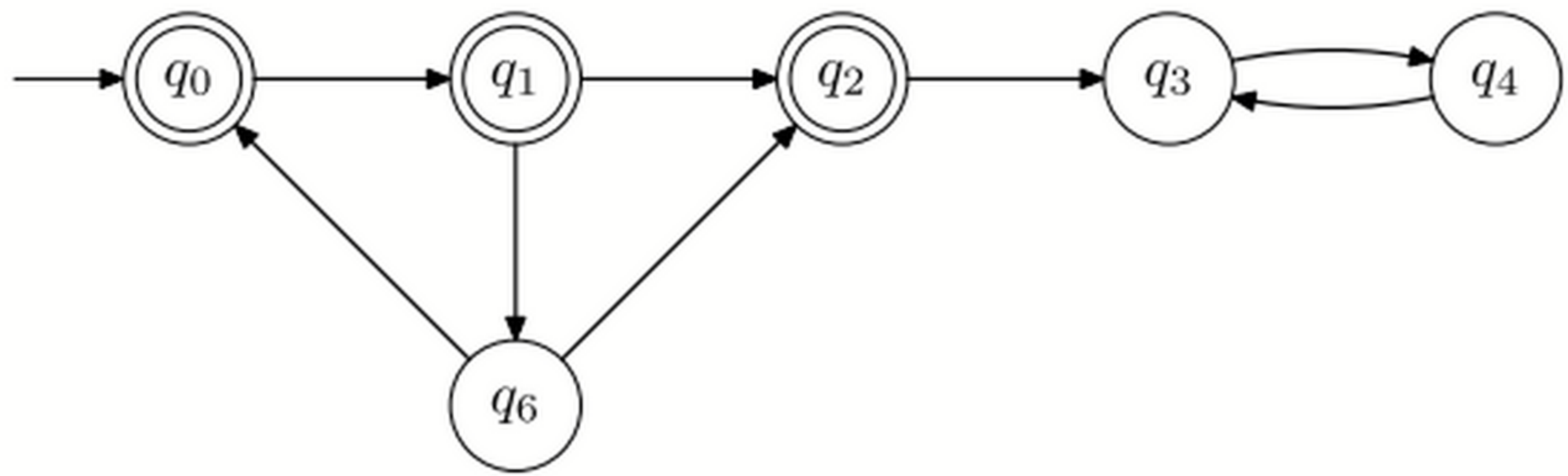
- use a DFS-search to discover the accepting states AND TO SORT THEM: q1, ..., qk
- conduct a DFS-search from each accepting state IN THE ORDER q1, ... , qk
  The order guarantees (proof required!) that if the search from qj hits a state already discovered in the search from qi (where i < j), then the search can backtrack from there.

Runtime: O(n+m)

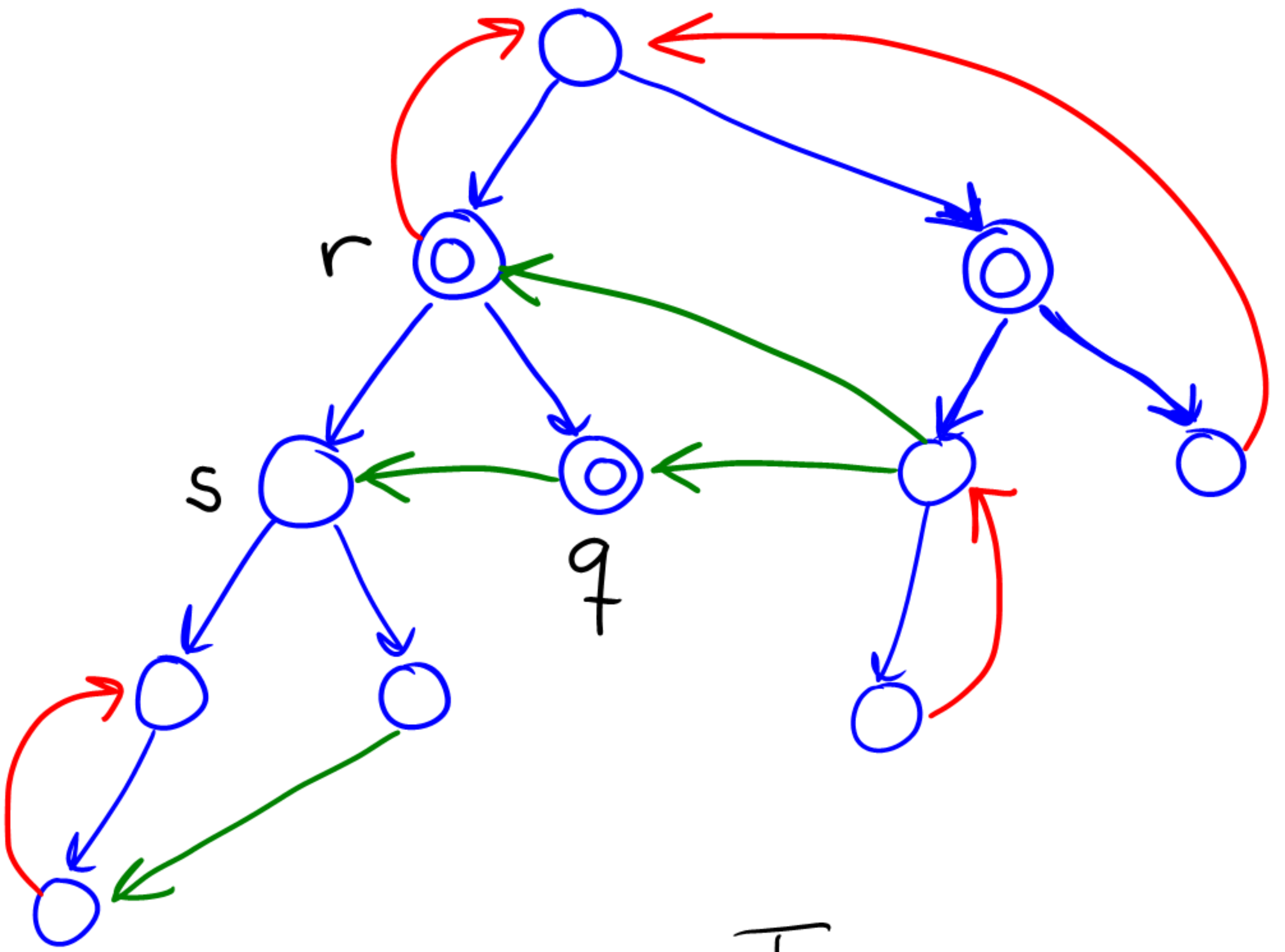
The order is the POSTORDER: accepting states are sorted according to their FINISHING TIME.



Example



Why does it work? (intuition)



To prove: $s \not\to r$

The proof

Lemma 13.3 *If $q \rightsquigarrow r$ and $f[q] < f[r]$ in some DFS-tree, then some cycle of A contains q.*

Proof: Let π be a path leading from q to r, and let s be the first node of π that is discovered by the DFS. By definition we have $d[s] \leq d[q]$. We prove that $s \neq q$, $q \rightsquigarrow s$ and $s \rightsquigarrow q$ hold, which implies that some cycle of A contains q, and $d[s] < d[q]$.

- $q \neq s$. If $s = q$, then at time $d[q]$ the path π is white, and so $I(r) \subseteq I(q)$, contradicting $f[q] < f[r]$.
- $q \rightsquigarrow s$. Obvious, because s belongs to π.
- $s \rightsquigarrow q$. By the definition of s, and since $s \neq q$, we have $d[s] \leq d[q]$. So either $I(q) \subseteq I(s)$ or $I(s) \prec I(q)$. We claim that $I(s) \prec I(q)$ is not possible. Since at time $d[s]$ the subpath of π leading from s to r is white, we have $I(r) \subseteq I(s)$. But $I(r) \subseteq I(s)$ and $I(s) \prec I(q)$ contradict $f[q] < f[r]$, which proves the claim. Since $I(s) \prec I(q)$ is not possible, we have $I(q) \subseteq I(s)$, and hence q is a descendant of s, which implies $s \rightsquigarrow q$.

Assume:

- q and r are acepting states
- f[q] < f[r]
- the search from q has finished without success
- the search from r has started, and has just discovered

  a state s that was already discovered in the search from q.

Then: $s \not\rightsquigarrow r$

Assume $s \rightsquigarrow r$ / $q \rightsquigarrow r$. Then we have $q \rightsquigarrow s$, $s \rightsquigarrow r$ and so $q \rightsquigarrow r$

By the Lemma some cycle contains q. But this contradicts that the search from q was unsuccessful.

Nesting the searches

The algorithm does not allow to "stop early". All states and transitions has to be examind at least once

If the NBA is nonempty, then in order to return a omega-word accepted by the automaton we have to use a lot of memory.

Better: nest the two searches.

- Perform a DFS from q_0.
- Whenever the search blackens an accepting state q, launch a new DFS from q. If this second DFS visits q again (i.e., if it explores some transition leading to q), stop with NONEMPTY. Otherwise, when the second DFS terminates, continue with the first DFS.
- If the first DFS terminates, output EMPTY.

NestedDFS(A)

Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$

Output: EMP if $\mathcal{L}_\omega(A) = \emptyset$
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 $\mathcal{L}_\omega(A) = \emptyset$
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] ∉ Q 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 succ = true then return [q, 2]
        if r = seed then
            report NEMP; succ ← true
    return
```

Evaluation

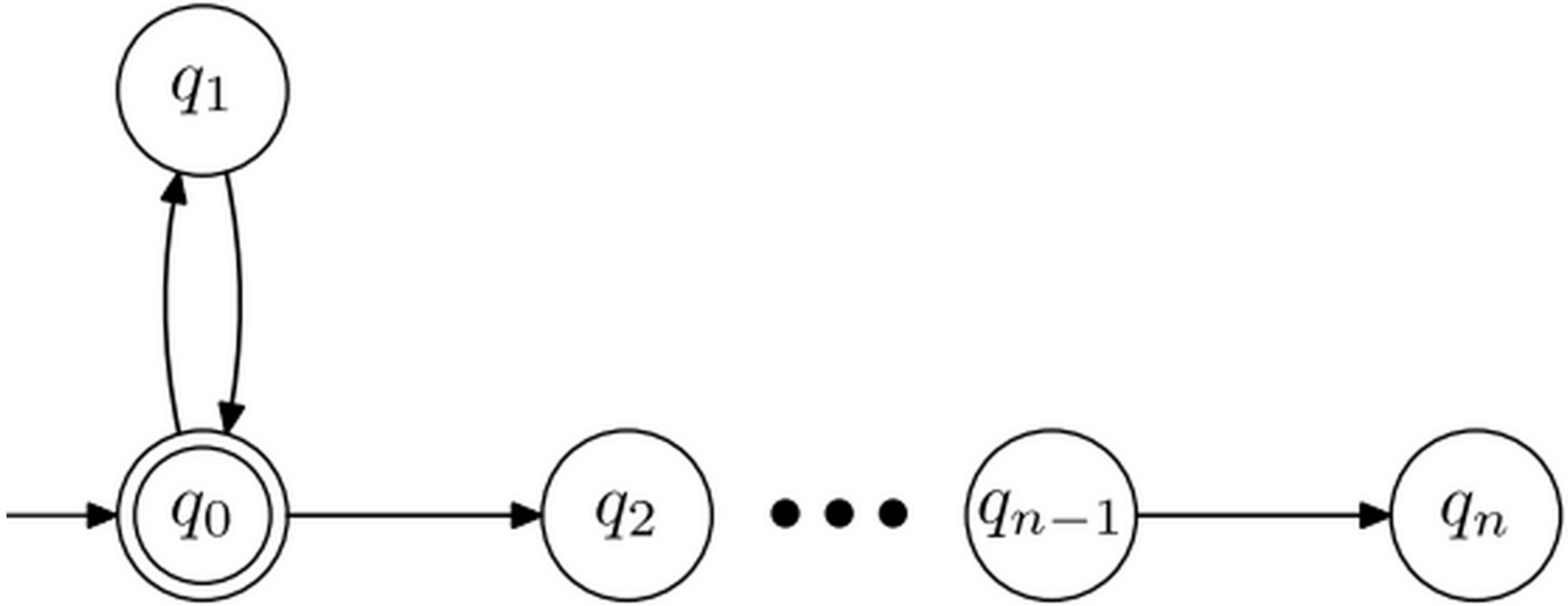
Positive points:

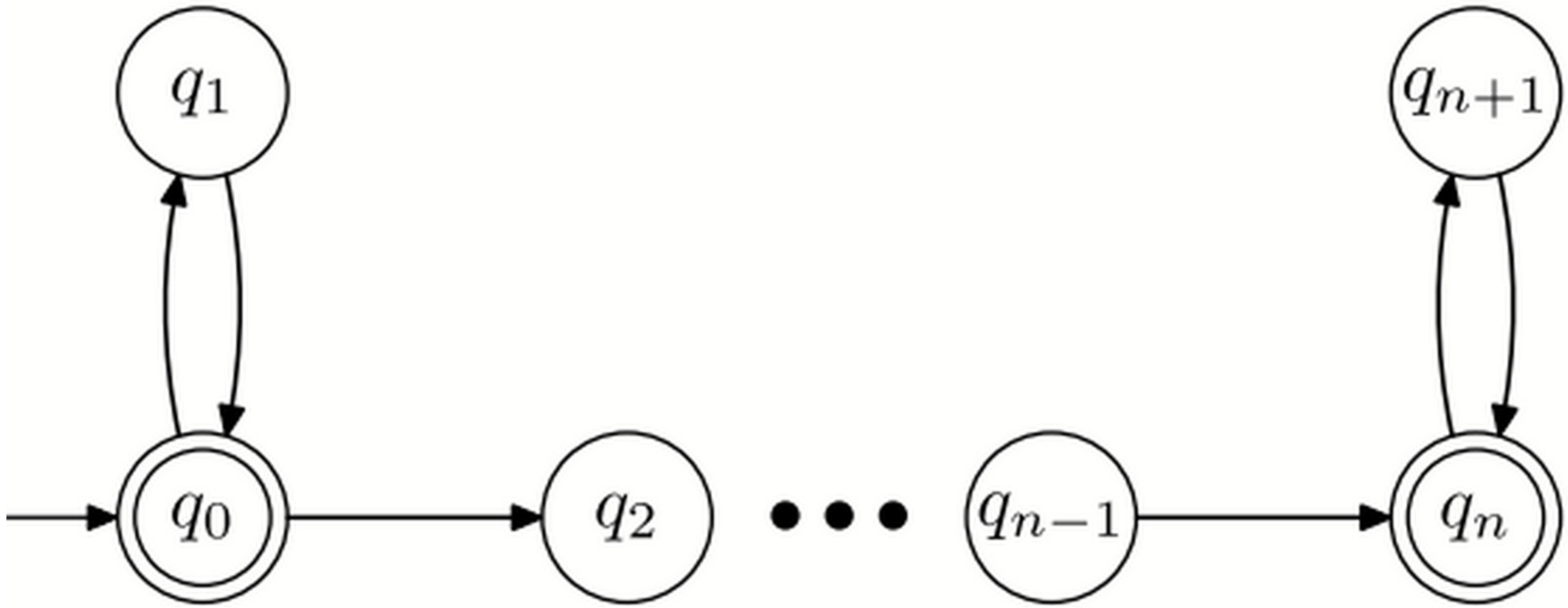
- Very low memory consumption: two (or even one) extra bit pro state.
- Easy to understand and prove correct.

Negative points:

- Cannot be generalized to NGAs.
- It is not OPTIMAL, and may return unnecessarily long witnesses for nonemptiness.

An algorithm is optimal if it answers "nonempty" after exploring a part of the NBA containing an accepting lasso, and before exploring any further.


q_1
q_0
q_2
q_{n-1}
q_n


q_1
q_{n+1}
q_0
q_2
q_{n-1}
q_n

Two generic approaches

1. Compute the set of accepting states.
   For every accepting state, check if it belongs to a cycle.

   Nested-depth-first algorithm

2. Compute the set of states that belong to a cycle
   For each of them, check if it is accepting.

   Two-stack algorithm

The second approach

Naive algorithm: conduct a DFS, and for every discovered state start a new DFS to check if it belongs to a cycle.
Again: far two expensive.

Goal: conduct ONE SINGLE DFS which has the possibility to mark 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 ...

When the DFS blackens a state, it has enough information to decide if the state belongs to a cycle or not.

Lemma 13.6 *Let A_t be the sub-NBA of A containing the states and transitions explored by the DFS up to (and including) time t. If a state q belongs to some cycle of A, then it already belongs to some cycle of $A_{f[q]}$.*

Proof: Let π be a cycle containing q, and consider the snapshot of the DFS at time $f[q]$. Let r be the last state of π after q that is black, i.e., the last state r, starting at q, such that $f[r] \leq f[q]$. If $r = q$, then π is a cycle of $A_{f[q]}$, and we are done. If $r \neq q$, let s be the successor of r in π (see Figure 13.4). We have $f[r] < f[q] < f[s]$. Moreover, since all successors of r have necessarily been discovered at time $f[r]$, we have $d[s] < f[r] < f[q] < f[s]$. By the Parenthesis theorem, s is a DFS-ascendant of q. Let π' be the cycle obtained by concatenating the DFS-path from s to q, the prefix of π from q to r, and the transition (r, s). By the Parenthesis Theorem, all the transitions in this path have been explored at time $f[q]$, and so the cycle belongs to $A_{f[q]}$ □

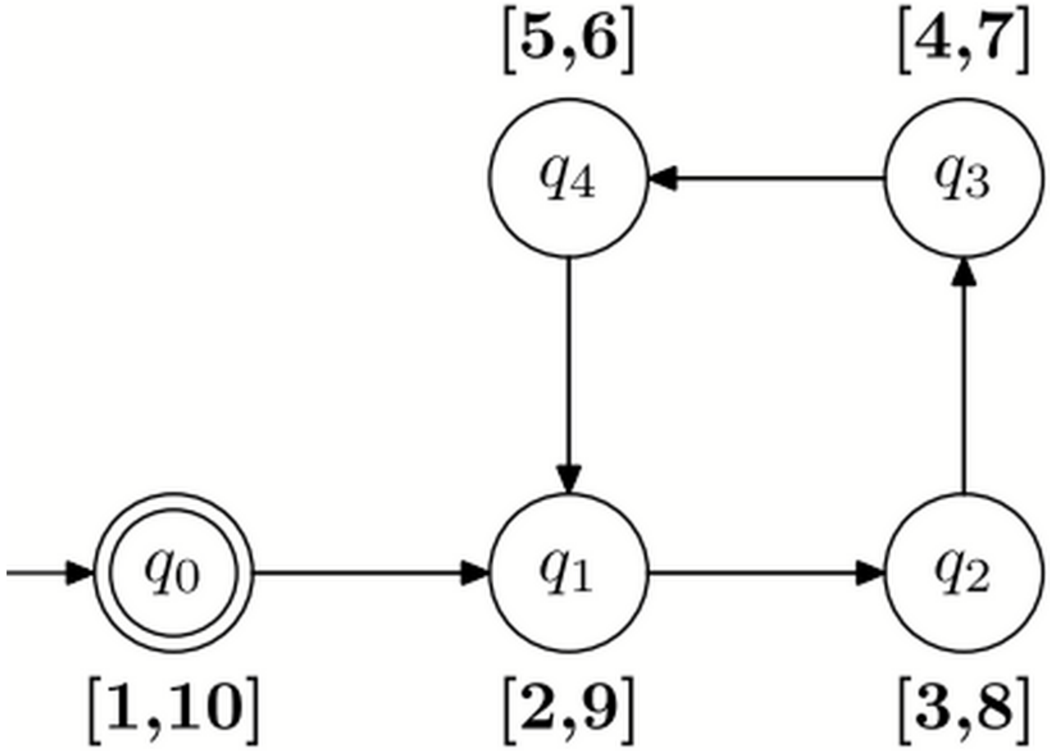
First ideas

Maintain a set C of "candidates", states for which the search cannot yet decide if they belong to a cycle or not.

- add a state to the set when grayed
- remove a state from the set when blackened, or before.

How to update C when a transition (q, r) is explored?

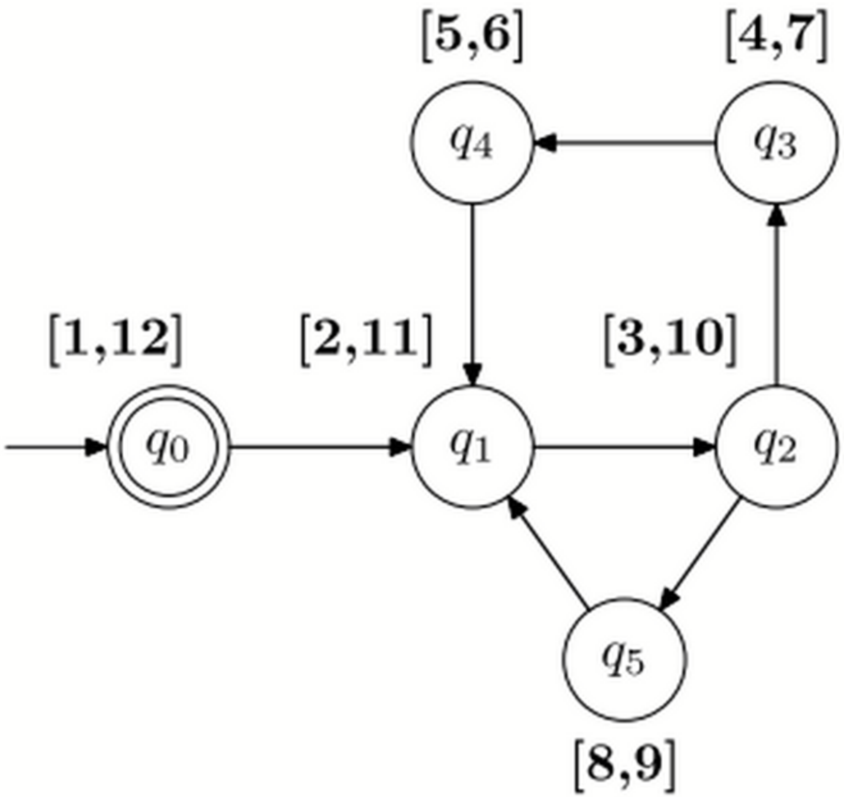
- If r is a new state (discovery), just add r to C
- If r has already been discovered, but q is not reachable from q, then do nothing
- If r has already been discovered, and q is reachable from r, then new cycles have been created ... Which states have to be removed from C ?

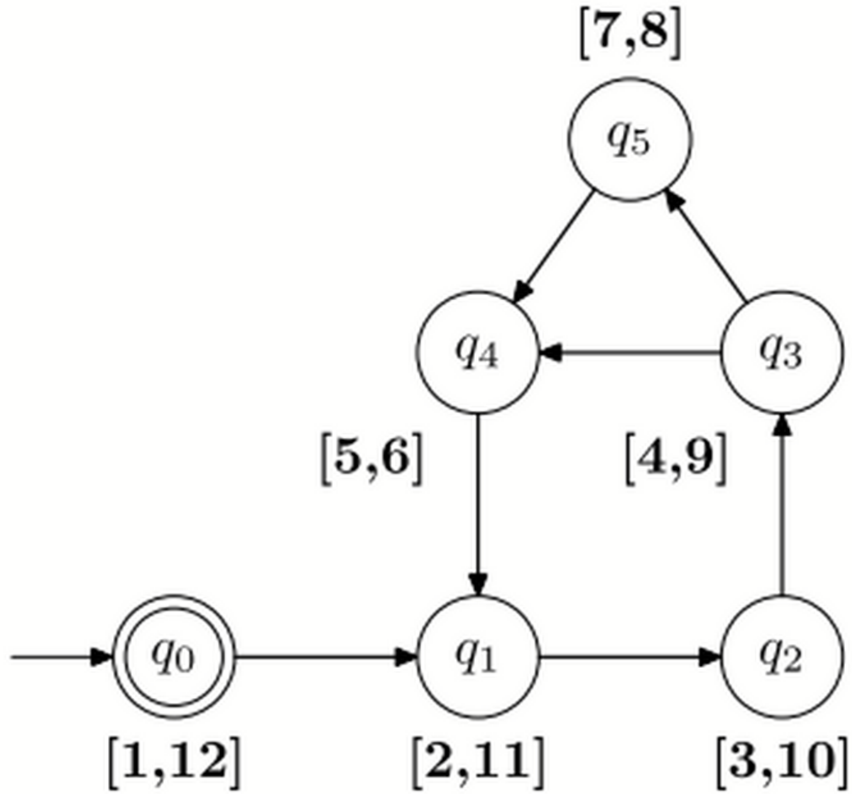


After exploring (q4, q1), we have to remove q1, ..., q4.
This suggests implementing C as a stack.

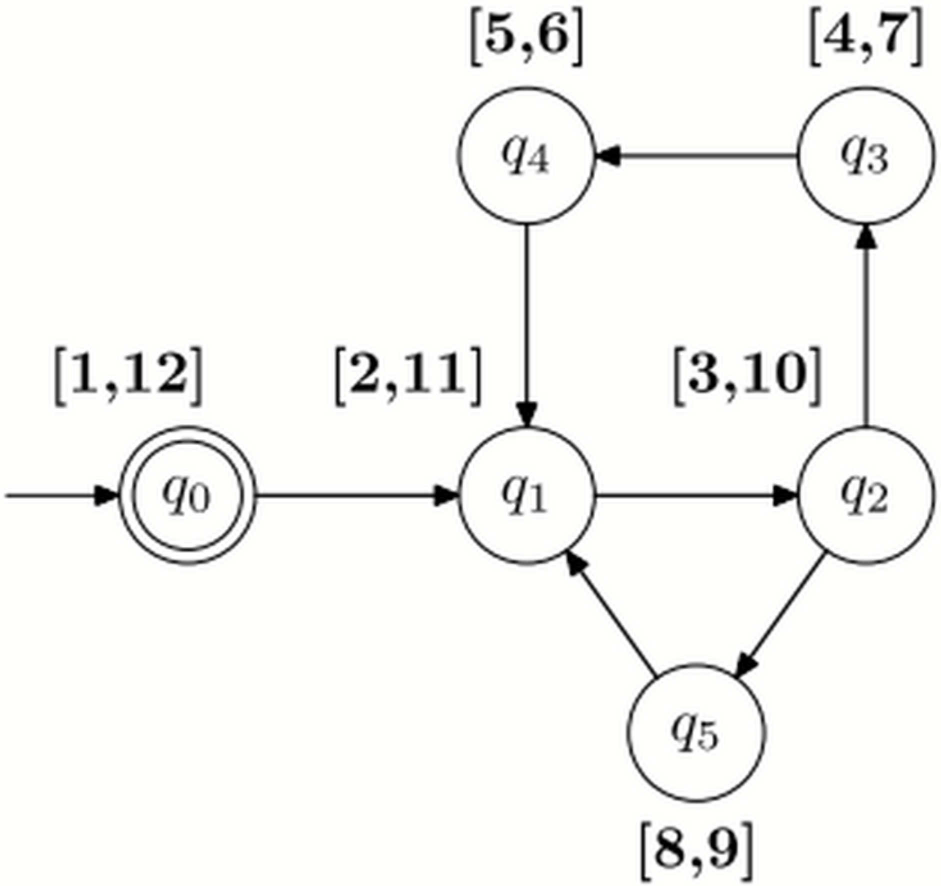
First naive idea: push when discovered,
when (q,r) explored and r seen before, pop
until r is popped.

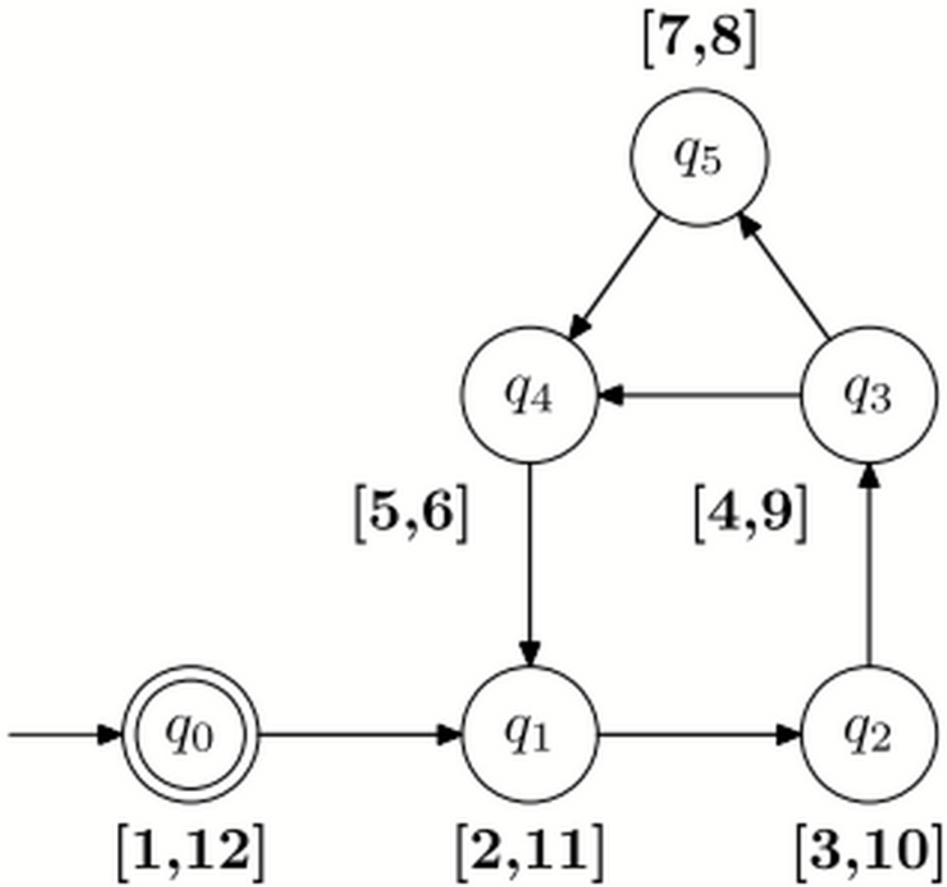
Problems


[5,6]
[4,7]
q_4
q_3
[1,12]
[2,11]
[3,10]
q_0
q_1
q_2
q_5
[8,9]


[7,8]
q_5
q_4
q_3
[5,6]
[4,9]
q_0
q_1
q_2
[1,12]
[2,11]
[3,10]

Problems


[5,6]
[4,7]
q_4
q_3
[1,12]
[2,11]
[3,10]
q_0
q_1
q_2
q_5
[8,9]


[7,8]
q_5
q_4
q_3
[5,6]
[4,9]
q_0
q_1
q_2
[1,12]
[2,11]
[3,10]

New attempt.

If (q, r) is being explored, r has already been discovered, and q is reachable form r , then:

- Pop until r or some state discovered before r is popped, and then push this state back.
- Pop when blackened.

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

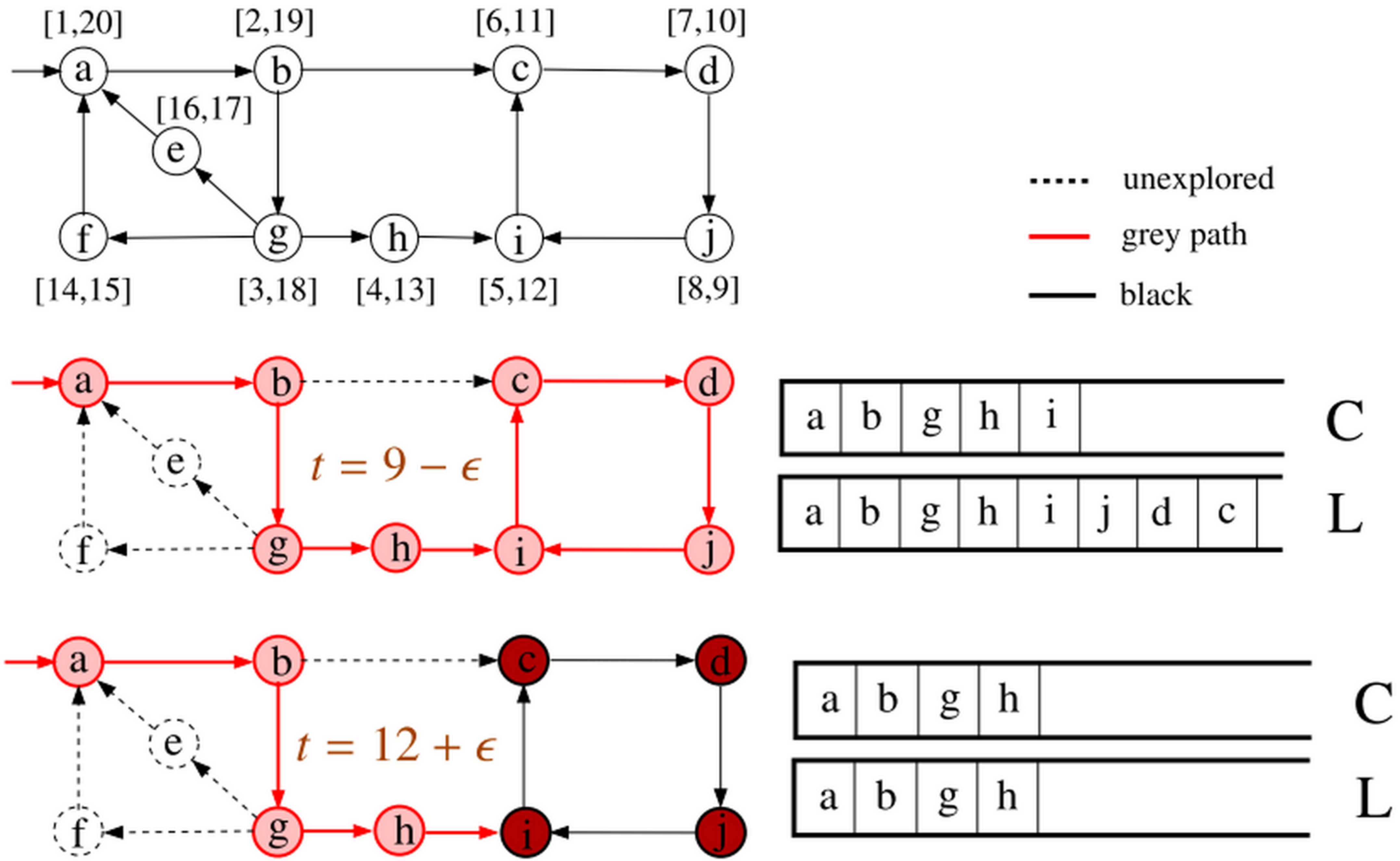
OneStack(A)

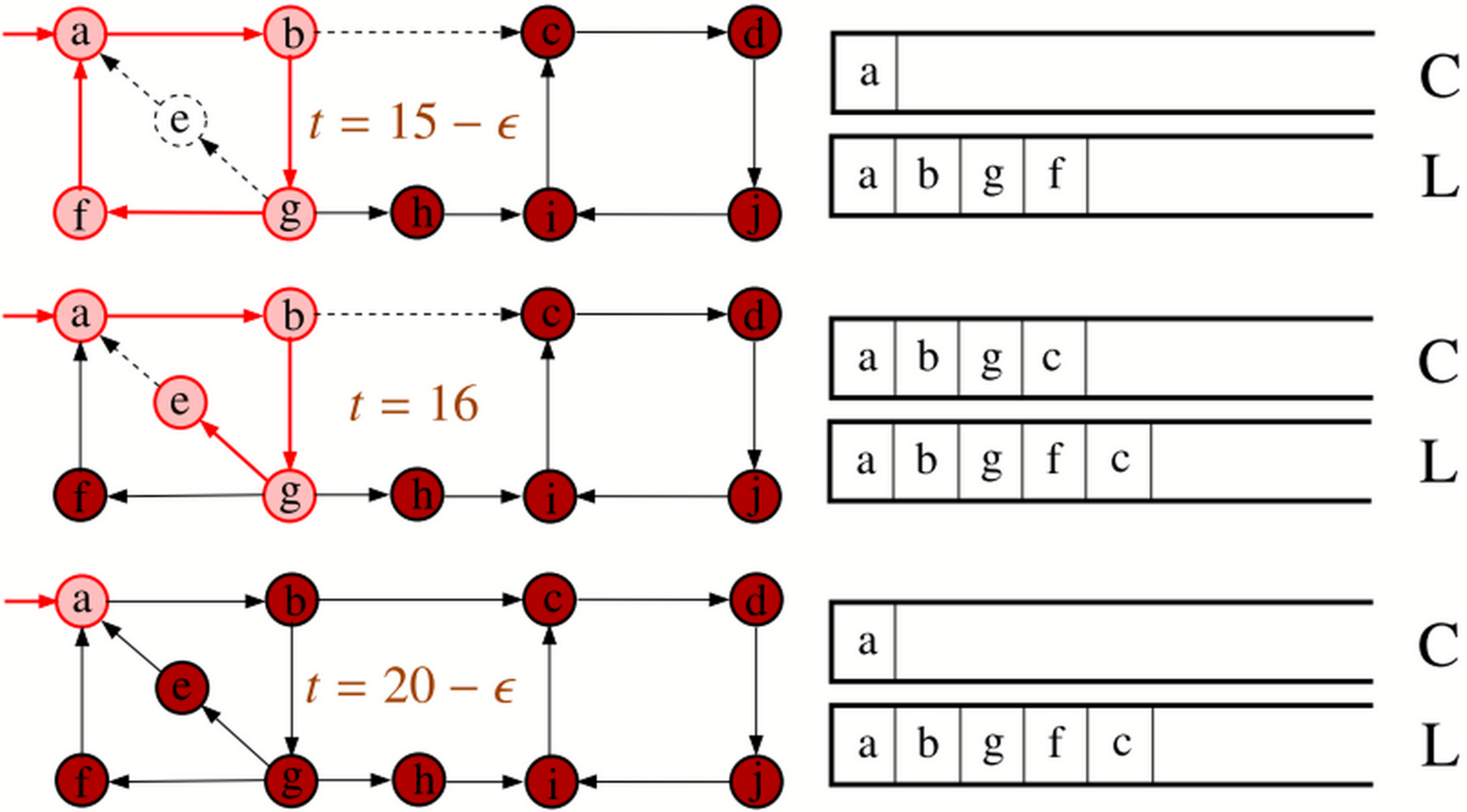
Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$

Output: EMP if $\mathcal{L}_\omega(A) = \emptyset$, NEMP otherwise

```
S, C ← ∅;
dfs(q0)
report EMP

dfs(q)
    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)
```


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


t = 15 − ϵ
a b c d e f g h i j
C: a
L: a b g f
t = 16
C: a b g c
L: a b g f c
t = 20 − ϵ
C: a
L: a b g f c

Questions about OneStack

Is it correct?

Proof obligations:
(1) Every node belonging to a cycle is eventually popped.
(2) Every node that is popped belongs to a cycle.

Is it optimal?

How do we implement the oracle ?

Proposition 13.8 *If q belongs to a cycle, then q is eventually popped by the repeat loop.*

Proof: Let π be a cycle containing q, let q' be the last successor of q along π such that at time $d[q]$ there is a white path from q to q', and let r be the successor of q' in π. Since r is grey 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 the transition (q', r) is explored before q is blackened. So when (q', r) is explored, q has not been popped at line 13. Since $r \leadsto q'$, either q has already been popped by at some former execution of teh repeat loop, or it is popped now, because $d[r] \leq d[q']$. □

Proposition 13.8 *If q belongs to a cycle, then q is eventually popped by the repeat loop.*

Proof: Let π be a cycle containing q, let q' be the last successor of q along π such that at time $d[q]$ there is a white path from q to q', and let r be the successor of q' in π. Since r is grey 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 the transition (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 by at some former execution of teh repeat loop, or it is popped now, because $d[r] \leq d[q']$. □

Proves optimality!

For the other direction (every popped node belongs to a cycle), we need some concepts:

- Strongly connected component (scc)
- Dag of strongly connected components
- Root of an scc in a DFS

Invariant of OneStack:
The repeat loop cannot remove a grey root from the stack. (remove: pop and do not push back), and, it only pops nodes with larger or equal discovery time.

Proof (sketch):
Take time at which repeat loop is executed because r ~> q for some r, q, and take root rt grey at this time.

r,q belong to the same scc. Let rt' be root of scc of r and q. Then rt' is also grey. So either rt ~> rt' or rt'~> rt, but if rt'~>rt then rt is not root. So rt ~> rt', which implies d[rt] =< d[r] =< d[s].

In particular, if rt is popped by the loop, then after the loop it is pushed back again.

Prop: Any state s popped at the repeat loop belongs to a cycle.

Prof (sketch). Assume loop execution for r, q. We have r ~> q, and r,q belong to an scc with root rt.

We show:

1) s is an ascendant of q.

   Both s and q are currently grey and beong to the grey path, and since dfs(q) is being executed it is the last state of the path.

2) rt is an ascendant of s.

   rt is ascendant of q (White-path). By 1) either rt is ascendant of s or viceversa. By the invariant we have d[rt] =< d[s], and so rt is ascendant of s.

By 1) and 2) we have rt~>s~>q~>r~>rt, and we are done.

Implementing the oracle

Lemma 13.12 *Assume that OneStack(A) is currently exploring a transition (q, r), and the state r has already been discovered. Let R be the scc of A satisfying $r \in R$. Then $r \leadsto q$ iff some state of R is not black.*

Proof: Assume $r \leadsto q$. Then r and q belong to R, and since q is not black because (q, r) is being explored, R is not black.

Assume $r \not\leadsto q$. We consider the colors of the states at the time (q, r) is explored, and show that all the states of R are black. We proceed by contradiction. Assume some state of R is not black. Not all states of R are white because r has already been discovered, and so at least one state $s \in R$ is grey. Since grey states form a path ending at the state whose output transitions are being currently explored, the grey path contains s and ends at q. So $s \leadsto q$, and, since s and r belong to R, we have $r \leadsto q$, contradicting the hypothesis. □

Idea: maintain a set V of "active" states, states whose sccs have not yet been completely explored.

Notice: the root is the first state of an scc to be grey, and the last to be blackened. So we can proceed as follows:

- States are added to V when they are discovered.
- States are removed from V when their root is blackened.

So V can be implemented as stack: when root is popped from the stack of candidates, we pop from V until we hit the root.

Problem: when blackening a node, decide if it is a root!!

Lemma 13.14 *When OneStack executes line 13, q is a root if and only if* **top**$(C) = q$.

Proof: Assume q is a root. By Lemma 13.10, q still belongs to C after the for loop at lines 6-12 is executed, and so **top**$(C) = q$ at line 13.

Assume now that q is not a root. Then there is a path from q to the root ρ of q's scc. Let r be the first state in the path satisfying $d[r] < d[q]$, and let q' be the predecessor of r in the path. By the White-path theorem, q' is a descendant of q, and so when transition (q, r) is explored, q is not yet black. When *OneStack* explores (q', r), it pops all states s from C satisfying $d[s] > d[r]$, and none of these states is pushed back at line 12. In particular, either *OneStack* has already removed q from C, or it removes it now. Since q has not been blackened yet, when *OneStack* executes line 14 for $dfs(q)$, the state q does not belong to C and in particular $q \neq$ **top**(C) □

TwoStack(A)

Input: NBA $A = (Q, \Sigma, \delta, q_0, F)$

Output: EMP if $\mathcal{L}_\omega(A) = \emptyset$, NEMP otherwise

```
S, C, V ← ∅;
dfs(q0)
report EMP

proc dfs(q)
    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 $\mathcal{L}_\omega(A) = \emptyset$, NEMP otherwise

```
S, C, V ← ∅;
dfs(q0)
report EMP

proc dfs(q)
    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
```