State Space Exploration

Wieger Wesselink

April 21, 2024

1 Graph Exploration

State space exploration is an instance of graph exploration. Consider a directed graph and take a node s_0. We assume there is a function *successors* that returns the successor nodes of a vertex. An abstract algorithm for exploring the graph starting from vertex s_0 is

Algorithm 1 Graph exploration

```
ExploreGraph(s_0)
 1: todo := {s_0}
 2: discovered := {s_0}
 3: while todo ≠ ∅ do
 4:     choose s ∈ todo
 5:     todo := todo \ {s}
 6:     discovered := discovered ∪ {s}
 7:     for s' ∈ successors(s) do
 8:         if s' ∉ discovered then
 9:             discovered := discovered ∪ {s'}
10:             todo := todo ∪ {s'}
```

1.1 Event points

There are many different applications of state space exploration. The Boost Graph Library ([**?**]) uses a clever idea to separate such applications from the exploration itself. It is done by distinguishing *event points* in the algorithm that the user can respond to by means of callback functions. For our purposes we select the following events:

discover_state	is invoked when a state is encountered for the first time
examine_transition	is invoked on every transition
start_state	is invoked on a state right before its outgoing transitions are being explored
finish_state	is invoked on a state after all of its outgoing transitions have been explored

The events are named in terms of states and transitions instead of vertices and edges, since this is closer to our application domain. The exploration algorithm with event points included looks like this:

Algorithm 2 Graph exploration with event points

ExploreGraph(s_0, discover_state, examine_transition, start_state, finish_state)

```
 1: todo := {s_0}
 2: discovered := {s_0}
 3: discover_state(s_0)
 4: while todo ≠ ∅ do
 5:     choose s ∈ todo
 6:     todo := todo \ {s}
 7:     start_state(s)
 8:     discovered := discovered ∪ {s}
 9:     for s' ∈ successors(s) do
10:         if s' ∉ discovered then
11:             discovered := discovered ∪ {s'}
12:             discover_state(s')
13:             todo := todo ∪ {s'}
14:         examine_transition(s, a, s')
15:     finish_state(s)
```

2 Applications

Many applications can be easily expressed in terms of the given event points.

2.1 Deadlock checking

With deadlock checking we are looking for states that have no outgoing transitions. By introducing one boolean variable has_transitions we can implement deadlock checking as follows. The callback functions are printed as comments in gray.

Algorithm 3 Deadlock checking implemented using event points

Input:
FindDeadlock(s_0, discover_state, examine_transition, start_state, finish_state)

```
 1:                                                          ▷ bool has_transitions
 2: todo := {s_0}
 3: discovered := {s_0}
 4: discover_state(s_0)
 5: while todo ≠ ∅ do
 6:     choose s ∈ todo
 7:     todo := todo \ {s}
 8:     start_state(s)                                       ▷ has_transitions := false
 9:     discovered := discovered ∪ {s}
10:     for s' ∈ successors(s) do
11:         if s' ∉ discovered then
12:             discovered := discovered ∪ {s'}
13:             discover_state(s')
14:             todo := todo ∪ {s'}
15:         examine_transition(s, a, s')                     ▷ has_transitions := true
16:     finish_state(s)                                      ▷ if (!has_transitions) report_deadlock(s)
```

3 Search strategies

Exploration can be done with different search strategies. We describe three of them: breadth-first, depth-first and highway. They mainly differ in the order in which the elements of the todo set are processed. In breadth-first search nodes at the present depth are explored before nodes at a higher depth. In depth-first search the highest-depth nodes are explored first. Highway search is a variant that uses a breadth-first search, but it only explores a part of the state space.

In all three cases the *todo* list is stored in a double ended queue. We use the slicing operator to denote parts of a list. For example, $A[m:n]$ corresponds to the sublist $A[m, \ldots, n-1]$.

3.1 Breadth-first search

Algorithm 4 Breadth-first search

Input:
EXPLOREGRAPHBREADTHFIRST(s_0, discover_state, examine_transition, start_state, finish_state)

```
1:  todo := [s_0]
2:  discovered := {s_0}
3:  discover_state(s_0)
4:  while |todo| > 0 do
5:      s := todo[0]
6:      todo := todo[1 : |todo|]
7:      start_state(s)
8:      discovered := discovered ∪ {s}
9:      for s' ∈ successors(s) do
10:         if s' ∉ discovered then
11:             discovered := discovered ∪ {s'}
12:             discover_state(s')
13:             todo := todo ++ [s']
14:         examine_transition(s, a, s')
15:     finish_state(s)
```

3.2 Depth-first search

Algorithm 5 Depth-first search

Input:
ExploreGraphDepthFirst(s_0, discover_state, examine_transition, start_state, finish_state)

```
1: todo := [s_0]
2: discovered := {s_0}
3: discover_state(s_0)
4: while |todo| > 0 do
5:     s := todo[|todo| − 1]
6:     todo := todo[0 : |todo| − 1]
7:     start_state(s)
8:     discovered := discovered ∪ {s}
9:     for s' ∈ successors(s) do
10:        if s' ∉ discovered then
11:            discovered := discovered ∪ {s'}
12:            discover_state(s')
13:            todo := todo ++ [s']
14:        examine_transition(s, a, s')
15:    finish_state(s)
```

3.3 Highway search

In highway search (see [**?**]) a breadth first search is done, with the restriction that at most N states are put in the todo list for each level. The variable L maintains the number of states in the todo list corresponding to the current level, and the variable c counts how many elements have been added corresponding to the next level. Once c reaches the maximum value N, elements are being overwritten randomly.

Remark 1 *The specification below deviates from the published version of highway search in the sense that overwritten elements are added to the set discovered. To avoid this, the structure of the algorithm needs to be changed significantly.*

In Algorithm 1 of [**?**], the set Q_d stores todo elements corresponding to the current level, and the set Q_{d+1} stores todo elements corresponding to the next level. The algorithm above uses only one list *todo* that stores both of them. At each iteration of the while loop the first L elements of *todo* list belong to the current level, and the remaining elements belong to the next level. Furthermore, the algorithm above contains only one application of a random generator, compared to two applications in the original version. The element k is chosen randomly in the range $[1, \ldots, c]$. There is an N/c probability that this value is in the range $[1, \ldots, N]$. If k is inside the range, the element in the *todo* list with index k (counting from the end) is overwritten. This behaviour matches with the published version.

Algorithm 6 Highway search

Input:
EXPLOREGRAPHHIGHWAY(s_0, N, discover_state, examine_transition, start_state, finish_state)

```
1: todo := [s_0]
2: discovered := {s_0}
3: discover_state(s_0)
4: L := |todo|
5: c := 0
6: while |todo| > 0 do
7:     s := todo[0]
8:     todo := todo[1 : |todo|]
9:     start_state(s)
10:    for s' ∈ successors(s) do
11:        if s' ∉ discovered then
12:            discovered := discovered ∪ {s'}
13:            discover_state(s')
14:            c := c + 1
15:            if c ≤ N then
16:                todo := todo ++ [s']
17:            else
18:                k := random({1, ..., c})
19:                if k ≤ N then
20:                    todo[|todo| − k] := s'
21:        examine_transition(s, a, s')
22:    finish_state(s)
23:    L := L − 1
24:    if L = 0 then
25:        L := |todo|
26:        c := 0
```

4 Cycle detection

For cycle detection the event points in table 1.1 are insufficient. In [**?**] the following recursive depth first algorithm is given:

Algorithm 7 Recursive cycle detection algorithm as specified in Boost

Input:
BOOST_DFS_RECURSIVE(u)
1: $color[u] := gray$
2: discover_vertex(u)
3: **for** $(a, v) \in out_edges(u)$ **do**
4: examine_edge(a, v)
5: **if** $color[v] = white$ **then**
6: tree_edge(a, v)
7: DFS_RECURSIVE(v)
8: **else if** $color[v] = gray$ **then**
9: back_edge(a, v)
10: **else**
11: forward_or_cross_edge(a, v)
12: $color[u] := black$
13: finish_vertex(u)

The code in Boost uses an iterative version:
For our purposes we rewrite this as:
Whenever the back_edge event is triggered, a cycle is found.

Algorithm 8 Iterative cycle detection algorithm as implemented in Boost

Input:
BOOST_DFS_ITERATIVE(u)

```
 1: color[u] := gray
 2: discover_vertex(u)
 3: stack := [(u, out_edges(u))]
 4: while |stack| > 0 do
 5:     u, E := stack.pop_back()
 6:     while |E| > 0 do
 7:         a, v := E[0]
 8:         examine_edge(u, a, v)
 9:         if color[v] = white then
10:             tree_edge(u, a, v)
11:             stack.push_back(u, E[1 :])
12:             u := v
13:             color[u] := gray
14:             discover_vertex(u)
15:             E := out_edges(u)
16:         else
17:             if color[v] = gray then
18:                 back_edge(u, a, v)
19:             else
20:                 forward_or_cross_edge(u, a, v)
21:             E := E[1 :]
22:     color[u] := black
23:     finish_vertex(u)
```

Algorithm 9 Recursive cycle detection

Input:
DFS_RECURSIVE($s_0, gray$)

```
 1: gray := gray ∪ {s_0}
 2: discovered := {s_0}
 3: discover_state(s_0)
 4: for (a, s_1) ∈ out_edges(s_0) do
 5:     examine_edge(s_0, a, s_1)
 6:     if s_1 ∉ discovered then
 7:         tree_edge(s_0, a, s_1)
 8:         discovered := discovered ∪ {s_1}
 9:         DFS_RECURSIVE(s_1, gray)
10:     else if s_1 ∈ todo then
11:         back_edge(s_0, a, s_1)
12:     else
13:         forward_or_cross_edge(s_0, a, s_1)
14:     gray := gray \ {s_0}
15:     finish_state(s_0)
```

Algorithm 10 Iterative cycle detection

Input:

DFS_ITERATIVE(s_0)

1: $todo := [(s_0, out_edges(s_0))]$
2: $discovered := \{s_0\}$
3: discover_state(s_0)
4: **while** $|todo| > 0$ **do**
5: $s, E := todo.back()$
6: **while** $|E| > 0$ **do**
7: $a, s_1 := E.pop_front()$
8: examine_edge(s_0, a, s_1)
9: **if** $s_1 \notin discovered$ **then**
10: tree_edge(s_0, a, s_1)
11: $discovered := discovered \cup \{s_1\}$
12: discover_state(s_1)
13: $todo.back() := (s, E)$
14: $todo := todo +\!\!+ [(s_1, out_edges(s_1))]$
15: $s, E := todo.back()$
16: **else if** $s_1 \in todo$ **then**
17: back_edge(s_0, a, s_1)
18: **else**
19: forward_or_cross_edge(s_0, a, s_1)
20: finish_state(s)

5 Untimed state space exploration

Consider the following untimed linear process specification P, with initial state d_0.

$$P(d) = \sum_{i \in I} \sum_{e_i} c_i(d, e_i) \to a_i(f_i(d, e_i)) \cdot P(g_i(d, e_i))$$

This linear process is a symbolic representation of a state space, or labeled transition system (LTS). The previously described graph exploration algorithms can be applied to explore a state space. Let *rewr* be a rewriter. An algorithm for untimed state space exploration is

The set E is computed using the ENUMERATE algorithm. This computation may be expensive. Hence the condition $c(d, e_i)$ is first rewritten, since if it evaluates to $false$ the computation of E can be skipped.

Algorithm 11 Untimed LPS exploration

Input:
EXPLORELPS($P(d), d_0, rewr$, discover_state, examine_transition, start_state, finish_state)

```
1: s_0 := rewr(d_0, [ ])
2: todo := {s_0}
3: discovered := {s_0}
4: discover_state(s_0)
5: while todo ≠ ∅ do
6:     choose s ∈ todo
7:     todo := todo \ {s}
8:     discovered := discovered ∪ {s}
9:     start_state(s)
10:    for i ∈ I do
11:        condition := rewr(c_i(d, e_i), [d := s])
12:        if condition = false then
13:            continue
14:        E := {e | rewr(condition, [e_i := e]) = true}
15:        for e ∈ E do
16:            a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))
17:            s' := rewr(g_i(d, e_i), [d := s, e_i := e])
18:            if s' ∉ discovered then
19:                todo := todo ∪ {s'}
20:                discovered := discovered ∪ {s'}
21:                discover_state(s')
22:            examine_transition(s, a, s')
23:    finish_state(s)
```

6 Timed state space exploration

Consider the following timed linear process specification P, with initial state d_0.

$$P(d) = \sum_{i \in I} \sum_{e_i} c_i(d, e_i) \to a_i(f_i(d, e_i)) \, \text{`}t_i(d, e_i) \cdot P(g_i(d, e_i)).$$

Note that the time tag $t_i(d, e_i)$ is optional. If it is omitted, the corresponding action may happen at an arbitrary time. In timed state space exploration, care is taken that on every trace the time tags are increasing. In order to achieve that, a time stamp is recorded for each state in the state space. We use the notation $t \ll s$ to denote the state s with associated time stamp t. An algorithm for timed state space exploration is

Algorithm 12 Timed LPS exploration

Input:

ExploreLPSTimed($P(d), d_0, rewr,$ discover_state, examine_transition, start_state, finish_state)

1: $s_0 := rewr(d_0, [\,])$
2: $todo := \{\ 0 \ll s_0)\ \}$
3: $discovered := \{\ 0 \ll s_0)\ \}$
4: discover_state$(0 \ll s_0)$
5: **while** $todo \neq \emptyset$ **do**
6: **choose** $t \ll s \in todo$
7: $todo := todo \setminus \{\ t \ll s\ \}$
8: $discovered := discovered \cup \{\ t \ll s\ \}$
9: start_state$(t \ll s)$
10: **for** $i \in I$ **do**
11: $condition := rewr(c_i(d, e_i), [d := s])$
12: **if** $condition = false$ **then**
13: **continue**
14: $E := \{e \mid rewr(condition, [e_i := e]) = true\}$
15: **for** $e \in E$ **do**
16: $t' := rewr(t_i(d, e_i), [d := s, e_i := e])$
17: **if** $t' \leq t$ **then**
18: **continue**
19: $a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))$
20: $s' := rewr(g_i(d, e_i), [d := s, e_i := e])$
21: **if** $t' \ll s' \notin discovered$ **then**
22: $todo := todo \cup \{\ t' \ll s'\ \}$
23: $discovered := discovered \cup \{\ t' \ll s'\ \}$
24: discover_state$(t' \ll s')$
25: examine_transition$(t \ll s, a \triangleleft t', t' \ll s')$
26: finish_state$(t \ll s)$

7 Stochastic state space exploration

Consider the following stochastic linear process specification P, with initial state $\frac{p(h)}{h} \cdot P(g(h))$.

$$P(d) = \sum_{i \in I} \sum_{e_i} c_i(d, e_i) \to a_i(f_i(d, e_i)) \frac{p_i(d, e_i, h_i)}{h_i} \cdot P(g_i(d, e_i, h_i)), \tag{1}$$

where p and p_i are stochastic distributions. We define a *stochastic state* as a set $\{(q_1, s_1), \ldots, (q_m, s_m)\}$ with $q_j, j = 1 \ldots m$ a sequence of non-zero probabilities that sum up to 1, and $s_j, j = 1 \ldots m$ a sequence of states. The function ComputeStochasticState is used to compute a stochastic state from its symbolic representation.

Algorithm 13 Computation of a stochastic state

Input:
ComputeStochasticState$(h, p, g, rewr, \sigma)$

1: $result := \emptyset$
2: $H := \{(h', q) \mid q = rewr(p, \sigma[h := h']) \wedge q > 0\}$
3: **for** $(h', q) \in H$ **do**
4: $\quad s := rewr(g, \sigma[h := h'])$
5: $\quad result := result \cup \{(q, s)\}$
return *result*

The set H is computed using the Enumerate algorithm.
An algorithm for stochastic state space exploration is

Algorithm 14 Stochastic LPS exploration

Input:
EXPLORELPSSTOCHASTIC($P(d)$, $\frac{p(h)}{h} \cdot P(g(h))$, $rewr$, discover_state, examine_transition, start_state, finish_state, discover_initial_state)

$\hat{s_0} :=$ COMPUTESTOCHASTICSTATE$(h, p(h), g(h), rewr, [\,])$
$S := \{s_i \mid (q_i, s_i) \in \hat{s_0}\}$
discover_initial_state$(\hat{s_0})$
for $s \in S$ **do**
    $todo := todo \cup \{s\}$
    $discovered := discovered \cup \{s\}$
    discover_state(s)
while $todo \neq \emptyset$ **do**
choose $s \in todo$
$todo := todo \setminus \{s\}$
$discovered := discovered \cup \{s\}$
start_state(s)
for $i \in I$ **do**
$condition := rewr(c_i(d, e_i), [d := s])$
if $condition = false$ **then continue**
$E := \{e \mid rewr(condition, [e_i := e]) = true\}$
for $e \in E$ **do**
    $a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))$
    $\hat{s'} :=$ COMPUTESTOCHASTICSTATE$(h_i, p_i(d, e_i, h_i), g_i(d, e_i, h_i), rewr, [d := s, e_i := e])$
    $S' := \{s_i \mid (q_i, s_i) \in \hat{s'}\}$
    for $s' \in S'$ **do**
        if $s' \notin discovered$ **then**
            $todo := todo \cup \{s'\}$
            $discovered := discovered \cup \{s'\}$
            discover_state(s')
    examine_transition$(s, a, \hat{s'})$
finish_state(s)

8 Caching

The computation of the set of solutions E in the ExploreLPS is expensive. Therefore it may be a good idea to cache these solutions. Caching can be done locally (i.e. using a separate cache for each summand), or globally. This leads to the following variants of the algorithm. We assume that FV is a function that computes free variables of an expression. Let $\mathcal{D}$ be the set of process parameters (i.e. the elements of d).

8.1 Local caching

In the local caching algorithm for each summand i a mapping C_i is maintained. The cache key is comprised of the actual values of the process parameters that appear in the condition $c_i(d, e_i)$.

Algorithm 15 LPS exploration with local caching

Input:
ExploreLPSLocallyCached($P(d), d_0, rewr$, discover_state, examine_transition, start_state, finish_state)

1: $s_0 := rewr(d_0, [\,])$
2: $todo := \{s_0\}$
3: $discovered := \{s_0\}$
4: discover_state(s_0)
5: **for** $i \in I$ **do**
6: $\quad C_i := \{:\}$
7: $\quad \gamma_i := FV(c_i(d, e_i)) \cap \mathcal{D}$
8: **while** $todo \neq \emptyset$ **do**
9: $\quad$ **choose** $s \in todo$
10: $\quad todo := todo \setminus \{s\}$
11: $\quad discovered := discovered \cup \{s\}$
12: $\quad$ start_state(s)
13: $\quad$ **for** $i \in I$ **do**
14: $\quad\quad key := \gamma_i[d := s]$
15: $\quad\quad$ **if** $key \in keys(C_i)$ **then**
16: $\quad\quad\quad E := C_i[key]$
17: $\quad\quad$ **else**
18: $\quad\quad\quad E := \{e \mid rewr(c_i(d, e_i), [d := s, e_i := e]) = true\}$
19: $\quad\quad\quad C_i := C_i \cup \{(key, E)\}$
20: $\quad\quad$ **for** $e \in E$ **do**
21: $\quad\quad\quad a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))$
22: $\quad\quad\quad s' := rewr(g_i(d, e_i), [d := s, e_i := e])$
23: $\quad\quad\quad$ **if** $s' \notin discovered$ **then**
24: $\quad\quad\quad\quad todo := todo \cup \{s'\}$
25: $\quad\quad\quad\quad discovered := discovered \cup \{s'\}$
26: $\quad\quad\quad\quad$ discover_state(s')
27: $\quad\quad\quad$ examine_transition(s, a, s')
28: $\quad$ finish_state(s)

8.2 Global caching

In the global caching algorithm one mapping C is maintained. To achieve this, the condition of the summands is added to the cache key. If many summands share the same condition, global caching may be beneficial. In practice this doesn't seem to happen much.

Algorithm 16 LPS exploration with global caching

Input:
EXPLORELPSGLOBALLYCACHED($P(d), d_0, rewr$, discover_state, examine_transition, start_state, finish_state)

1: $todo := \{d_0\}$
2: $discovered := \emptyset$
3: $C := \emptyset$
4: **for** $i \in I$ **do**
5: $\quad \gamma_i := FV(c_i(d, e_i)) \cap \mathcal{D}$
6: **while** $todo \neq \emptyset$ **do**
7: $\quad$ **choose** $s \in todo$
8: $\quad todo := todo \setminus \{s\}$
9: $\quad discovered := discovered \cup \{s\}$
10: $\quad$ start_state(s)
11: $\quad$ **for** $i \in I$ **do**
12: $\quad\quad key := c_i(d, e_i) +\!\!+ \gamma_i[d := s]$
13: $\quad\quad$ **if** $key \in keys(C)$ **then**
14: $\quad\quad\quad T := C[key]$
15: $\quad\quad$ **else**
16: $\quad\quad\quad T := \{t \mid rewr(c_i(d, e_i), [d := s, e_i := t]) = true\}$
17: $\quad\quad\quad C := C \cup \{(key, T)\}$
18: $\quad\quad$ **for** $e \in E$ **do**
19: $\quad\quad\quad a := a_i(rewr(f_i(d, e_i), [d := s, e_i := e]))$
20: $\quad\quad\quad s' := rewr(g_i(d, e_i), [d := s, e_i := e])$
21: $\quad\quad\quad$ **if** $s' \notin discovered$ **then**
22: $\quad\quad\quad\quad todo := todo \cup \{s'\}$
23: $\quad\quad\quad\quad discovered := discovered \cup \{s'\}$
24: $\quad\quad\quad\quad$ discover_state(s')
25: $\quad\quad\quad$ examine_transition(s, a, s')
26: $\quad$ finish_state(s)

In this algorithm C is a mapping, with $keys(C) = \{k \mid \exists_v : (k, v) \in C\}$. We use the notation $C[k]$ to denote the unique element v such that $(k, v) \in C$.

9 Confluence Reduction

Confluence reduction (see [**?**], [**?**] and [**?**]) is an on-the-fly state space exploration method that produces a reduced state space. For confluence reduction we assume that the set of summands I is partitioned into a set $I_{regular}$ of 'regular' summands, and a set $I_{confluent}$ of confluent *tau*-summands. The confluent *tau*-summands are used to determine a unique representative state that is reachable via confluent τ steps. This is done using the graph algorithm FindRepresentative. This leads to the following variant of the algorithm:

Algorithm 17 LPS exploration with confluence reduction

Input:
ExploreLPSConfluence($P(d), d_0, rewr$, discover_state, examine_transition, start_state, finish_state)

1: $s_0 :=$ FindRepresentative($rewr(d_0, [\,])$)
2: $todo := \{ s_0 \}$
3: $discovered := \{s_0\}$
4: discover_state(s_0)
5: **while** $todo \neq \emptyset$ **do**
6: **choose** $s \in todo$
7: $todo := todo \setminus \{s\}$
8: $discovered := discovered \cup \{s\}$
9: start_state(s)
10: **for** $i \in I_{regular}$ **do**
11: $condition := rewr(c_i(d, e_i), [d := s])$
12: **if** $condition = false$ **then**
13: **Continue**
14: $E := \{e \mid rewr(condition, [e_i := e]) = true\}$
15: **for** $e \in E$ **do**
16: $a := a_i(rewr(f_i(d, e_i), [d := s, e_i := t]))$
17: $s' :=$ FindRepresentative($rewr(g_i(d, e_i), [d := s, e_i := t])$)
18: **if** $s' \notin discovered$ **then**
19: $todo := todo \cup \{s'\}$
20: $discovered := discovered \cup \{s'\}$
21: discover_state(s')
22: examine_transition(s, a, s')
23: finish_state(s)

As suggested in [**?**] Tarjan's strongly connected component (SCC) algorithm (see [**?**]) can be used to compute a unique representative.

9.1 Tarjan's SCC algorithm

A recursive implementation of Tarjan's strongly connected components algorithm that uses four global variables *stack*, *low*, *disc* and *result*. The helper function StrongConnect computes the connected component reachable from node u. In this function it is assumed that the function call successors(u) returns the successor states of u in a deterministic order.

A side effect of a call Tarjan(u) is that *result* contains the connected components that have been found.

9.2 FindRepresentative

Due to properties of confluent τ-summands, there is always only one terminal strongly connected component, i.e. a strongly connected component without outgoing edges. Furthermore, the first strongly connected component reported by Tarjan's algorithm is always terminating. For our implementation of FindRepresentative we prefer to use an iterative version of Tarjan's SCC algorithm. The reason for this is that an

Algorithm 18 Tarjan's Strongly Connected Component Algorithm

Input: $G = (V, E)$: A graph with nodes V and edges E.
Output: *result*: A sequence containing all strongly connected components of the graph.
TARJAN(G):

```
1: stack := [ ]
2: low := {:}                                  ▷ the empty mapping is denoted as {:}
3: disc := {:}                                 ▷ u ∈ low means u is a key of mapping low
4: result := [ ]
5: for u ∈ V do
6:     if u ∉ low then STRONGCONNECT(u)
7: return result
```

Algorithm 19 Helper function StrongConnect

Input: u: An element of V.
STRONGCONNECT(u):

```
1: k := |disc|                                 ▷ k is the discovery time that is assigned to node u
2: disc[u] := k
3: low[u] := k                                 ▷ initially low[u] = disc[u]
4: stack := stack ++ [u]
5: for v ∈ successors(u) do
6:     if v ∉ low then
7:         STRONGCONNECT(v)
8:         low[u] := min(low[u], low[v])
9:     else if v ∈ stack then
10:        low[u] := min(low[u], disc[v])
11: if low[u] = disc[u] then                   ▷ an SCC has been found
12:     comp := [ ]
13:     while true do
14:         v := stack[|stack| − 1]            ▷ assign the top of the stack to v
15:         stack := stack[0 : |stack| − 1]     ▷ pop an element from the stack
16:         comp := comp ++ [v]
17:         if v = u then break
18:     result := result ++ [comp]
```

iterative version can be more easily interrupted once the first SCC has been found. The algorithm description in [**?**] has been used as a model for our solution.

Algorithm 20 Find a unique representative node in a graph

Input:

FindRepresentative(u)

```
 1: stack := [ ]
 2: low := {:}
 3: disc := {:}
 4: work := [(u, 0)]
 5: while work ≠ [ ] do
 6:     (u, i) := work[|work| − 1]
 7:     work := work[0 : |work| − 1]
 8:     if i = 0 then
 9:     k := |disc|
10:     disc[u] := k
11:     low[u] := k
12:     stack := stack ++ [u]
13:     recurse := false
14:     for j ∈ [i, . . . , |successors(u)|] do
15:     v := successors(u)[j]
16:     if v ∉ disc then
17:         work := work ++ [(u, j + 1)]
18:         work := work ++ [(v, 0)]
19:         recurse := true
20:         break
21:     else if v ∈ stack then
22:         low[u] := min(low[u], disc[v])
23:     if recurse then continue
24:     if low[u] = disc[u] then
25:         result := u
26:         while true do
27:             v := stack[|stack| − 1]
28:             stack := stack[0 : |stack| − 1]
29:             if v == u then
30:                 break
31:             if v < result then
32:                 result := v
33:         return result
34:     if work ≠ [ ] then
35:         v := u
36:         (u, z) := work[|work| − 1]
37:         low[u] := min(low[u], low[v])
```