Contents

Chapter 9

Response Verification

9.1 Response Requirements

9.1.1 The fair emptiness problem

The basic problem in the analysis of transition graphs is to determine whether some state in a target region is reachable. The corresponding basic problem in the analysis of fair graphs is to determine whether a fair graph has an initialized fair trajectory.

> FAIR EMPTINESS PROBLEM
>
> An instance of the fair-emptiness problem is a fair graph $\mathcal{G}$. The answer to the fair-emptiness problem $\mathcal{G}$ is YES if the ω-language $\mathcal{L}_{\mathcal{G}}$ is nonempty, and NO otherwise.

Remark 9.1 [Fair emptiness for machine-closed graphs] If $\mathcal{G}$ is a machine-closed fair graph then the answer to the fair-emptiness problem $\mathcal{G}$ is YES. ■

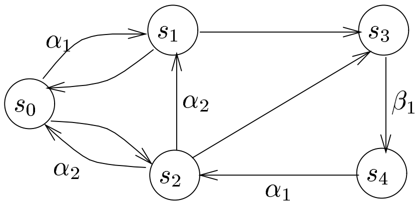

Figure 9.1: Fair Cycles

The fair cycle problem

To establish that the ω-language of a fair graph is empty, we need to find an initialized fair trajectory. Our algorithms will search for initialized fair trajectories of a special form, namely, the eventually periodic ones. This is because eventually periodic trajectories can be represented in a finite manner using cycles.

A *cycle* of the transition graph G is a trajectory $\overline{s}_{0..m}$ such that $s_m = s_0$. Fairness of a cycle with respect to an action, fairness constraint, and a fairness assumption is defined by considering all the actions involved in the cycle.

> FAIR CYCLE
>
> Let G be a transition graph. The cycle $\overline{s}_{0..m}$ of G is *α-fair*, for an action α of G, if $s_i \xrightarrow{\alpha} s_{i+1}$ for some $0 \le i < m$. The cycle $\overline{s}_{0..m}$ is f-fair for a fairness constraint $f = (\alpha, \beta)$ of G, if it is either β-fair or not α-fair. The cycle $\overline{s}_{0..m}$ is F-fair, for a fairness assumption F of G, if it is f-fair for all fairness constraints f in F.
>
> A *fair cycle* of the fair graph (G, F) is a F-fair cycle of G.

Remark 9.2 [Fair trajectory of a fair cycle] If $\overline{s}_{0...m}$ is a fair cycle of the fair graph $\mathcal{G}$, then the periodic ω-trajectory $(\overline{s}_{0...m-1})^\omega$ is a fair trajectory of $\mathcal{G}$. ■

Example 9.1 [Fair cycles] Consider the fair graph of Figure 9.1 with $F = \{(\alpha_1, \beta_1), (\alpha_2, \beta_2)\}$. The cycle $s_0 s_1 s_3 s_4 s_2 s_0$ is not F-fair. Similarly, the cycle $s_1 s_3 s_4 s_2 s_1$ is not F-fair. On the other hand, the cycle $s_2 s_3 s_4 s_2$ is F-fair. ■

To solve the fair-emptiness problem, we look for fair cycles that are reachable. The cycle $\overline{s}_{0..m}$ of a transition graph G is reachable if the state s_0 is reachable, or equivalently, each s_i, for $0 \le i \le m$, belongs to the reachable region σ^R of G.

FAIR CYCLE PROBLEM

An instance of the fair-cycle problem is a fair graph $\mathcal{G}$. The answer to the fair-cycle problem is YES if there exists a reachable fair cycle of $\mathcal{G}$, and NO otherwise. A *witness* for the fair-cycle problem $\mathcal{G}$ consists of (1) an initialized trajectory $\overline{s}_{0...m}$ of $\mathcal{G}$, and (2) a source-s_m fair cycle $\overline{t}_{0...k}$ of $\mathcal{G}$.

Consider a fair graph $\mathcal{G}$. Suppose the answer to fair-cycle problem is YES, and let $(\overline{s}_{0...m}, \overline{t}_{0...k})$ be a witness to the fair-cycle problem. Then, the eventually periodic ω-trajectory $\overline{s}_{0...m-1}(\overline{t}_{0...k-1})^{\omega}$ is an initialized fair trajectory of $\mathcal{G}$, and thus, the answer to the fair-emptiness problem $\mathcal{G}$ is YES. Conversely, existence of an initialized fair trajectory guarantees the existence of a reachable fair cycle, provided that the graph is finite.

Proposition 9.1 [Fair cycle vs. fair trajectory] *Let $\mathcal{G}$ be a finite fair graph, and let s be a state of $\mathcal{G}$. Then, there exists a source-s fair cycle of $\mathcal{G}$ iff there exists a source-s fair trajectory of $\mathcal{G}$.*

Exercise 9.1 {T2} [Fair cycle vs. fair trajectory] (1) Prove Proposition 9.1. (2) Show that Proposition 9.1 does not hold for infinite fair graphs. ■

Corollary 9.1 [Fair emptiness vs. fair cycle problems] *For a finite fair graph $\mathcal{G}$, the answer to the fair-emptiness problem $\mathcal{G}$ coincides with the answer to the fair-cycle problem $\mathcal{G}$.*

9.1.2 The recurrence verification problem

The basic problem in the analysis of modules is invariant verification which asks whether a given predicate is an invariant of the module. The basic problem in the analysis of fair modules is recurrence verification.

RECURRENT

Let $\mathcal{P}$ be a fair module, and let p be an observation predicate for $\mathcal{P}$. The predicate p is a *recurrent* of $\mathcal{P}$ if every fair ω-trajectory of $\mathcal{P}$ is p-fair

In other words, given fair module $\mathcal{P}$ with fairness assumption F, the observation predicate p is a recurrent of $\mathcal{P}$ iff every fair trajectory of $\mathcal{P}$ is guaranteed to contain infinitely many p-states: for every F-fair ω-trajectory $\underline{s}$, for infinitely many positions i, the state s_i satisfies the observation predicate p.

RECURRENCE-VERIFICATION PROBLEM

An instance $(\mathcal{P}, p)$ of the *recurrence-verification problem* consists of (1) a fair module $\mathcal{P}$ and (2) an observation predicate p for $\mathcal{P}$. The answer to the recurrence-verification problem is YES if p is a recurrent of $\mathcal{P}$, and otherwise NO.

Example 9.2 [Starvation freedom of mutual exclusion] Let us revisit the mutual exclusion problem from Chapter 1. A liveness requirement for the mutual exclusion algorithms is the starvation freedom property which asserts that if a process requests the critical section, then that process eventually enters the critical section. Requiring starvation freedom rules out solutions that always prefer one process over the other. Starvation freedom for process i corresponds to checking whether the predicate $pc_i \neq reqC$ is recurrent. Verify that the answers to both the recurrence-verification problems $(SyncMutex, pc_1 \neq reqC)$ and $(FairPete, pc_2 \neq reqC)$ are YES. ■

Exercise 9.2 {P2} [Starvation freedom for railroad controller] Recall the railroad example from Chapter 2. Show that the requirement that a train does not wait at the signal forever can be formulated as a recurrence-verification problem. Does *RailroadSystem* satisfy the requirement? ■

From recurrence verification to fair emptiness

The answer to the recurrence-verification problem $(\mathcal{P}, p)$ is NO when there is a fair trajectory of $\mathcal{P}$ that is not $[\![p]\!]$-fair. Observe that an ω-trajectory is not σ-fair, for a region σ, iff it is $(\sigma, \emptyset)$-fair. Consequently, to solve recurrence verification problem $(\mathcal{P}, p)$ we consider the fair graph $\mathcal{G}_{\mathcal{P}}$, add the region-constraint $([\![p]\!], \emptyset)$, and check whether the resulting fair graph has an initialized fair trajectory. For an instance $(\mathcal{P}, p)$ of the recurrence-verification problem, the fair graph $(G_{\mathcal{P}}, F \cup \{([\![p]\!], \emptyset)\}$ is denoted $\mathcal{G}_{\mathcal{P},p}$.

Proposition 9.2 [Recurrence-verification to fair emptiness] *The answer to the recurrence-verification problem $(\mathcal{P}, p)$ is* YES *iff the answer to the fair emptiness problem $\mathcal{G}_{\mathcal{P},p}$ is* NO.

As discussed earlier, to solve a fair emptiness problem, we solve the corresponding fair cycle problem. Thus, to solve the recurrence-verification problem $(\mathcal{P}, p)$, we solve the fair-cycle problem $\mathcal{G}_{\mathcal{P},p}$ If the answer to fair cycle problem is YES, the answer to the recurrence-verification problem is NO; if the answer to the fair-cycle problem is YES, and the module is finite, then the answer to the recurrence-verification problem is YES.

When the answer to the fair-cycle problem $\mathcal{G}_{\mathcal{P},p}$ is YES, the corresponding witness can be reported as an *error trajectory* for the recurrence-verification problem. It consists of an initialized trajectory $\overline{s}_{0\ldots m}$ of $\mathcal{P}$, and a source-s_m fair cycle $\overline{t}_{0\ldots k}$ of $\mathcal{P}$ such that t_i does not satisfy p for all $0 \leq i \leq k$. The fair cycle $\overline{t}_{0\ldots k}$ corresponds to a loop in which all fairness requirements of $\mathcal{P}$ are satisfied, but the predicate p never holds. Thus, it corresponds to a "bad" cycle in the execution. The initialized trajectory $\overline{s}_{0\ldots m}$ is an evidence that the bad cycle is reachable. Together, they provide useful information for debugging.

Exercise 9.3 {T3} [Eventual Invariants] Let $\mathcal{P}$ be a fair module, and let p be an observation predicate for $\mathcal{P}$. The observation predicate is an *eventual invariant* of $\mathcal{P}$ if for every fair trajectory $\underline{s}$ of $\mathcal{P}$, for some $i \geq 0$, $s_i \models p$ for all $j \geq i$. Thus, p is an eventual invariant of $\mathcal{P}$ if every fair trajectory of $\mathcal{P}$ has a suffix all of whose states satisfy p. The eventual-invariant verification problem is to check, given a fair module $\mathcal{P}$ and an observation predicate p for $\mathcal{P}$, whether or not p is an eventual invariant of $\mathcal{P}$.

Show that the eventual-invariant verification problem $(\mathcal{P}, p)$ can be reduced to fair emptiness problem. ■

9.1.3 The response verification problem

An observation predicate p is a recurrent of a fair module $\mathcal{P}$ if every fair trajectory contains infinitely many p-states. A more general and common requirement is the response requirement that stipulates that every request be followed by an eventual response.

> RESPONSE
>
> Let $\mathcal{P}$ be a fair module, and let p and q be two observation predicates of $\mathcal{P}$. The predicate q is said to be a *response* to the predicate p in $\mathcal{P}$, denoted $p \leadsto_{\mathcal{P}} q$, if for every fair trajectory $\underline{s}$ of $\mathcal{P}$, for all $i \geq 0$, if $s_i \models p$ then for some $j \geq i$, $s_i \models q$.

Intuitively, if p denotes a request, and q denotes the fulfillment of the request, then "q is a response to p" corresponds to "every request is eventually fulfilled." In particular, every fair trajectory that contains infinitely many p-states must contain infinitely many q-states.

Remark 9.3 [Transitivity of response] Let $\mathcal{P} = (P, F)$ be a fair module, and let p, q, and r be observation predicates of $\mathcal{P}$. (1) If $p \leadsto_P q$ then $p \leadsto_{\mathcal{P}} q$. (2) If $p \rightarrow q$ is an invariant of P (i.e., for every reachable state s of P, if s satisfies p, then s satisfies q) then $p \leadsto_P q$. (3) If $p \leadsto_{\mathcal{P}} q$ and $q \leadsto_{\mathcal{P}} r$, then $p \leadsto_{\mathcal{P}} r$. ■

> RESPONSE VERIFICATION
>
> An instance $(\mathcal{P}, p, q)$ of the *response-verification problem* consists of (1) a fair module $\mathcal{P}$, (2) [request predicate] an observation predicate p for $\mathcal{P}$, and (3) [response predicate] an observation predicate q for $\mathcal{P}$. The answer to the response-verification problem $(\mathcal{P}, p, q)$ is YES if $p \leadsto_{\mathcal{P}} q$, and otherwise NO.

Remark 9.4 [Invariant verification as response verification] Let P be a module, and let p be an observation predicate of P. The predicate p is an invariant of the module P iff $\neg p \leadsto_P \mathit{false}$. Thus, the invariant verification problem (P, p) reduces to the response verification problem $(P, \neg p, \mathit{false})$. ■

Remark 9.5 [Recurrence verification as response verification] Let $\mathcal{P}$ be a fair module, and let p be an observation predicate of $\mathcal{P}$. The predicate p is a recurrent of $\mathcal{P}$ iff $true \leadsto_P p$. Thus, the recurrence-verification problem (P, p) reduces to the response verification problem $(P, true, p)$. ■

Example 9.3 [Mutual exclusion] Let us revisit the mutual exclusion problem. The observation predicate

$$p_{reqC} \;=\; (pc_1 = reqC) \vee (pc_2 = reqC)$$

characterizes the states in which some process is requesting the critical section. The observation predicate

$$p_{inC} \;=\; (pc_1 = inC) \vee (pc_2 = inC)$$

characterizes the states in which some process is inside the critical section. Checking absence of deadlocks corresponds to checking whether p_{inC} is a response to p_{reqC}. Verify that the answer to the response verification problem $(SyncMutex, p_{reqC}, p_{inC})$ is YES. The answer to the response verification problem $(Pete, p_{reqC}, p_{inC})$ is NO (why?). However, along every fair trajectory of *FairPete* a state satisfying p_{reqC} is eventually followed by a state satisfying p_{inC}, and thus, the answer to the response verification problem $(FairPete, p_{reqC}, p_{inC})$ is YES. Observe that requiring p_{inC} to be a response to p_{reqC} is a weaker requirement than requiring $pc_1 \neq reqC$ to be recurrent (why?). ■

Example 9.4 [Fair synchronous communication] Consider the fair module *FairSyncMsg* obtained by composing the fair receiver with *SyncSender* and hiding the variables used for communication (see Section 8.5). A response requirement for the module *FairSyncMsg* stipulates that every message produced by the sender is eventually consumed by the receiver. If v is a value of the type $\mathbb{M}$, then we can use $msg_P = v$ as the request predicate, and $msg_C = v$ as the response predicate. Verify that the answer to the response verification problem $(FairSyncMsg, msg_P = v, msg_C = v)$ is YES. ■

Exercise 9.4 {P2} [Dining philosophers] Recall the fair version of the dining philosophers problem from Chapter 8. Formulate the starvation freedom requirement for an individual philosopher as a response verification problem. ■

From response-verification to recurrence-verification

Response-verification problem can be reduced to a recurrence-verification problem by adding monitors. Consider the response-verification problem $(\mathcal{P}, p, q)$.

Consider the reactive module *ResponseMonitor*:

> **module** *ResponseMonitor* **is**
>   **external** p, q
>   **private** *alert* : $\mathbb{B}$
>   **atom controls** *alert* **reads** p, q
>   **init**
> $[\!]\ true \rightarrow alert' := 0$
>   **update**
> $[\!]\ alert = 0 \wedge p \wedge \neg q \rightarrow alert' := 1$
> $[\!]\ alert = 1 \wedge q \qquad\quad \rightarrow alert' := 0$

The monitor *ResponseMonitor* observes the behavior of the module $\mathcal{P}$ and updates its private state *alert* accordingly. In the description of *ResponseMonitor*, the declaration **external** p, q stands for **external** $x_1, x_2, \ldots x_k$, where $x_1, x_2, \ldots x_k$ are the variables appearing in the predicates p and q. The value of the private variable *alert* is initially 0. When a state that satisfies the request predicate p, but not the response predicate q, is encountered, the variable *alert* is updated to 1. Thus, $alert = 1$ indicates a situation in which the request has been issued, but the subsequent response has not been issued. Once a state satisfying the response predicate is encountered, the variable *alert* is reset to 0. The following proposition asserts that q is a response to p in $\mathcal{P}$ iff $alert = 0$ is a recurrent of the compound module $\mathcal{P} \,\|\, ResponseMonitor$.

Proposition 9.3 [From response-verification to recurrence-verification] *The answer to the response-verification problem* $(\mathcal{P}, p, q)$ *coincides with the answer to the recurrence-verification problem* $(\mathcal{P} \,\|\, ResponseMonitor, alert = 0)$.

Proof. Consider a ω-trajectory $\underline{s}$ of $\mathcal{P}$. Since the update of the state of *ResponseMonitor* is deterministic, and from the definition of the parallel composition, there exists a unique ω-trajectory $\underline{t}$ of $\mathcal{P} \,\|\, ResponseMonitor$ such that $s_i = X_{\mathcal{P}}[t_i]$ for all $i \geq 0$. Furthermore, $\underline{s}$ is a fair-trajectory of $\mathcal{P}$ iff $\underline{t}$ is a fair trajectory of $\mathcal{P} \,\|\, ResponseMonitor$.

From the initialization and update commands for the variable *alert*, for all $i \geq 0$, $alert[t_i] = 1$ iff there exists $j < i$ such that $s_j \models p$ and $s_k \not\models q$ for $i \leq k < j$. Consequently, every p-state is followed by an q-state along $\underline{s}$ (i.e. for all $i \geq 0$, if $s_i \models p$ then there some $j \geq i$, $s_j \models q$) iff for infinitely many indices $i \geq 0$, $alert[t_i] = 0$. Consequently, q is a response to p in $\mathcal{P}$ iff every fair trajectory of $\mathcal{P} \,\|\, ResponseMonitor$ is $(alert = 0)$-fair. ■

Exercise 9.5 {T2} [Response-verification for action predicates] In our formulation of the response-verification problem, the request and the response predicates define regions. Formulate a variant of the problem in which the request and the response predicates for a module P are predicates over $\mathsf{obs}X_P \cup \mathsf{obs}X'_P$, and

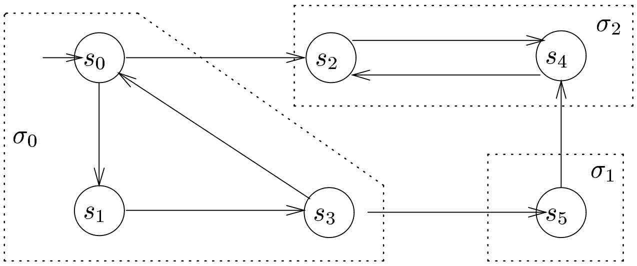

Figure 9.2: Strongly connected components

define actions of P. Show that this variant of the response-verification problem can be reduced to a recurrence-verification problem by adding an appropriate monitor. ■

9.2 Enumerative Search

In this section, we present a solution to the fair cycle problem based on classical enumerative graph-search algorithms. A naive way to solve the fair-cycle problem (G, F) would be to find all reachable cycles in the graph G, and check if any of them is F-fair. This is not efficient, since the number of cycles in a graph can be exponential in the number of its states. A more reasonable approach is to consider cycles of a special form, namely, the strongly connected components of the graph.

9.2.1 Strongly connected regions

Two states of a transition graph are strongly connected if there is a cycle that contains both of them. A strongly connected component is a maximal set of states that are strongly connected to one another.

> STRONGLY CONNECTED COMPONENT
>
> Let G be a transition graph. Two states s and t of G are *strongly connected*, written $s \cong_{scc} t$, iff $s \in post^*(t)$ and $t \in post^*(s)$. A region σ of G is *strongly connected* if $s \cong_{scc} t$ for all states s and t in σ. A region σ of G is a *strongly connected component* if (1) σ is strongly connected, and (2) [maximality] no strongly connected region of G is a strict superset of σ.

Remark 9.6 [Strongly connected components] The relation $\cong_{scc}$ is an equivalence relation on the state space of G. A strongly connected component is an equivalence class of $\cong_{scc}$. ■

Example 9.5 [Strongly connected components] For the transition graph of Figure 9.1, there is a single strongly connected component σ that contains all the states. Consider the transition graph of Figure 9.2. It has three strongly connected components: $\sigma_0 = \{s_0, s_1, s_3\}$, $\sigma_1 = \{s_2, s_4\}$, and $\sigma_2 = \{s_5\}$. ■

Figure 9.3 shows an algorithm to compute the partition $\cong_{scc}$ in time linear in the number of states and transitions. The algorithm involves two depth first searches. The first search involves the function *DepthFirstSearch*. For every reachable state s, there is precisely one invocation of *DepthFirstSearch* with input s. Let us order states in σ^R according to the termination times of the corresponding invocations of *DepthFirstSearch*: with each state $s \in \sigma^R$, associate a number $1 \leq done_s \leq |\sigma^R|$ such that if *DepthFirstSearch*(s) terminates before *DepthFirstSearch*(t) then $done_s < done_t$. Verify that at the end of the first search, the stack E contains all reachable states ordered in reverse according to the numbering *done*.

For every state s, the *forefather* of s, denoted $forefather_s$, is the state $t \in post^*(s)$ such that for all $u \in post^*(s)$, $done_u \leq done_t$. Thus, forefather of a state s is the state for which *DepthFirstSearch* terminates last among all the states reachable from s. States belonging to the same strongly connected component share the forefather:

Lemma 9.1 [Forefather and strongly connected component] *For two states s and t in σ^R, $s \cong_{scc} t$ iff $forefather_s = forefather_t$.*

Exercise 9.6 {T3} [Forefather and strongly connected component] Prove Lemma 9.1. ■

Now consider the state s with the highest value of $done_s$. Clearly, s is its own forefather, and furthermore, if $s \in post^*(t)$, then $forefather_t = s$. By Lemma 9.1, the strongly connected component containing s contains precisely those states from which s is reachable, that is, states in $pre^*(s)$. The second depth first search begins by invoking *DepthFirstSearch2*(s) since s is on top of the stack, and searches the graph to compute $pre^*(s)$. When *DepthFirstSearch2*(s) terminates, the region τ equals $pre^*(s)$, and is the first strongly connected component. The remaining strongly connected components are computed in the same fashion.

Theorem 9.1 [Strongly connected components] *Let G be a finite transition graph. Algorithm 9.1 correctly computes the strongly connected components of the reachable region of G.*

Example 9.6 [Computation of strongly connected components] Figure 9.4 illustrates a possible execution of Algorithm 9.1 on the transition graph of Figure 9.2. The value of *done* and *forefather* is listed along with each state. The

Algorithm 9.1 [Strongly connected components]

Input: a finite transition graph $G = (\Sigma, \sigma^I, \rightarrow)$.
Output: the partition π_{scc} of the reachable region σ^R of G into strongly connected components.

```
input G: enumgraph
local σ, τ: enumreg;  E: stack of state;  s: state;  π: partition
begin
  σ := EmptySet;
  π := EmptySet;
  E := EmptyStack;
  foreach s in InitQueue(G) do
    if not IsMember(s, σ) then DepthFirstSearch(s) fi
    od;
  σ := EmptySet;
  while not EmptyStack(E) do
    s := Top(E);
    E := Pop(E);
    if not IsMember(s, σ) then
      τ := EmptySet;
      DepthFirstSearch2(s);
      π := Insert(τ, π)
      fi
    od;
  return π
  end.

function DepthFirstSearch
  input s: state;
  local t: state;
  begin
    σ := Insert(s, σ);
    foreach t in PostQueue(s, G) do
      if not IsMember(t, σ) then DepthFirstSearch(t) fi
      od;
    E := Push(s, E)
    end.

function DepthFirstSearch2
  input s: state;
  local t: state;
  begin
    σ := Insert(s, σ);
    τ := Insert(s, τ);
    foreach t in PreQueue(s, G) do
      if not IsMember(t, σ) then DepthFirstSearch2(t) fi
      od;
    end.
```

Figure 9.3: Computing strongly connected components

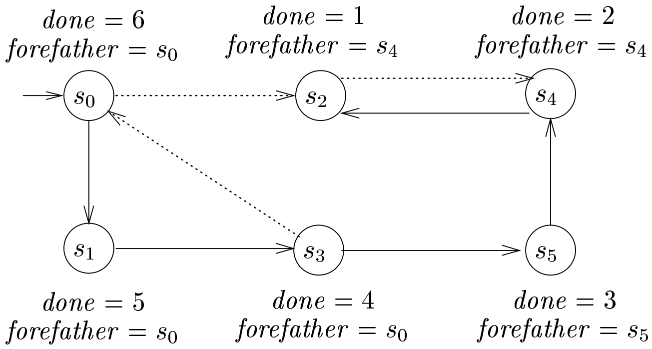

Figure 9.4: Computation of strongly connected components

undotted edges correspond to the edges that lead to exploration of states for the first time. At the end of the first depth-first search, s_0 is on top of the stack E with $done_{s_0} = forefather_{s_0}$. The first strongly connected component equals $\sigma_0 = pre^*(s_0) = \{s_0, s_1, s_3\}$. Once *DepthFirstSearch2*(s_0) terminates, the value of σ equals σ_0, and the state s_5 is on top of the stack with $done_{s_5} = forefather_{s_5}$. Consequently, the second strongly connected component equals $\sigma_2 = pre^*(s_5)\backslash\sigma_0 = \{s_5\}$. Once *DepthFirstSearch2*(s_5) terminates, the value of σ equals $\sigma_0 \cup \sigma_2$, and the state s_4 is on top of the stack with $done_{s_4} = forefather_{s_4}$. Consequently, the third strongly connected component equals $\sigma_1 = pre^*(s_4)\backslash(\sigma_0 \cup \sigma_2) = \{s_2, s_4\}$. ■

Given an enumerative representation of the transition graph G, Algorithm 9.1 can be implemented so that it requires time $O(n+m)$, where n is the number of states in G and m is the number of transitions in G. For this purpose, in the second depth first search, the computation of *PreQueue* needs to be efficient. Given an adjacency list representation of a transition graph that gives, for every state s, a list of states in $post(s)$, one can construct, in linear time, a representation that gives, for every state s, a list of states in $pre(s)$.

The optimization techniques discussed in Section 2.3.4 can be applied to improve the performance of Algorithm 9.1. In particular, we can use an on-the-fly representation of a graph.

Exercise 9.7 {`P3`} [Computing predecessor region] Consider the on-the-fly representation of the transition graph G_P of a propositional module P. Write an algorithm for computing the operation *PreQueue* for the on-the-fly representation. What is the running time of your algorithm? ■

Exercise 9.8 {`T3`} [Tarjan's algorithm] Algorithm 9.1 uses two separate depth first searches to compute strongly connected components. An alternative strategy is to explicitly compute the forefather of every state. Then, the strongly connected components can be computed in the first search itself. During the call *DepthFirstSearch*(s), once all the successors of s are explored, the forefather of s can be computed from the forefathers of its successors. The set τ is used to store the current strongly connected component, and the set π is used to store all the strongly connected components computed so far. Once $forefather_s$ is determined, s is added to τ. If s is its own forefather, then we can conclude that τ is a strongly connected component, add it to π, and reinitialize τ to the empty region. Write an algorithm that implements this strategy. Unlike Algorithm 9.1, this strategy does not require implementation of the operation *PreQueue*, but it needs to store forefathers explicitly. In an on-the-fly implementation, this strategy would require $O(n \log n)$ memory (that gets accessed randomly) as opposed to $O(n)$ memory required by Algorithm 9.1. ■

9.2.2 Fair components

For a region σ of a transition graph G, a σ-cycle is a cycle $\overline{s}_{0..m}$ of G such that $s_i \in \sigma$ for each $0 \leq i \leq m$. A strongly connected region σ is *F-fair* for the fairness assumption F if there exists a F-fair σ-cycle. Since every strongly connected region is a subset of a strongly connected component, if σ is a F-fair strongly connected region, then the strongly connected component that contains σ is also F-fair.

Proposition 9.4 [Fair components] *A fair graph (G, F) has a reachable fair cycle iff some strongly connected component of the reachable subgraph is F-fair.*

Thus, to solve the fair cycle problem, it suffices to compute the strongly connected components of the reachable subgraph, and check if one of them is F-fair. For a single fairness constraint $f = (\alpha, \beta)$, the strongly connected component σ is f-fair precisely when either one of the following two conditions holds: (1) $(s,t) \in \beta$ for two states s and t in σ, or (2) there is a σ-cycle $\overline{s}_{0..m}$ such that $(s_i, s_{i+1}) \notin \alpha$ for all $0 \leq i < m$. Each of these conditions may be tested using a depth-first search of the region σ. For a fairness assumption with several constraints, checking the fairness of a strongly connected component is trickier.

Exercise 9.9 {`T3`} [Fair regions] Let (G, F) be a finite fair graph, and σ be a strongly connected component of G. Prove or disprove the claim: σ is F-fair iff σ is f-fair for each fairness constraint $f \in F$. Is the claim true when F is local? ■

Consider a strongly connected component σ of G. If σ is β-fair for every (α, β) in F then σ is F-fair. Let $f = (\alpha, \beta)$ be a fairness constraint in F. If there are no two states s and t in σ such that $s \xrightarrow{\beta} t$, then σ cannot contain a β-fair cycle. Consequently, for a σ-cycle to be F-fair, it needs to be α-unfair. This implies that to search for F-fair σ-cycles, we can delete all the transitions in α. After this transformation we no longer need to consider the constraint f. However,

Algorithm 9.2 [Fair strongly connected components]

> **function** *FSCC*
> Input: a finite transition graph $G = (\Sigma, \sigma^I, \rightarrow)$, and a fairness assumption F for G.
> Output: YES if there is a F-fair strongly connected component of G, and NO otherwise.
>
> **foreach** $\sigma \in SCC(G)$ **do**
> $\rightarrow' := \{(s,t) \mid s \rightarrow t \text{ and } s,t \in \sigma\}$;
> $F' := \emptyset$;
>   **if** $\rightarrow' \neq \emptyset$ **then**
>   **foreach** $(\alpha, \beta) \in F$ **do**
>   **if** $\rightarrow' \cap \beta \neq \emptyset$
>   **then** $F' := \mathit{Insert}((\alpha, \beta), F')$
>   **else** $\rightarrow' := \rightarrow' \setminus \alpha$
>   **fi**
>   **od**;
>   **if** $F' = F$
>   **then return** YES
>   **else if** $FSCC(\sigma, \sigma, \rightarrow', F') =$ YES **then return** YES **fi**
>   **fi**
>   **od**;
> **return** NO.

Figure 9.5: Computing fair strongly connected components

following this transformation, σ may no longer be strongly connected, hence, we need to compute strongly connected components of σ again. Algorithm 9.5 presents a recursive scheme that implements this strategy.

The fair components are computed by the recursive function *FSCC*. It uses the subroutine *SCC* that, given an input transition graph, returns the list of its strongly connected components. The function *SCC* can be implemented using, for instance, Algorithm 9.1.

Proposition 9.5 [Fair components] *Let G be a transition graph with finitely many reachable states, and let F be a fairness assumption for G. Algorithm 9.2 computes the F-fair strongly connected components of G.*

Example 9.7 [Computation of fair components] Consider the transition graph of Figure 9.1. The fairness assumption F contains two fairness constraints

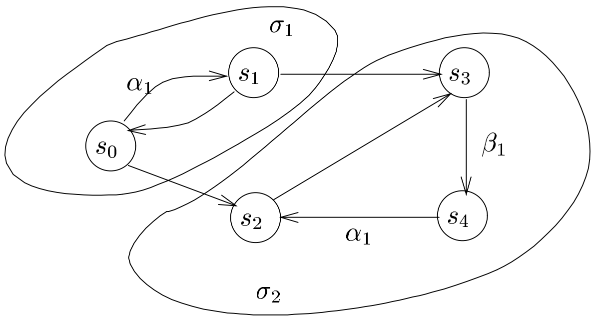

Figure 9.6: Computation of fair components

(α_1, β_1) and (α_2, β_2). The original graph contains a single strongly-connected component σ. Since σ does not contain any transition in β_2, all the transitions in α_2 are deleted along with the fairness constraint (α_2, β_2). Consequently, *FSCC* calls itself recursively on the graph shown in Figure 9.6 with a single fairness constraint (α_1, β_1). Observe that this graph is no longer strongly connected, and has two strongly connected components $\sigma_1 = \{s_0, s_1\}$, and $\sigma_2 = \{s_2, s_3, s_4\}$.

The component σ_1 does not contain any transition in β_1. So the algorithm deletes the α_1-transition from s_0 to s_1. A recursive call would split σ_1 into two singleton components $\{s_0\}$ and $\{s_1\}$, and since both of these do not contain any transitions, $FSCC(\{s_0, s_1\}, \{s_0, s_1\}, \{(s_1, s_0)\}, \emptyset)$ returns NO.

The component σ_2 contains a transition in β_1, and hence, is fair. ■

Exercise 9.10 {P3} [Fair cycle computation] Write an algorithm to solve the fair-cycle problem. The input to the algorithm is a finite fair graph $\mathcal{G}$, and if the answer to the fair-cycle problem $\mathcal{G}$ is YES, it should return a witness. ■

Observe that in Algorithm 9.2, every time *FSCC* calls itself recursively, at least one fairness constraint is removed. Consequently, the depth of recursion is bounded by the number of fairness constraints in the original fairness assumption.

One possible way to implement Algorithm 9.2 is to employ an enumerative representation of the transition graph, and to represent each action of each fairness constraint as a list of pairs of states.

Theorem 9.2 [Enumerative fair components] *Let G be a finite transition graph with n states and m transitions, and let F be a fairness assumption with ℓ constraints. Algorithm 9.2 computes the F-fair strongly connected components of G in time $O((n+m) \cdot \ell^2)$.*

As in case of invariant verification, instead of computing the transition graph of a module a priori, one can use an on-the-fly representation. Each fairness constraint of a fair module is represented by its update choice and its type (i.e. weak versus strong). The request and the response regions are represented by the corresponding predicates. The computation within the function *FSCC* involves deletion of transitions. Instead of performing this deletion explicitly, it suffices to remember which actions have been deleted, and pass this information to *SCC*.

9.3 Recurrence Verification for Weak Fairness

Consider the recurrence verification problem $(\mathcal{P}, p)$ where the module $\mathcal{P}$ has only weak-fairness constraints. Then, the graph $\mathcal{G}_{\mathcal{P},p}$ has weak-fairness constraints with an additional constraint $([\![p]\!], \emptyset)$. In this case, the fair-cycle problem can solved more efficiently without explicitly computing the strongly-connected components.

9.3.1 Single Büchi constraint

A weak-fairness constraint is of the form $(\rightarrow, \alpha)$. When the action α is specified by a region, then the weak-fairness constraint is called Büchi constraint. Thus, a Büchi constraint for a transition graph G is specified by a region σ^T of G, and an ω-trajectory $\underline{s}$ is fair if it is σ^T-fair (i.e. $s_i \in \sigma^T$ for infinitely many $i \geq 0$).

Let us consider the special case of the fair-cycle problem (G, F) when the fairness assumption F contains a single Büchi constraint specified by the region σ^T. The fair-cycle problem is, thus, to decide if there exists a reachable cycle that contains some state in σ^T. One solution is to first compute the reachable strongly connected components of G, and then check if the region σ^T has nonempty intersection with some strongly connected component. However, in this special case, there is no need to explicitly compute the strongly connected components. The algorithm of Figure 9.7 presents an improved solution.

The algorithms involves two nested searches, a primary search performed by the function *DepthFirstSearch* and a secondary (or nested) search performed by the function *NestedDepthFirstSearch*. The states encountered during the primary search are stored in the set σ, while the states visited during the secondary search are stored in the set τ. As in a standard depth first search, for every reachable state s of G, the function *DepthFirstSearch* is invoked at most once with input state s. Once the primary search originating at s terminates, if the state s belongs to the target region σ^T, then a secondary search is initiated by calling *NestedDepthFirstSearch* with input s. The objective of this secondary search is to find a cycle starting at the state s. When *NestedDepthFirstSearch*(s) is invoked, the stack E contains an initialized trajectory leading to state s. Thus, if the secondary search visits a state belonging to the stack, then it concludes that there is a cycle that contains s. This establishes that whenever the algorithm returns the answer YES the graph contains a reachable cycle containing a state in σ^T.

Let us again order the states according the termination times of the primary search: with each state $s \in \sigma^R$, associate a number $1 \leq done_s \leq |\sigma^R|$ such that if *DepthFirstSearch*(s) terminates before *DepthFirstSearch*(t) then $done_s < done_t$. Suppose the graph G contains a reachable cycle containing some state in the target region σ^T. Let $s_0, \ldots s_k$ be the ordering of states in $\sigma^T \cap \sigma^R$ according to the numbering *done*. Let s_i be the first state in this ordering that belongs to a cycle, and let $\upsilon = \cup 0 \leq j < i.\, post^*(s_j)$. Verify that s_i does not belong to υ (otherwise, there is a cycle containing some s_j for $j < i$). In fact, the cycle that contains s_i is disjoint from υ. When the primary search from state s_i is over, the set τ containing the states visited by the secondary search so far equals υ. Consequently, *NestedDepthFirstSearch* will be invoked with input s_i, and will find a cycle containing s_i.

Proposition 9.6 [Nested search for single Büchi] *Let G be a transition graph with finitely many reachable states, and let σ^T be a region of G. Then Algorithm 9.3 solves the fair cycle problem $(G, \{(\Sigma, \sigma^T)\})$.*

For complexity analysis of Algorithm 9.3, for a reachable state s, both the routines *DepthFirstSearch* and *NestedDepthFirstSearch* are invoked at most once with input s. Using a standard enumerative representation, the algorithm can be implemented with linear running time.

Theorem 9.3 [Fair-cycle for single Büchi] *Let G be a finite transition graph, and let σ^T be a region of G. Given the input $\{G\}_e$ and $\{\sigma\}_e$, Algorithm 9.3 solves the fair-cycle problem $(G, \{(\Sigma, \sigma^T)\})$ in $O(n+m)$ time and $\Theta(n+m)$ space, where n is the number of states and m is the number of transitions of G.*

Observe that, unlike the solution to the fair cycle problem from the previous section, Algorithm 9.3 does not involve computation of *PreQueue*, and may terminate even before visiting all the reachable states of the graph.

Exercise 9.11 {T3} [Witness reporting] Given an input graph G and a region σ^T, suppose Algorithm 9.3 terminates with answer YES. Let $\overline{s}_{0\ldots m}$ be the contents of the stack E, in reverse order, upon termination of the algorithm. Show that $\overline{s}_{0\ldots m}$ is an initialized trajectory of G, and $s_i \in post(s_m)$ for some $0 \leq i \leq m$. Modify Algorithm 9.3 so that it returns either the answer NO or a witness to the fair-cycle problem. ■

Algorithm 9.3 [Nested Depth-first Search for Single Büchi]

Input: a finitely branching transition graph G, and a finite region σ^T of G.
Output: the answer to the fair-cycle problem $(G, \{(\Sigma, \sigma^T)\})$

```
input G: enumgraph;  σ^T: enumreg;
local σ, τ: enumreg;  E: stack of state;  s: state
begin
  σ := EmptySet; τ := EmptySet; E := EmptyStack;
  foreach s in InitQueue(G) do
    if not IsMember(s, σ) then
      if DepthFirstSearch(s) then return YES fi;
      fi;
    od;
  return NO
  end.

function DepthFirstSearch: 𝔹
  input s: state;
  local t: state;
  begin
    E := Push(s, E); σ := Insert(s, σ);
    foreach t in PostQueue(s, G) do
      if not IsMember(t, σ) then
        if DepthFirstSearch(t) then return true fi;
        fi;
        od;
    if IsMember(s, σ^T) and not IsMember(s, τ) then
      if NestedDepthFirstSearch(s) then return true fi
      fi;
    E := Pop(E);
    return false
    end.

function NestedDepthFirstSearch: 𝔹
  input s: state;
  local t: state;
  begin
    τ := Insert(s, τ);
    foreach t in PostQueue(s, G) do
      if IsMember(t, E) then return true fi;
      if not IsMember(t, τ) then
        if NestedDepthFirstSearch(t) then return true fi;
        fi;
      od;
    return false
    end.
```

Figure 9.7: Nested search for a fair cycle with a single Büchi constraint

Exercise 9.12 {P3} [Nested DFS for a single weak-fairness constraint] Algorithm 9.3 solves the fair-cycle problem for a single Büchi constraint specified by a region. Modify the algorithm to solve the fair-cycle problem for a single weak-fairness constraint. That is, write an algorithm that takes as input, a transition graph G and an action α of G, returns the answer to the fair-cycle problem $(G, \{(\rightarrow, \alpha)\})$. ■

9.3.2 Multiple weak-fairness constraints

Now, let us consider the fair-cycle problem (G, F) when F is a weak-fairness assumption. It is possible to translate this problem to a fair-cycle problem for a single weak-fairness constraint by augmenting the states of G with a counter variable. Let $\alpha_1, \ldots \alpha_\ell$ be an enumeration of the actions in F. The counter is initially 1, and is incremented from i to $i+1$, treating $\ell+1 = 1$, when transition in α_i is encountered. Visiting all of α_i infinitely often corresponds to updating the counter from ℓ to 1 infinitely often.

FROM MULTIPLE WEAK CONSTRAINTS TO A SINGLE WEAK CONSTRAINT

Let $\mathcal{G} = (\Sigma, \sigma^I, \rightarrow, F)$ be a weakly-fair graph, where F is a weak-fair assumption with ℓ weak-fair constraints specified by actions $\alpha_1 \ldots \alpha_\ell$. The fair graph $B_\mathcal{G}$ has the following components:

- for every state s of $\mathcal{G}$, and for every $1 \leq i \leq \ell$, the pair (s, i) is a state of $B_\mathcal{G}$;
- for every initial state s of $\mathcal{G}$, the pair $(s, 1)$ is an initial state of $B_\mathcal{G}$;
- for every action α_i in F, for every transition $s \stackrel{\alpha_i}{\rightarrow} t$ in α_i, if $i < \ell$ then $B_\mathcal{G}$ has a transition from (s, i) to $(t, i+1)$, and if $i = \ell$ then $B_\mathcal{G}$ has a transition from (s, i) to $(t, 1)$;
- the fairness assumption of $B_\mathcal{G}$ contains a single weak constraint α: for every transition $s \stackrel{\alpha_\ell}{\rightarrow} t$, $(s, \ell) \stackrel{\alpha}{\rightarrow} (t, 1)$.

Proposition 9.7 [Multiple weak-fairness to single weak constraint] *For every weakly-fair graph $\mathcal{G}$, the answer to the fair-cycle problem $\mathcal{G}$ coincides with the answer to the fair-cycle problem $B_\mathcal{G}$.*

Exercise 9.13 {T2} [Multiple weak-fairness to single weak constraint] Prove Proposition 9.7. ■

If the weakly-fair graph $\mathcal{G}$ has n states, m transitions, and ℓ fairness constraints, the weakly-fair graph $B_\mathcal{G}$ has $n \cdot \ell$ states and $m \cdot \ell$ transitions. It follows that the fair-cycle problem $\mathcal{G}$ can be solved in time $O((n+m) \cdot \ell)$. Contrast this with the complexity $O((n+m) \cdot \ell^2)$ of solving the fair-cycle problem in the general case using Algorithm 9.2.

9.3.3 Recurrence verification

Now we are ready to present the solution to the recurrence-verification problem $(\mathcal{P}, p)$ when $\mathcal{P}$ has only weak-fairness constraints. In this case, to answer positively to the fair-cycle problem, the algorithm needs to find a reachable cycle that contains at least one transition from each of the weak-fairness constraints of $\mathcal{P}$, and does not contain any p-state. This can be formulated as searching for a F-fair $[\![\neg p]\!]$-cycle. Figure 9.8 shows an algorithm to solve this problem. The input is a transition graph G together with a set F of weak-fairness constraints and a region σ^T. The goal is to find a reachable σ^T-cycle that is F-fair.

The algorithm is similar in spirit to Algorithm 9.3, and involves two nested depth-first searches. Multiple weak-fairness constraints are reduced to a single weak-fairness constraint by adding a counter variable as discussed in Section 9.3.2. If F contains ℓ constraints, then the modified search space is $\Sigma_G \times \{1, \ldots \ell\}$. The algorithm uses the following operations:

Size: **set** $\mapsto$ **integer.** The operation *Size* returns the number of elements in its set argument.

MultiWeakPost: **state** $\times$ **integer** $\times$ **enumgraph** $\times$ **set of action** $\mapsto$ **queue of state** $\times$ **integer.** The operation $\mathit{MultiWeakPost}(s, i, G, F)$ returns a queue that contains the successors of the state (s, i) in the fair graph $B_{G,F}$.

The operation $\mathit{MultiWeakPost}(s, i, G, F)$ can be implemented from the operation $\mathit{PostQueue}(s, G)$ and membership test for the actions in F.

The fair graph $B_{G,F}$ has a single weak-fairness constraint α containing transitions in which the counter is updated from ℓ to 1. The secondary search is invoked at the conclusion of the primary search from a source state of some transition in α. The secondary search is performed by the function *NestedDepthFirstSearch* of Figure 9.9. The goal of the secondary search is to find a fair cycle. Consider a fair transition from (s, ℓ) to $(t, 1)$. Then, the variable *Root* is set to s, and the function *NestedDepthFirstSearch* checks if the state (s, ℓ) is reachable from the state $(t, 1)$. Since the fair cycle is required to stay within the region σ^T, search is restricted to the region σ^T. The correctness of Algorithm 9.4 can be established as in case of Algorithm 9.3: if $(s, \ell) \rightarrow (t, 1)$ and $(s', \ell) \rightarrow (t', 1)$ are two fair transitions such that (1) the primary search from (s, ℓ) terminates before the primary search from (s', ℓ), (2) there is no σ^T-path from $(t, 1)$ to (s, ℓ), then if there is a σ^T-path from $(t', 1)$ to (s', ℓ), then it does not contain any state reachable from $(t, 1)$.

Proposition 9.8 [Recurrence verification for weak-fairness] *Let G be a transition graph with finitely many reachable states, let F be a set of weak-fairness constraints for G, and let σ^T be a region of G. Then Algorithm 9.4 solves the fair cycle problem $(G, F \cup \{(\neg\sigma^T, \emptyset)\})$.*

Algorithm 9.4 [Primary Depth First Search for Weak Recurrence Verification]

Input: a finitely branching transition graph G, weak fairness assumption F, and a region σ^T of G.
Output: the answer to the fair-cycle problem $(G, F \cup \{(\neg\sigma^T, \emptyset)\})$

```
input G: enumgraph;  F: set of action;  σ^T: enumreg;
local σ, τ: set of state × integer; s, Root: state
begin
  σ := EmptySet;
  τ := EmptySet;
  foreach s in InitQueue(G) do
    if not IsMember((s, 1), σ) then
      if DepthFirstSearch(s, 1) then return YES fi;
      fi;
    od;
  return NO
  end.

function DepthFirstSearch: 𝔹
  input s: state;  i: integer;
  local t: state;
  begin
    σ := Insert((s, i), σ);
    foreach (t, j) in MultiWeakPost(s, i, G, F) do
      if not IsMember((t, j), σ) then
        if DepthFirstSearch(t, j) then return true fi;
        fi;
      od;
    if i = Size(F) and IsMember(s, σ^T)
        and not IsMember((s, i), τ) then
      Root := s;
      foreach (t, j) in MultiWeakPost(s, i, G, F) do
        if j = 1 and IsMember(t, σ^T)
            and not IsMember((t, j), τ) then
          if NestedDepthFirstSearch(t, j) then return true fi
          fi
        od;
      fi;
    return false
    end.
```

Figure 9.8: Primary search for a fair cycle for weak recurrence verification

function *NestedDepthFirstSearch*: $\mathbb{B}$
  input s: **state**; i: **integer**
  local t: **state**; j: **integer**
  begin
    $\tau := \mathit{Insert}((s,i),\tau)$;
    foreach (t,j) **in** $\mathit{MultiWeakPost}(s,i,G,F)$ **do**
      if $t = \mathit{Root}$ **and** $j = \mathit{Size}(F)$ **then return** *true* **fi**;
      if not $\mathit{IsMember}((t,j),\tau)$ **and** $\mathit{IsMember}(t,\sigma^T)$ **then**
        if $\mathit{NestedDepthFirstSearch}(t,j)$ **then return** *true* **fi**;
      fi;
    od;
    return *false*
    end.

Figure 9.9: Secondary search for a fair cycle for weak recurrence verification

For complexity analysis of Algorithm 9.4, for a reachable state s, and an integer $1 \le i \le \ell$, both the routines *DepthFirstSearch* and *NestedDepthFirstSearch* are invoked at most once with input (s,i). Using a standard enumerative representation, the algorithm can be implemented with running time $((n+m)\cdot\ell)$.

Exercise 9.14 {P3} [Witness reporting] Modify Algorithm 9.4 so that it returns either the answer No or a witness to the fair-cycle problem. ■

Exercise 9.15 {T3} [Modifying the weak recurrence verification algorithm] Suppose we modify Algorithm 9.4 so that at the conclusion of the primary search from a state $(s,\ell) \notin \tau$ with $s \in \sigma^T$, the state (s,ℓ) is added to the set τ before invoking the secondary search (i.e. add the line $\tau := \mathit{Insert}((s,i),\tau)$ immediately after the assignment $\mathit{Root} := s$ in Figure 9.8). Does the modified algorithm correctly solve the fair-cycle problem? ■

Exercise 9.16 {P3} [Fair-cycle for single strong fairness constraint] Consider the fair-cycle problem $(G,\{(\sigma,\tau)\})$ with a single strong-fairness constraint. The goal is, then, to find a reachable cycle that is either τ-fair or not σ-fair. Write a nested depth first search algorithm to solve this special case. ■