STRONGLY CONNECTED COMPONENTS

Andrea Corsini andrea.corsini@mail.polimi.it

Massimo Perini massimo.perini@mail.polimi.it

DEFINITION

- Let G be a directed graph. Suppose that for each pair of vertices v,w in G, there exist paths Pl:v->w and P2:w->v Then G is said to be **strongly connected.**
- **equivalence relation on the set of vertices**: two vertices v,w are path equivalent if there is a closed path v->v which contains w
  - Vi = distinct maximal equivalence classes generated with the relation.
  - Gi is the set of vertices and edges contained in a Vi class. Gi = strongly connected components of G

FIND SCC

- Let v and w be vertices in G which lie in the same strongly connected component. U is the highest common ancestor of v and w
  in a spanning forest of G generated by depth-first search -> u lies in the same strongly connected component
- Let C be a strongly connected component. The vertices of C define a subtree of a tree in the spanning forest of the graph. The root of this subtree is the root of the strongly connected component
- The problem of finding the strongly connected components reduces to the problem of finding the roots of the strongly connected components

TARJAN ALGORITHM

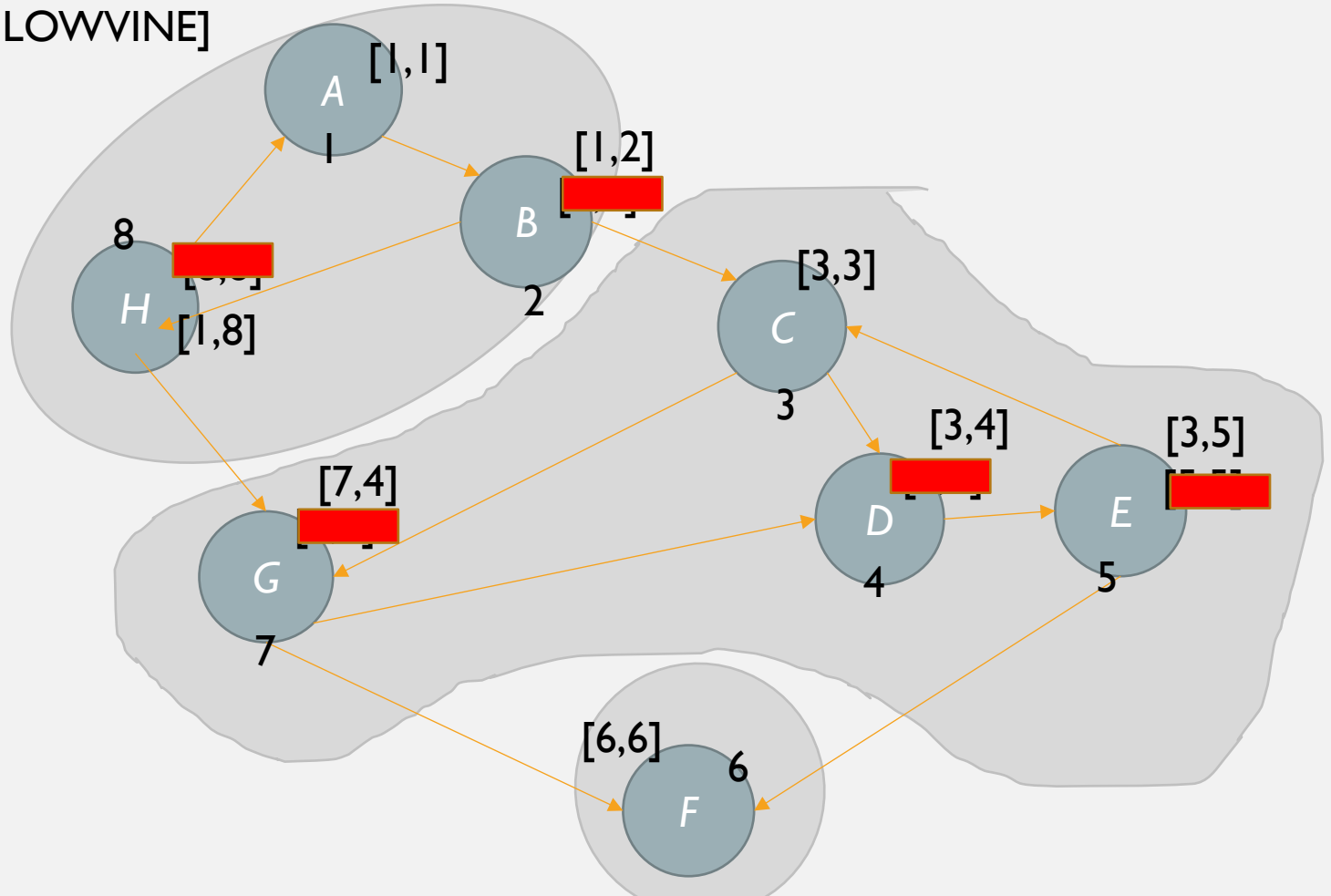
- It starts doing a DFS visit. The algorithm produces a spanning forest
- Every vertex is visited once from the graph and saved on a stack.
- Each vertex v is associated to the following properties:
  - NUMBER(v): the discovery index of the DFS visit
  - LOWPT(v): the node with the smallest NUMBER among the set {v} U {w | w is a node already visited but reachable from v or from one of his descendant nodes}
  - LOWVINE(v): the node with smallest NUMBER among set {v} U {w | w and v have common ancestor u & u and w belongs to same SCC, but not v}.

LOWPT(v) and LOWVINE(v) are initialised with NUMBER(v)

V is a root of SCC iff LOWPT(V) == NUMBER(V) and LOWVINE(V) == NUMBER(V)
Once a root v is found, nodes on the stack are visited and added to the new SCC until they have been discovered after root v.

TARJAN ALGORITHM

[LOWPT, LOWVINE]



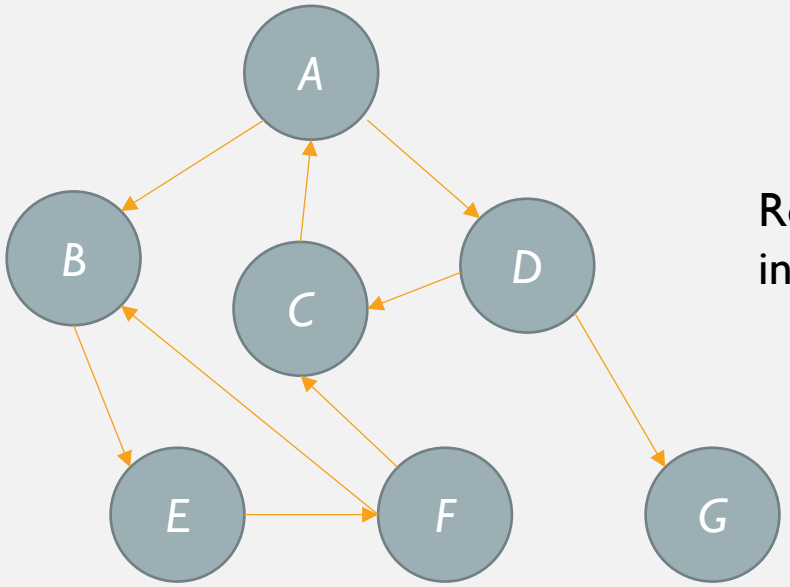
Stack

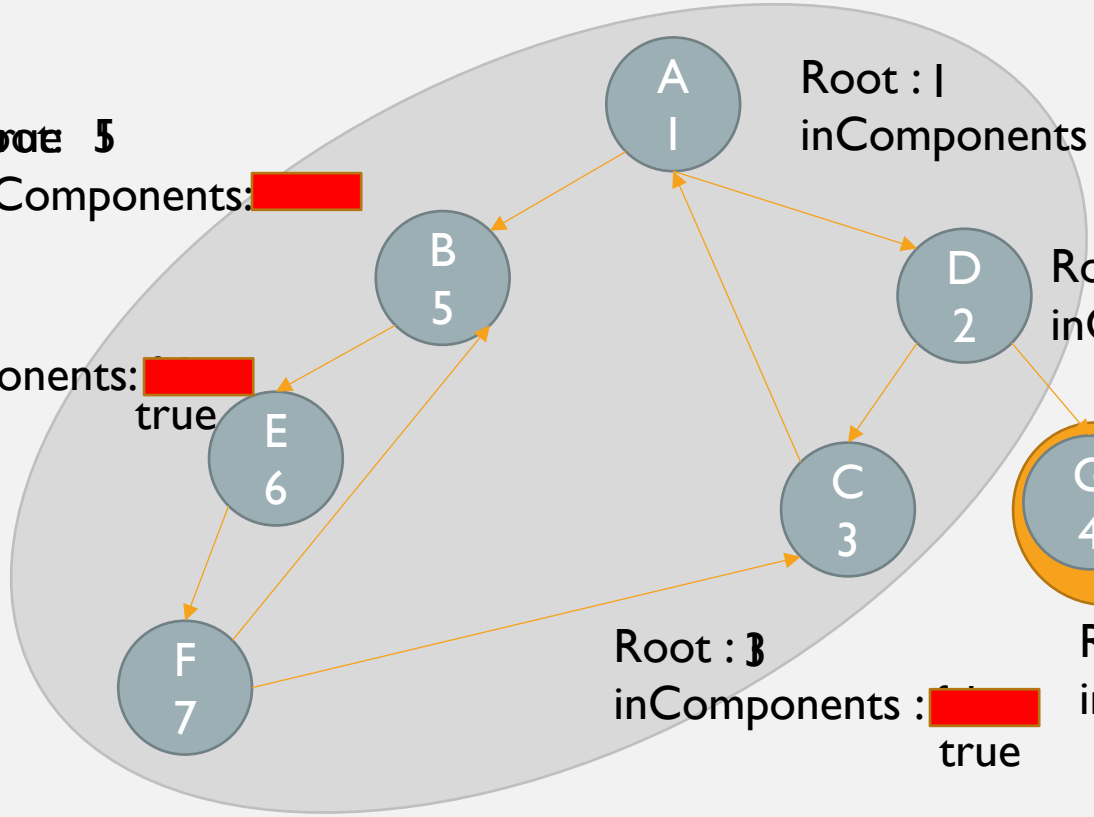
1
2
8 3
4
5
7 6

NUUTILA I

- Doesn't use stack when processing trivial components: stores a node on the stack only after it:
  - has processed all edges leaving the node
  - knows the node is not a root
- For each node we need to store:
  - Visitation index
  - Index of the root of the strongly connected component
  - If it is part of another strongly connected component
- We need to store the stack of nodes too

EXAMPLE





Root : 1
inComponents [redacted]
true

Root : 5
inComponents: [redacted]

Root : 2
inComponents : [redacted]
true

Root: 6
inComponents: [redacted]
true

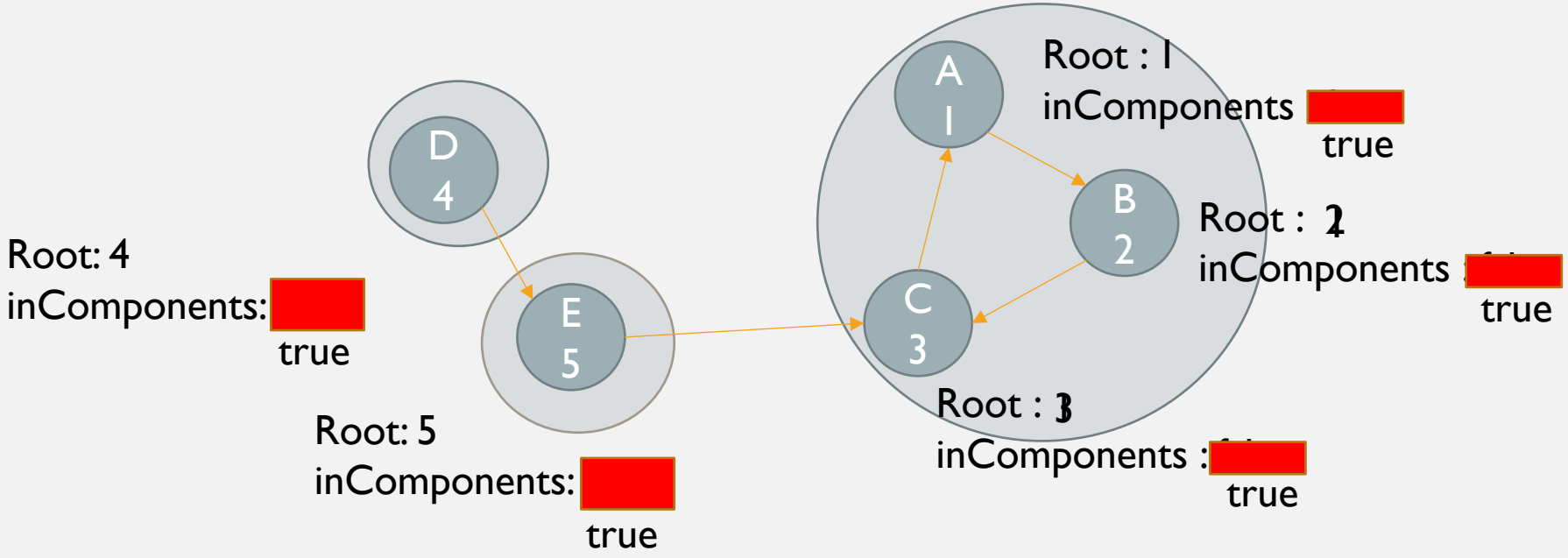
Root : 3
inComponents : [redacted]
true

Root: 4
inComponents: [redacted]
true

Root: 7
inComponents: [redacted]
true

Stack: B (5) E (6) F (7) D (2) C (3)

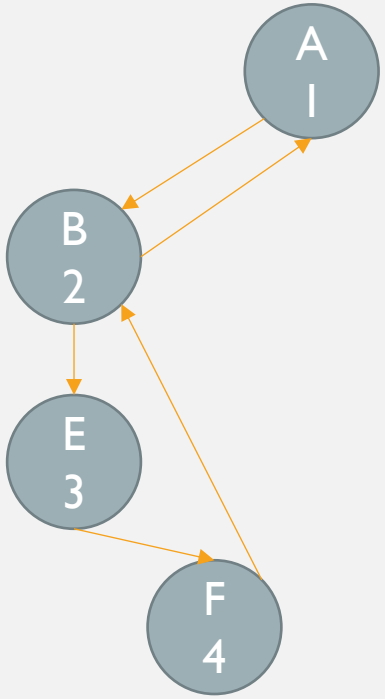
WHY INCOMPONENTS



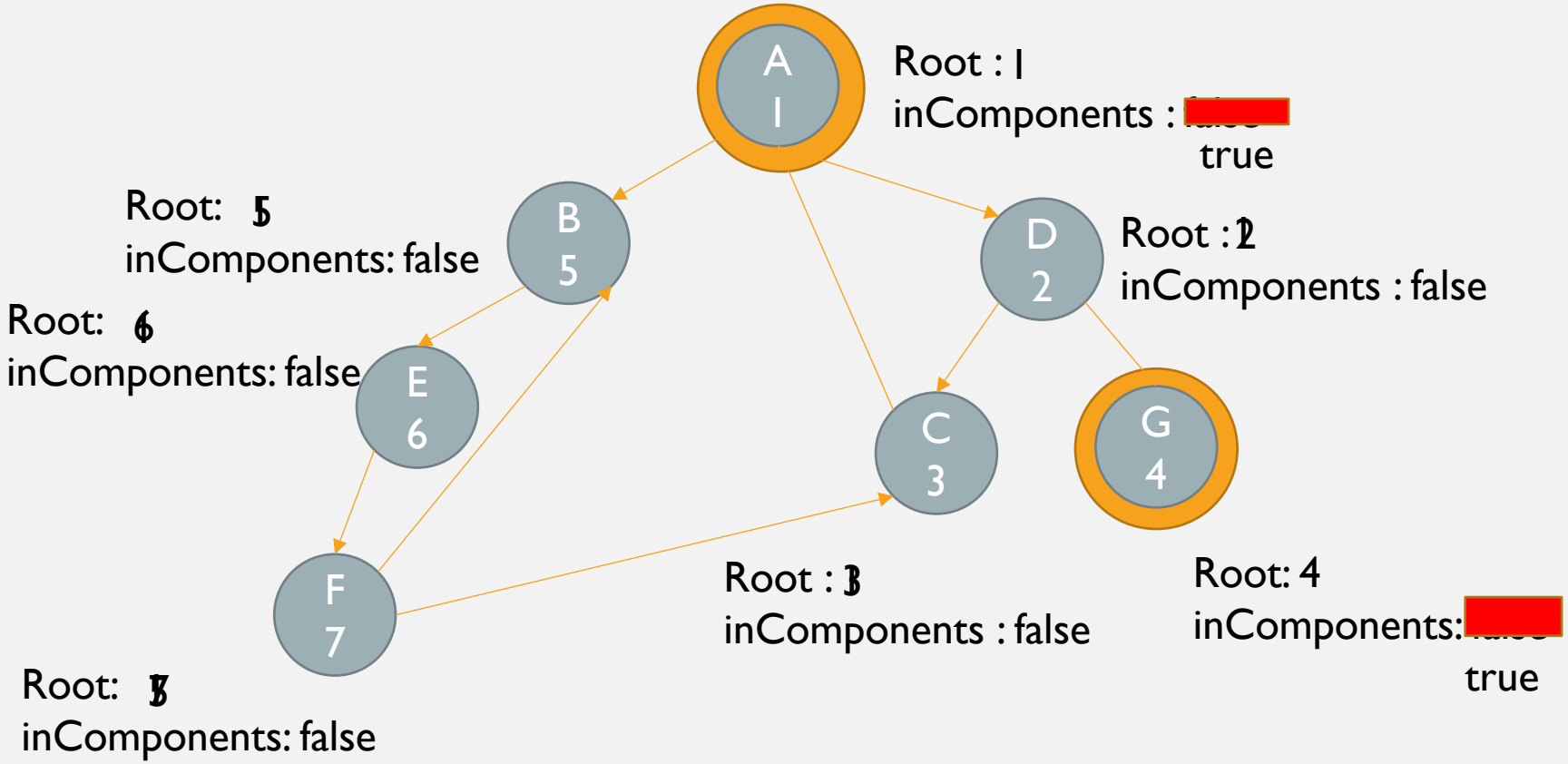
NUUTILA 2

- Useful in case we need only the detection of the component roots (for example to compute the number of Strongly Connected Components) and not the vertices inside them
- The child node w belongs to the same component as node v if root [v] = root[w]
- InComponent[w] = InComponent[root[w]]
- A node w is a final candidate root if exists y | w = root[y] after all the edges leaving y have been processed
- Only final candidate roots are stored on the stack -> nontrivial component candidate roots are stored on the stack

EXAMPLE



Here A and B may be stored in the stack

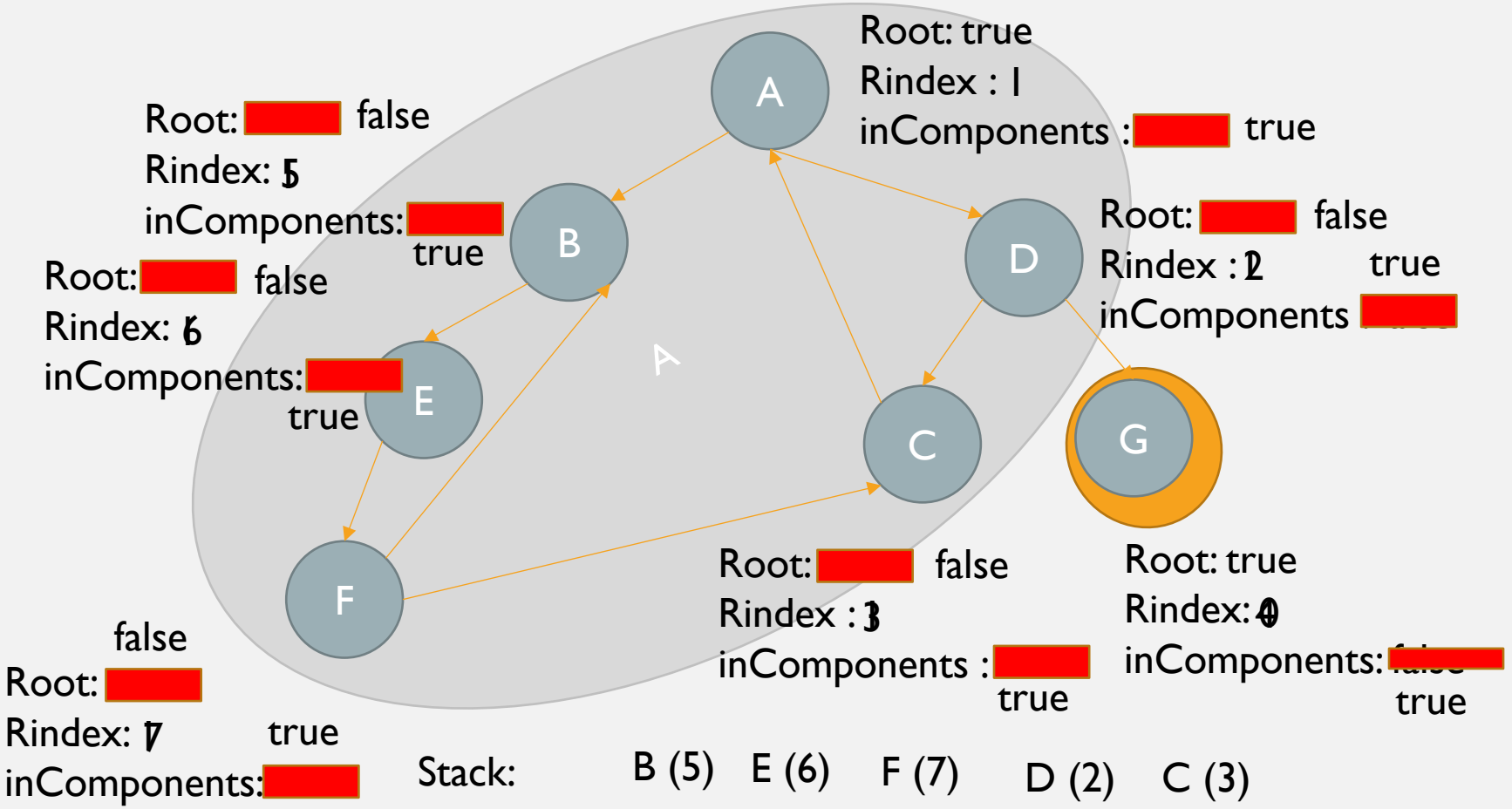


Stack: A (1)

PEARCE

- Pearce I
  - We can use only a single array that maps each vertex to the visitation index of its local root (instead of having an array for the indexes and another one for the roots)
  - If, after the visit of the following nodes, the local root of *v* is not v, push v onto a stack; otherwise, v is the root of a component (like Nuutila I)
  - the local root of a vertex *v* is *v* iff *rindex*[*v*] has not changed after visiting any successor
- Pearce II
  - remove visited array (visited iff rindex[v] >0)
  - remove inComponents: *rindex*[*v*] ≤ *rindex*[*w*] in the case that *w* has already been assigned to a component.
  - c = |*V* | − 1 and decremented by one whenever a vertex is assigned to a component. Another counter instead counts progressively.

EXAMPLE PEARCE 1



PERFORMANCES

- Tarjan Time complexity:
  - First iteration: O(vertices + edges)
  - Second iteration: O(vertices)
- Nuutila improves the second iteration only. Pearce has the same complexity
- Space complexity:
  - Tarjan: O(vertices)
  - Nuutila 1: O(vertices – Strongly Connected Components) – the root of each scc is never saved
  - Nuutila 2: same Space complexity of Nuutila 1 in worst case
  - Important difference if the graph does not fit in main memory

ALGORITHMS IMPLEMENTED

- Tarjan recursive (Tarjan's paper)
- Nuutila 1 recursive (Nuutila's paper) and iterative
- Nuutila 2 recursive (Nuutila's paper) and iterative
- Pearce 1 recursive (Pearce's paper)
- Pearce 2 recursive version (Pearce's paper) and iterative (Pearce's paper)

PROJECT STRUCTURE

- Code repository
  at https://github.com/massimoPerini/StronglyConnectedComponentsProj

- Code organized as CMake project:
  - ./libsccalgorithms: library that contains strong connectivity algorithms implementation and their tests: Tarjan, Nuutila1 (+ iterative), Nuutila2 (+ iterative), Pearce1, Pearce2 (+ iterative). Algorithms are generic with respect to the input graph, that must be trait of Boost graphs.
  - ./app: main application that runs algorithms on random generated graphs measuring the duration time of their execution or memory usages.
  - We adopted Travis CI

RUNNING TIME - 1

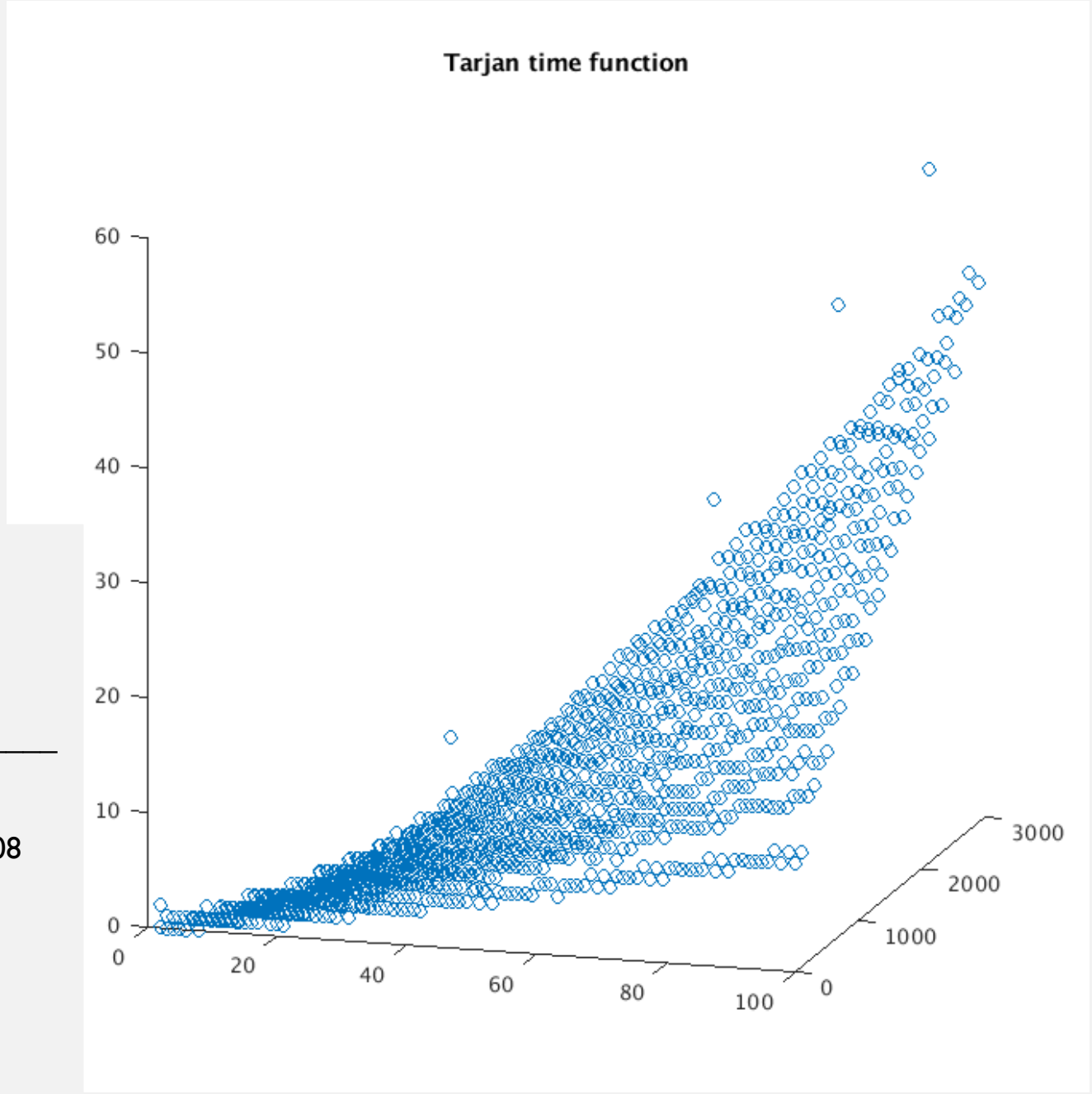
- O(V, E) for each of algorithm.
- The main program runs every (or those choosen) algorithm on a set of random graphs, generated with increasing number of vertecis and edges. Then a csv report is saved, containing the time duration of each algorithm for every graph.

Example of the report:

V	E	#SCCs	tarjan	Nuutila1 (recursive)	Nuutila2 (recursive)	Pearce1 (recursive)	Pearce2 (recursive)
38	101	15	5ms	4ms	4ms	5ms	3ms
...							
71	352	3	11ms	10ms	9ms	10ms	7ms
...							

RUNNING TIME - 2

- Running time for Tarjan algorithm
- Approximation of a plane (see from matlab .fig file)



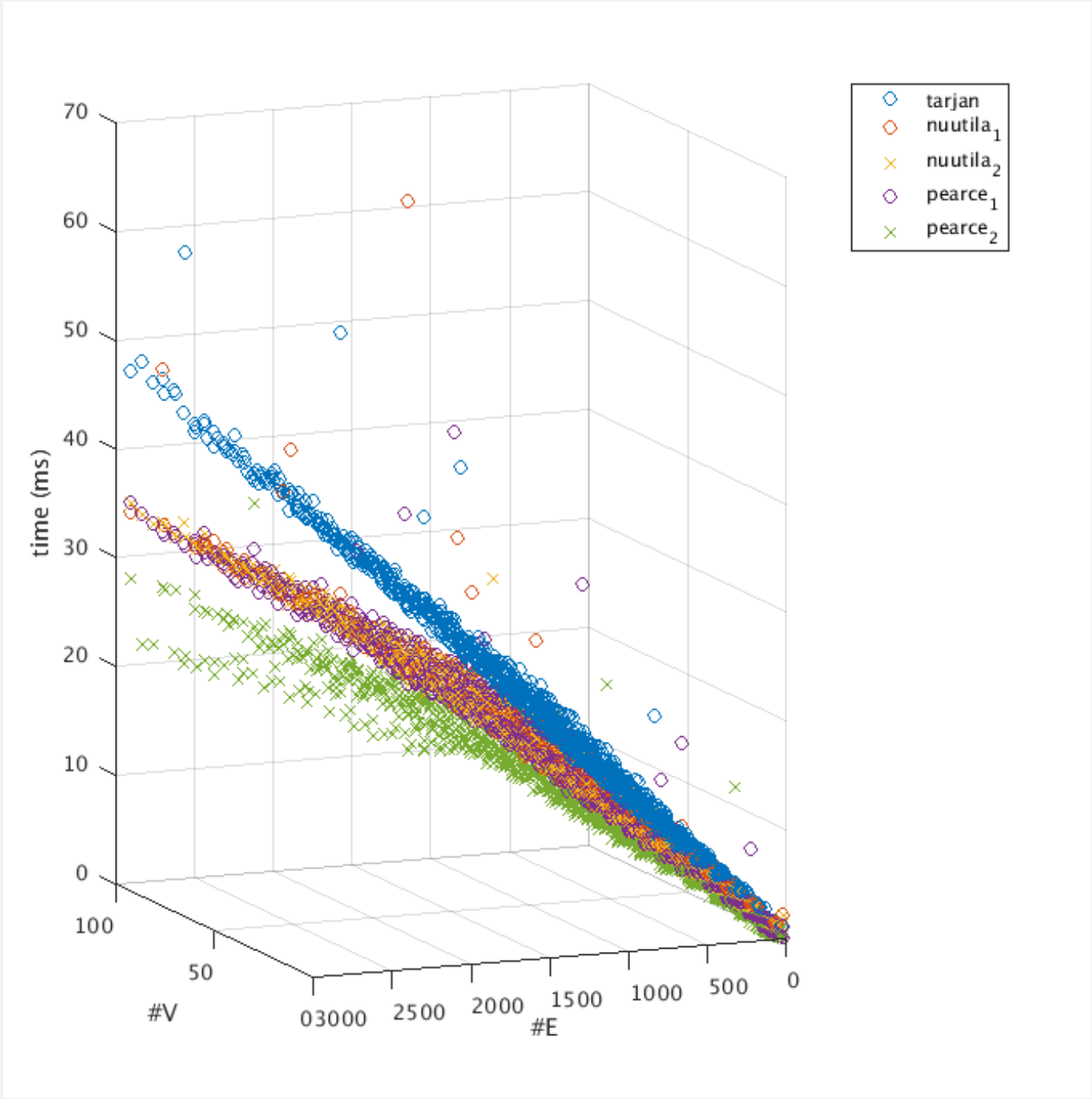
Linear regression model:
y ~ 1 + x1 + x2

Estimated Coefficients:

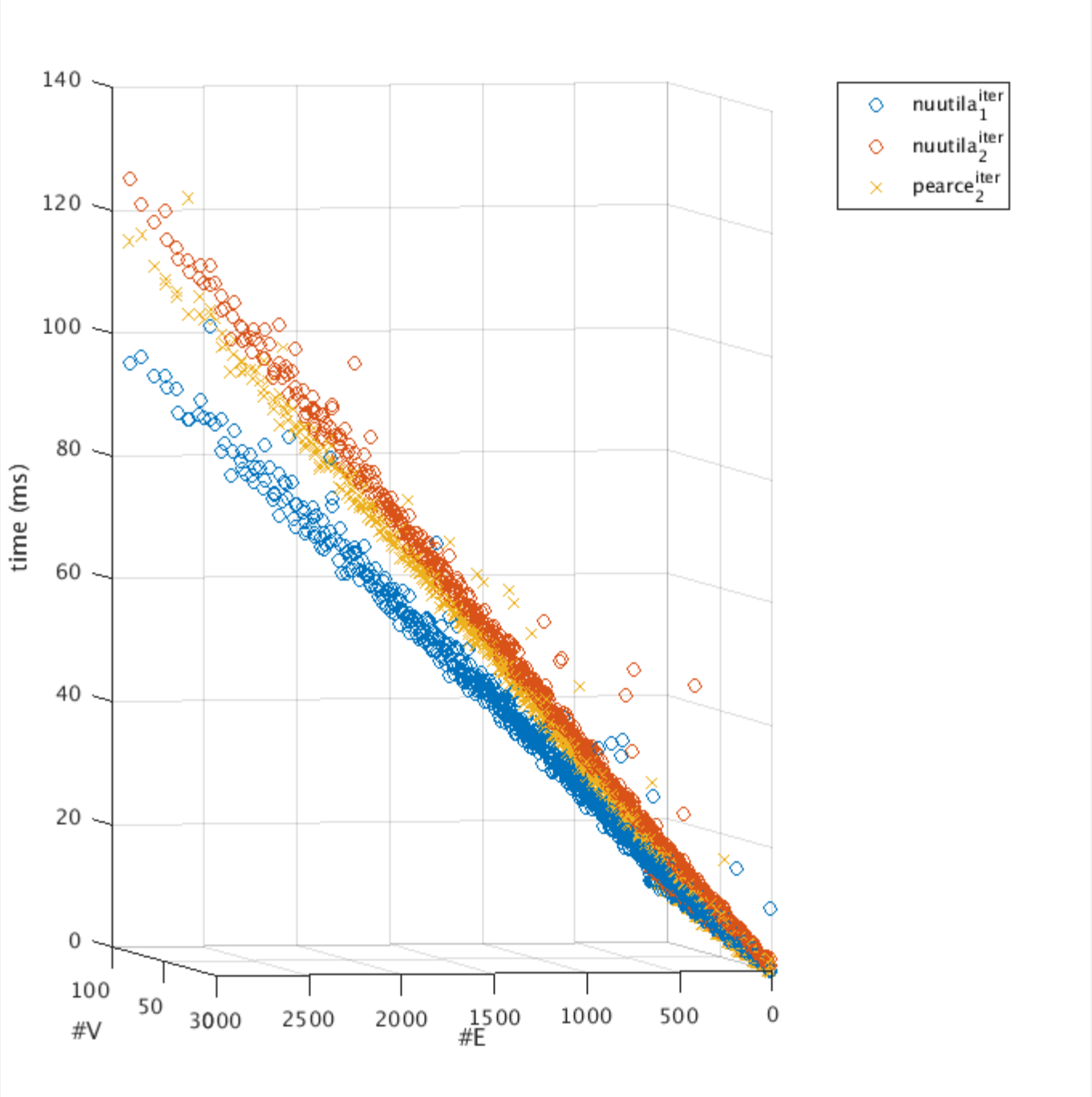
	Estimate	SE	tStat	pValue
1	-0.29004	0.053218	-5.45	5.8911e-08
x1	0.09821	0.00127	77.328	0
x2	0.013293	6.0676e-05	219.07	0

Number of observations: 1485
Root Mean Squared Error: 0.954
R-squared: 0.991, Adjusted R-Squared 0.991
F-statistic vs. constant model: 8.45e+04, p-value = 0

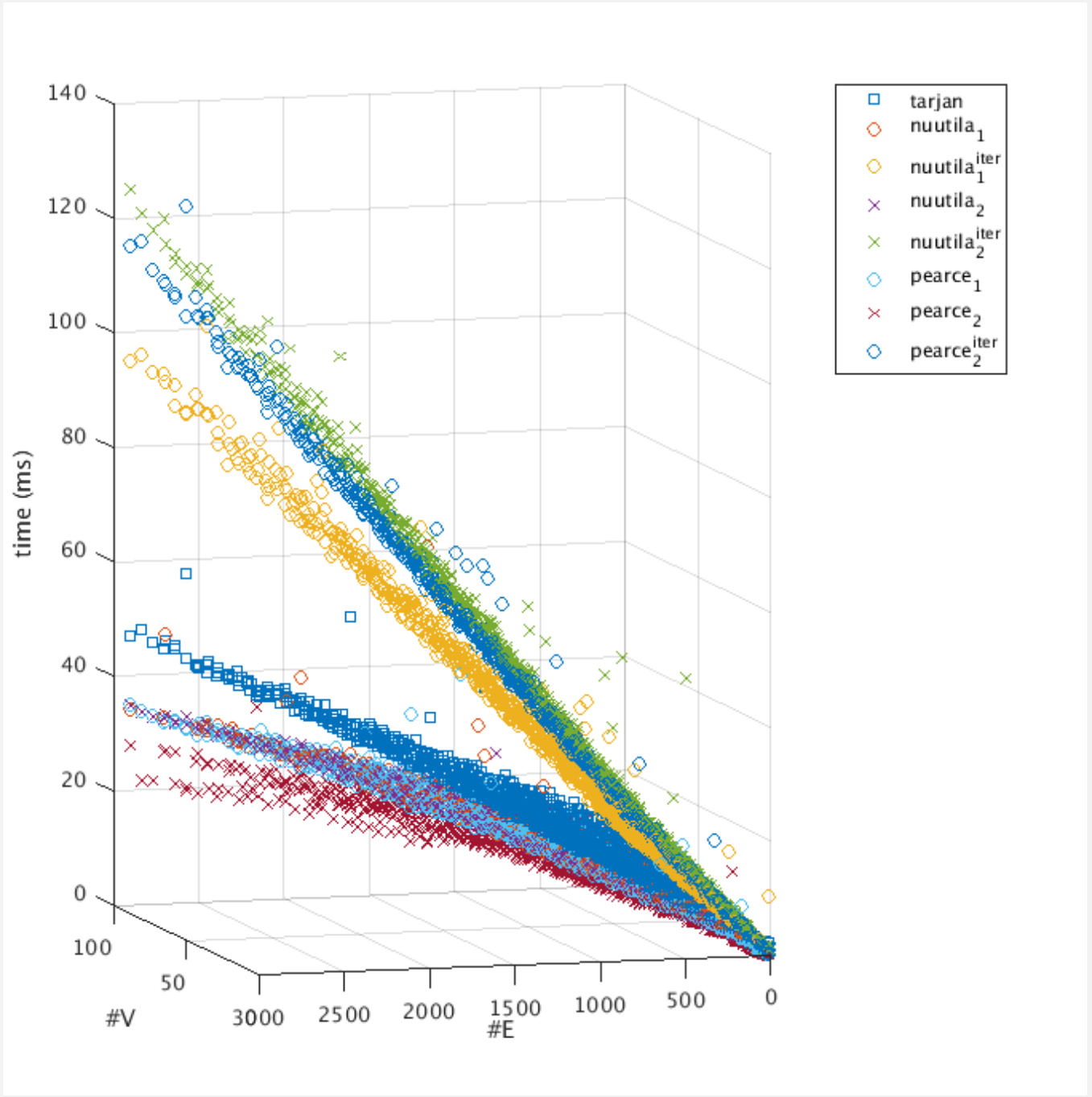
RUNNING TIME – 3 ALGORITHM COMPARISONS (RECURSIVE)



RUNNING TIME – 3 ALGORITHM COMPARISONS (ITERATIVE)



OVERALL


time (ms)
140
120
100
80
60
40
20
0
#V
100
50
#E
3000
2500
2000
1500
1000
500
0
tarjan
nuutila$_1$
nuutila$_1^{iter}$
nuutila$_2$
nuutila$_2^{iter}$
pearce$_1$
pearce$_2$
pearce$_2^{iter}$

VERTICES STACK CALLS - 1

- As seen, every algorithm use a stack of vertices where to save (some) of visited nodes.
- Using the code coverage, we can count the number of invocations of stack.push(_) to find the dimension of stack data structure used by algorithms.
- Tool gcov on debug build, runned on 100-vertices graph in two cases: few SCCs (12) and many SCCs (96)

- Checks the online README.md on the project reporistory to generate the original code coverage reports. Code coverage images are available in the repository

VERTICES STACK CALLS - 2

- *Coverage on file ./libsccalgorithms/include/scc_map_tarjan.hpp*
- *Tarjan always put every node once on the stack*

```
        :                 typename boost::property_traits<ComponentMap>::value_type & num_sccs)
        :         {
      100 :             boost::put(numbers, v, ++i);
      100 :             boost::put(lowpts, v, i);
      100 :             boost::put(lowvines, v, i);
      100 :             boost::put(is_on_stack, v, true);
      100 :             visited.push(v);
        :
      399 :             BGL_FORALL_ADJ_T(v, w, graph, Graph) {
      299 :                 if (boost::get(numbers, w) == NOT_ASSIGNED_NUMBER) {
        :                     // (v, w) is a tree arc;
       96 :                     tarjan_map_visit(graph, w, comp, lowpts, lowvines, numbers, is_on_stack, visited, i, num_sccs);
       96 :                     boost::put(lowpts, v,
       96 :                                std::min(boost::get(lowpts, v),
       96 :                                         boost::get(lowpts, w)));
       96 :                     boost::put(lowvines, v,
       96 :                                std::min(boost::get(lowvines, v),
       96 :                                         boost::get(lowvines, w)));
      498 :                 } else if (boost::get(lowpts, w) == boost::get(numbers, w) && // v is anchestor of w
      288 :                            boost::get(numbers, w) < boost::get(numbers, v) &&
      288 :                            boost::get(is_on_stack, w)) {
       71 :                     boost::put(lowpts, v,
        :                                std::min(
       71 :                                         boost::get(lowpts, v),
       71 :                                         boost::get(numbers, w)));
      341 :                 } else if (boost::get(numbers, w) < boost::get(numbers, v) &&
      209 :                            boost::get(is_on_stack, w)) {
       51 :                     boost::put(lowvines, v,
        :                                std::min(
       51 :                                         boost::get(lowvines, v),
       51 :                                         boost::get(numbers, w)));
        :                 }
        :             }
        :
      120 :             if(boost::get(lowpts, v) == boost::get(numbers, v) &&
       20 :                boost::get(lowvines, v) == boost::get(numbers, v)) {
        :                 // start new connected componet
        :
      212 :                 while (!visited.empty() && boost::get(numbers, visited.top()) >= boost::get(numbers, v)) {
      100 :                     boost::put(is_on_stack, visited.top(), false);
      100 :                     boost::put(comp, visited.top(), num_sccs);
      100 :                     visited.pop();
        :                 }
        :
       12 :                 ++num_sccs;
        :             }
      100 :         }; // tarjan_map_visit
        :
```

VERTICES STACK CALLS - 3

- *Coverage on file ./libsccalgorithms/include/scc_map_nuutila.hpp*
- *Nuutila 1 algorithm*
- *# vertices pushed = #V - # SCCs*
- *Pearce algorithms push the same number of vertices of Nuutila 1.*

```
52  100 :              if (root[index] == index) {
53      :                  //cout<<"NEW COMPONENT FOUND: "<<index;
54   12 :                  index_components[index] = true;
55      :
56   12 :                  boost::put(components, node, total_scc);
57      :
58   12 :                  if (!stack.empty()) {
59    6 :                      int old_index = stack.top();
60  180 :                      while (visitation_index[old_index] > visitation_index[index]) {
61      :                          //cout << " , " << old_index;
62   88 :                          index_components[old_index] = true;
63   88 :                          stack.pop();
64      :
65   88 :                          boost::put(components, old_index, total_scc);
66      :
67   88 :                          if (stack.empty()) {
68    1 :                              break;
69      :                          }
70   87 :                          old_index = stack.top();
71      :                      }
72      :                  }
73      :
74   12 :                  total_scc++;
75      :                  //cout<<endl;
76      :              } else {
77   88 :                  stack.push(index);  //oppure pushare il nodo direttamente e poi prend
78      :              }
79  100 :          }
80      :
81      :      }; // namespace nuutila_details
```

12 SCCs

```
59    7 :                      int old_index = stack.top();
60   13 :                      while (visitation_index[old_index] > visitation_index[index
61      :                          //cout << " , " << old_index;
62    4 :                          index_components[old_index] = true;
63    4 :                          stack.pop();
64      :
65    4 :                          boost::put(components, old_index, total_scc);
66      :
67    4 :                          if (stack.empty()) {
68    1 :                              break;
69      :                          }
70    3 :                          old_index = stack.top();
71      :                      }
72      :                  }
73      :
74   96 :                  total_scc++;
75      :                  //cout<<endl;
76      :              } else {
77    4 :                  stack.push(index);  //oppure pushare il nodo direttamente e poi
78      :              }
```

96 SCCs

VERTICES STACK CALLS - 4

- *Coverage on file ./libsccalgorithms/include/scc_map_nuutila.hpp*
- *Nuutila 2 algorithm*
- *# vertices pushed = #V - # SCCs in worst case*

```
151      100 :            if (root[index] == index) {
152       12 :                boost::put(components, root[index], total_scc);
153          :                //cout << endl << "NEW COMPONENT FOUND: " << index <<" IS THE ROOT (?)"<< endl;
154       12 :                if (!stack.empty() && visitation_index[stack.top()] >= visitation_index[index]) {
155       17 :                    while (!stack.empty() && visitation_index[stack.top()] >= visitation_index[index]) {
156        8 :                        index_components[stack.top()] = true;
157        8 :                        is_on_stack[stack.top()] = false;
158        8 :                        stack.pop();
159          :                    }
160          :                } else {
161       11 :                    index_components[index] = true;
162          :                }
163          :
164       12 :                total_scc++;
165          :            }
166       88 :            else if (!is_on_stack[root[index]]) {
167        8 :                stack.push(root[index]);
168        8 :                is_on_stack[root[index]] = true;
169          :            }
170      100 :        }
171          :
172          :    }; // namespace nuutila2_details
```

12 SCCs

```
156        1 :                        index_components[stack.top()] = true;
157        1 :                        is_on_stack[stack.top()] = false;
158        1 :                        stack.pop();
159          :                    }
160          :                } else {
161       95 :                    index_components[index] = true;
162          :                }
163          :
164       96 :                total_scc++;
165          :            }
166        4 :            else if (!is_on_stack[root[index]]) {
167        1 :                stack.push(root[index]);
168        1 :                is_on_stack[root[index]] = true;
169          :            }
170      100 :        }
171          :
172          :    }; // namespace nuutila2_details
```

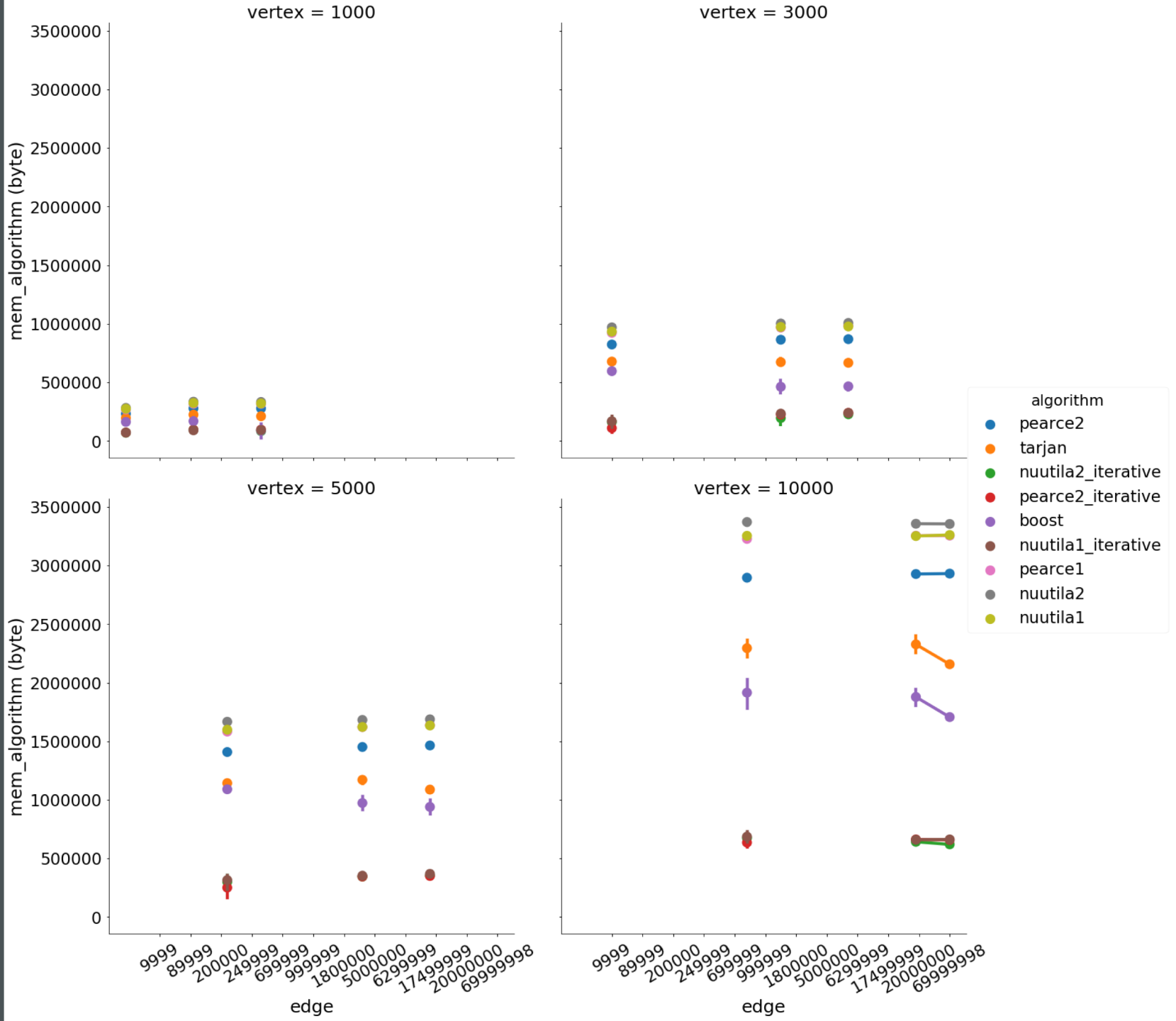
96 SCCs

RESIDENT SET SIZE

- A process's resident set size is the amount of memory that belongs to it and is currently present in RAM (in order to improve the accuracy we disabled the swap)
- Most OSes track the *current* and *peak* resident set sizes for each process. E.g. **getrusage()** returns ru_maxrss, the maximum resident size (on linux/OS X).
- RSS is a good approximation to maximum memory usage. It is valid in order to see memory allocation from OS perspective: OS allocates resources in chuncks, this means that not each app-request is a new resource.
- The program computes the maximum RSS of a scc algorithm

RSS - CHART

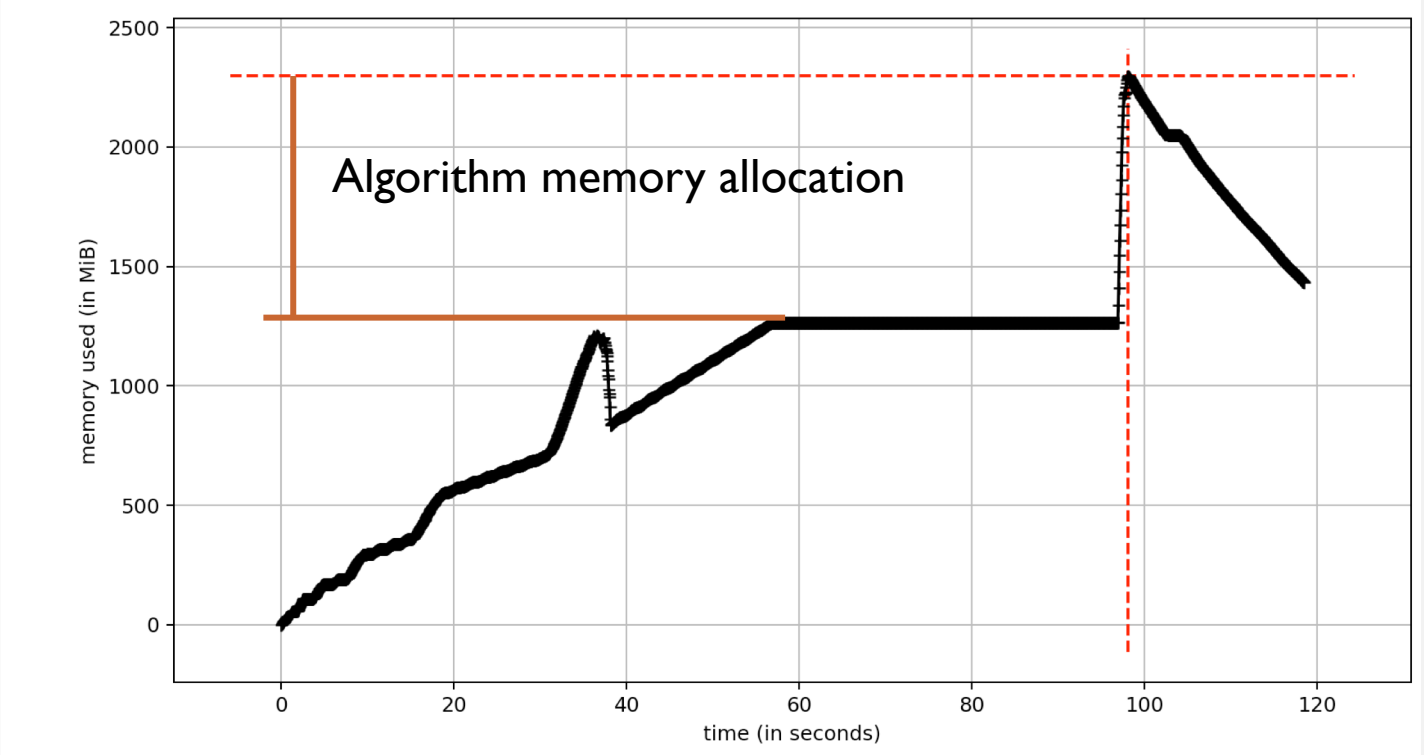
Average RSS (7 computations). Nuutila 2, Nuutila 1 and Pearce 1 have a similar memory usage. Tarjan and Boost tarjan have an even lower memory usage. The best algorithms, however, are the iterative implementations of Nuutila1, Nuutila 2 and Pearce 2. The allocated memory does not change among different number of edges.



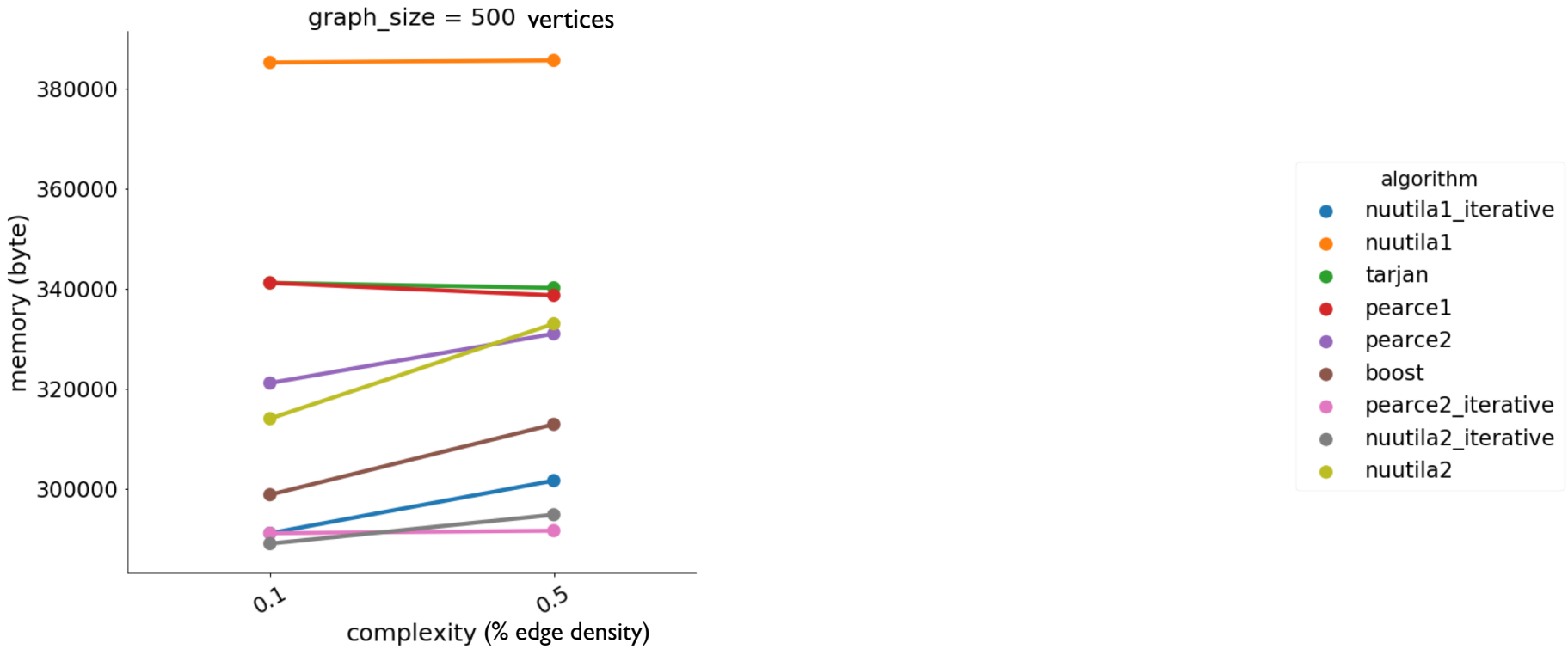
MEMORY PROFILING

We used mprof (a Python memory profiling tool) in order to have a fine grained analysis of the memory used by our code

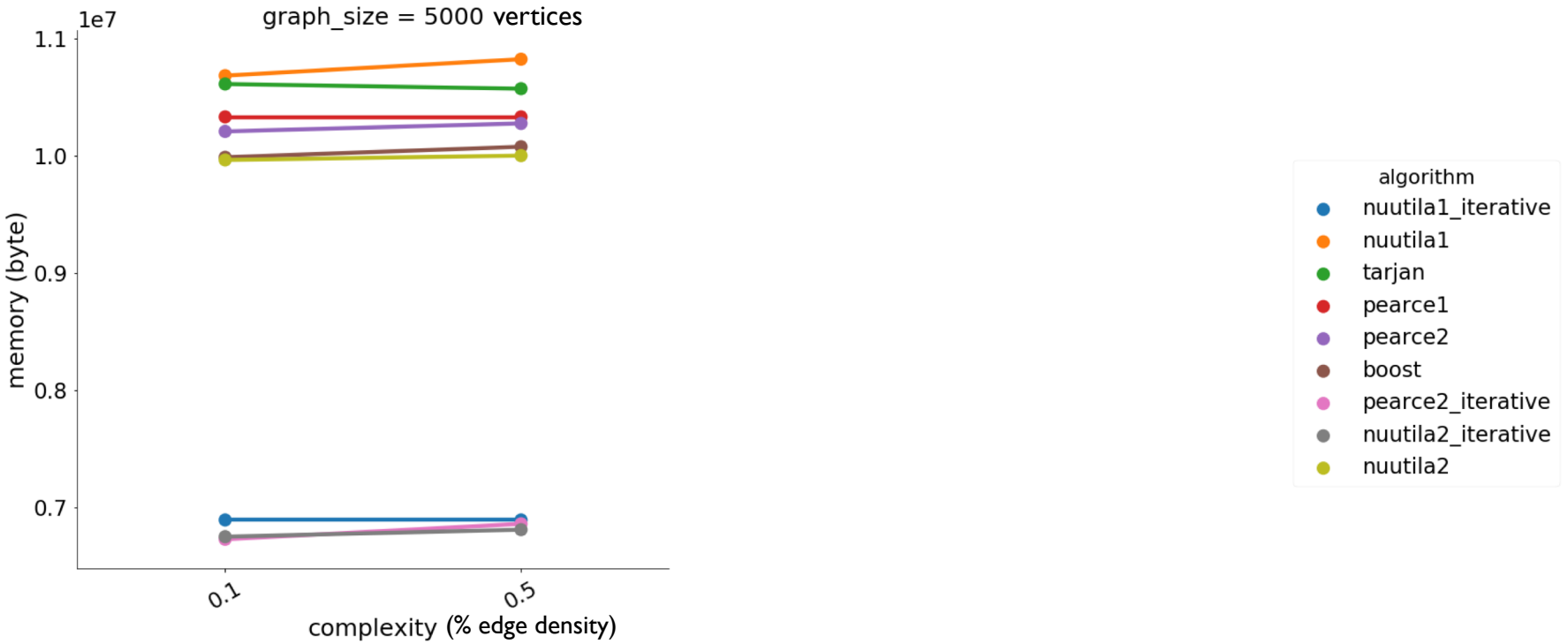
Example:



MEMORY PROFILING



MEMORY PROFILING



CONCLUSIONS

- All the algorithms implemented runs linearly with respect to number of vertices (V) and number of edges (E) of the input graphs.
  The iteratives algorithms are slower maybe because they actually emulates recursion: instead of a recursive call, the function returns and is called again with different data stored in the stack variables. Recursive versions, instead, return and starts visiting with data just ready.
- Not all the algorithms push the same number of vertices on their stack. As can be seen from coverage, Tarjan pushes every vertex, while the other algorithms improves the stack usage.
- All the implementation memory usages do not depend from the number of edges, so the memory is stable when only E variates, while, with the increment of V, it changes proportionally to it, as the theory requires.
  Iterative algorithms use less memory. We suppose it is due to the fact that they don't use the function calls on the memory stack