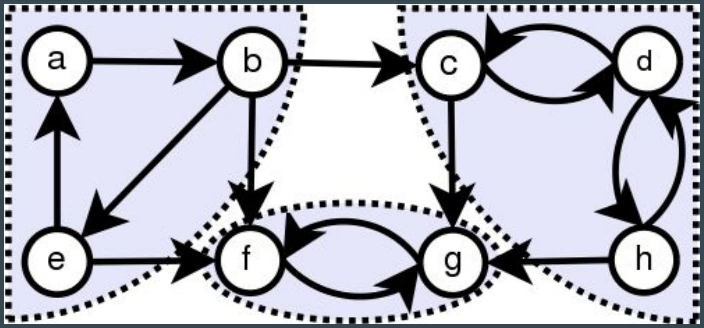
Advanced Algorithms project

A comparison of three variations of the same algorithm to find strongly connected components in graphs



Philippe Scorsolini
Emanuele Ricciardelli

Index

1. Context
2. Applications of SCC algorithms
3. Algorithms with examples
    a. Tarjan’s Algorithm
    b. Nuutila’s Algorithm
    c. Pearce’s Algorithm
    d. Differences
    e. A practical usage of Nuutila’s - Transitive Closure
4. Our implementations
    a. Algorithms
    b. Modifications
    c. Timing results
    d. Memory analysis
5. Conclusions

Context

The aim of all the algorithms is to find Strongly Connected Components (**SCC**) in directed graphs, they all achieve it by performing a Depth-First Search (**DFS**) over the graphs.

- **SCC** : A directed graph in which there is a **path between all pairs of vertices**. A strongly connected component of a directed graph is a maximal strongly connected subgraph.
- **Trivial SCC**: a SCC composed only by one vertex

The seminal work is presented by **Tarjan** in his “Depth-first search and linear graph algorithms” paper where the main approach is illustrated:

- “Corollary 10 : Let C be a strongly connected component in G. Then the vertices of C define a subtree of a tree in F, the spanning forest of G. The root of this subtree is called the **root of the strongly connected component C**.”
- **Spanning forest** generated by **DFS**
- The problem of **finding the strongly connected components of a graph G thus reduces to the problem of finding the roots of the strongly connected components.**

The **following works by Nuutila and Pearce** adopt the same DFS approach but achieve better memory usages by optimizing the usage of data structures (or removing some of them) and avoiding useless operations such as stacking “trivial” components.

Applications of SCC algorithms

“...can help find the **cyclic dependencies in a program**. Given a dependency graph G(directed graph), the Strongly Connected Components which contains more than one node of G (if any), gives the cycles in the graph.”

“Finding Strongly Connected Components in a **social network graph** can give you information about the **communities** of people that have formed on those networks. Social Networks can study the evolution of those communities and getting to know what community a person belongs to may help getting better ads for him.”

“... may be used to solve **2-satisfiability problems** (systems of Boolean variables with constraints on the values of pairs of variables): as Aspvall, Plass & Tarjan (1979) showed, a 2-satisfiability instance is unsatisfiable if and only if there is a variable v such that v and its complement are both contained in the same strongly connected component of the implication graph of the instance”

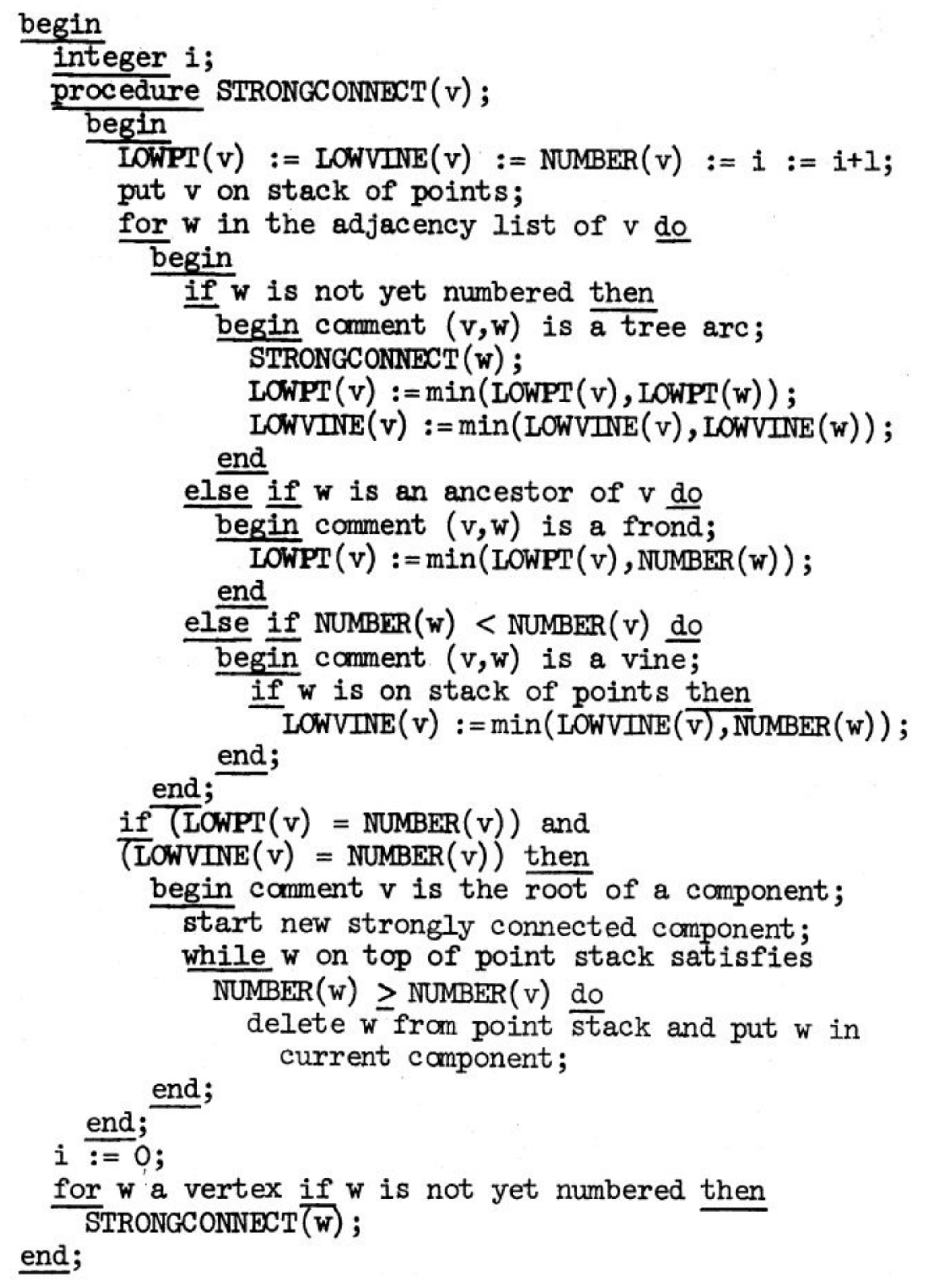
Tarjan’s Algorithm

Vine : edges running from one subtree to another in the tree.

Frond : edges running from descendants to ancestors in the tree.

Needed structures:

- **Number (int**[V]) : store vertices visit order
- **Lowpt (int**[V]) : store value of lowest parent reached through a frond
- **Lowvine (int**[V]) : store value of lowest node reached through a vine
- **Stack** : store vertices already visited but not yet in a component



```
begin
  integer i;
  procedure STRONGCONNECT(v);
    begin
      LOWPT(v) := LOWVINE(v) := NUMBER(v) := i := i+1;
      put v on stack of points;
      for w in the adjacency list of v do
        begin
          if w is not yet numbered then
            begin comment (v,w) is a tree arc;
              STRONGCONNECT(w);
              LOWPT(v) :=min(LOWPT(v),LOWPT(w));
              LOWVINE(v) :=min(LOWVINE(v),LOWVINE(w));
            end
          else if w is an ancestor of v do
            begin comment (v,w) is a frond;
              LOWPT(v) :=min(LOWPT(v),NUMBER(w));
            end
          else if NUMBER(w) < NUMBER(v) do
            begin comment (v,w) is a vine;
              if w is on stack of points then
                LOWVINE(v) :=min(LOWVINE(v),NUMBER(w));
            end;
        end;
      if (LOWPT(v) = NUMBER(v)) and
      (LOWVINE(v) = NUMBER(v)) then
        begin comment v is the root of a component;
          start new strongly connected component;
          while w on top of point stack satisfies
            NUMBER(w) ≥ NUMBER(v) do
              delete w from point stack and put w in
                current component;
        end;
    end;
  i := 0;
  for w a vertex if w is not yet numbered then
    STRONGCONNECT(w);
end;
```

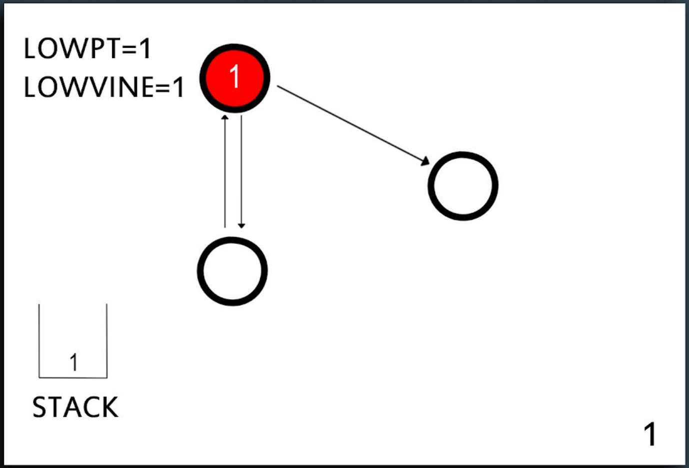
Tarjan’s Algorithm complexity

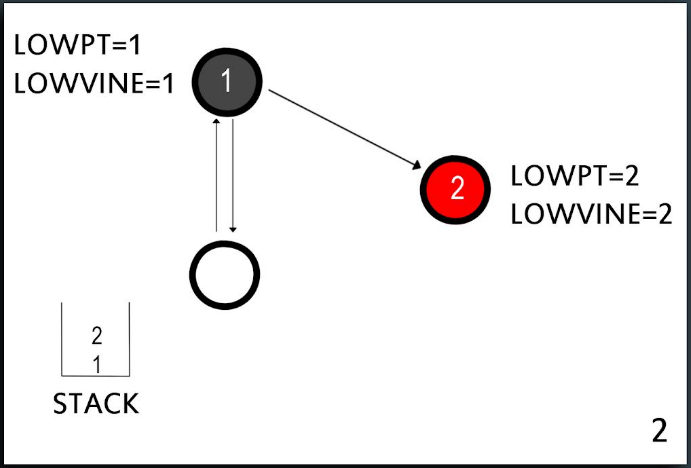
Time complexity : O(V + E)

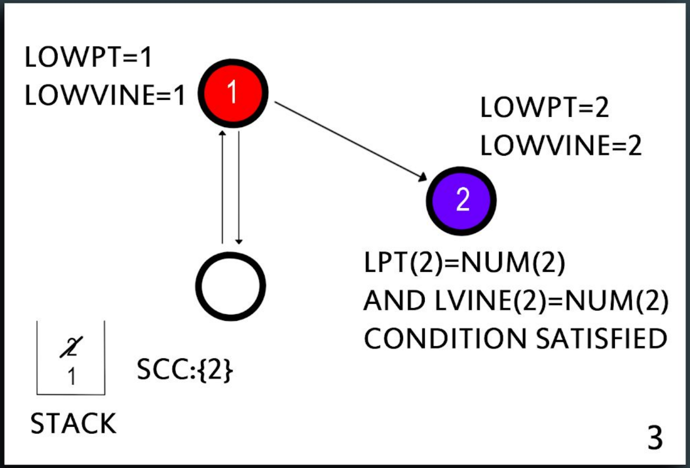
Space complexity :

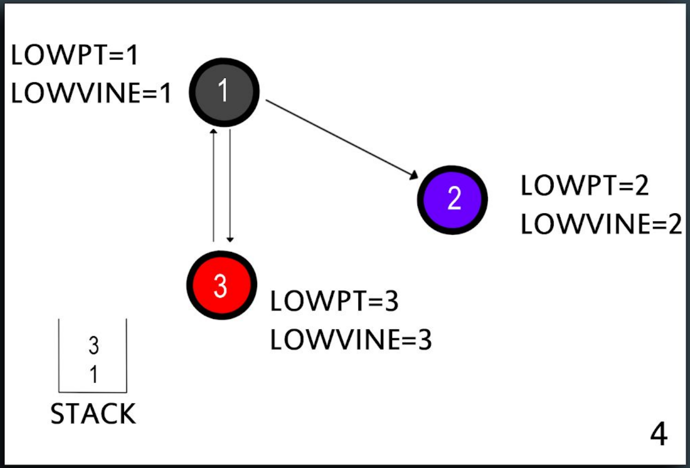
- O(V) from Tarjan’s paper
- v * (2 + 5w) (specified in Pearce’s paper)

Tarjan Example 1/2

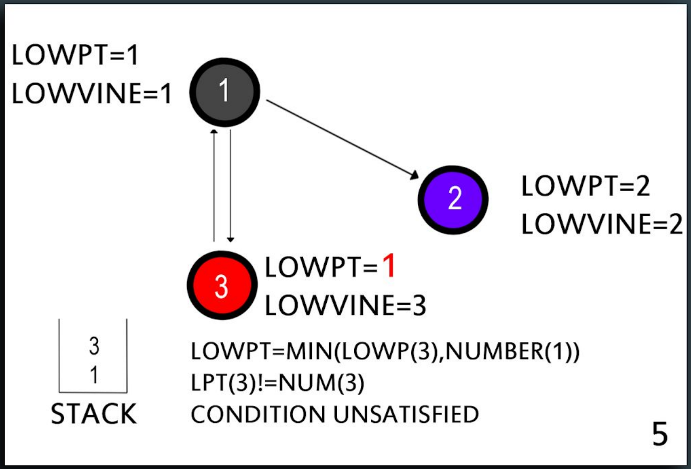








Tarjan Example 2/2



LOWPT=1
LOWVINE=1

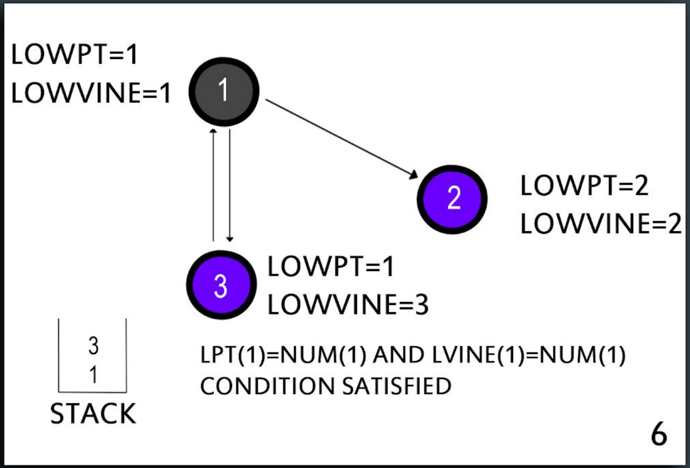
LOWPT=2
LOWVINE=2

LOWPT=1
LOWVINE=3

3
1
STACK

LOWPT=MIN(LOWP(3),NUMBER(1))
LPT(3)!=NUM(3)
CONDITION UNSATISFIED

5



LOWPT=1
LOWVINE=1

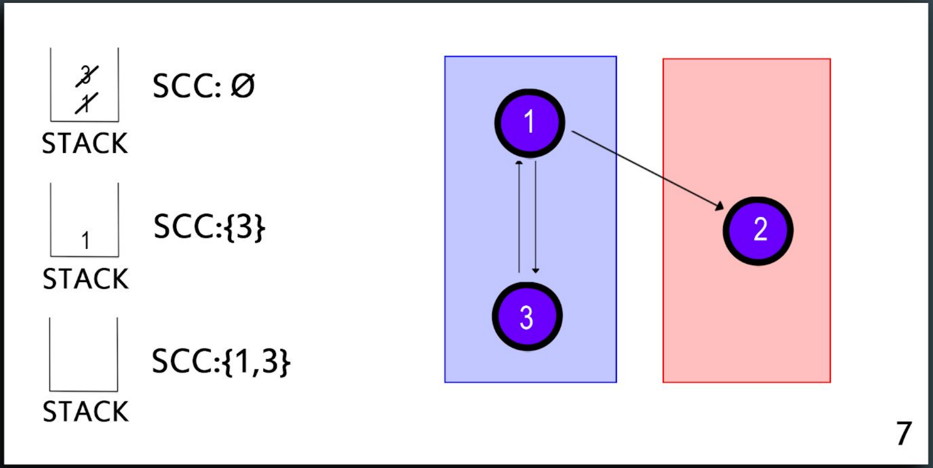
LOWPT=2
LOWVINE=2

LOWPT=1
LOWVINE=3

3
1
STACK

LPT(1)=NUM(1) AND LVINE(1)=NUM(1)
CONDITION SATISFIED

6



~~3~~
~~1~~
STACK

SCC: Ø

1
STACK

SCC:{3}

STACK

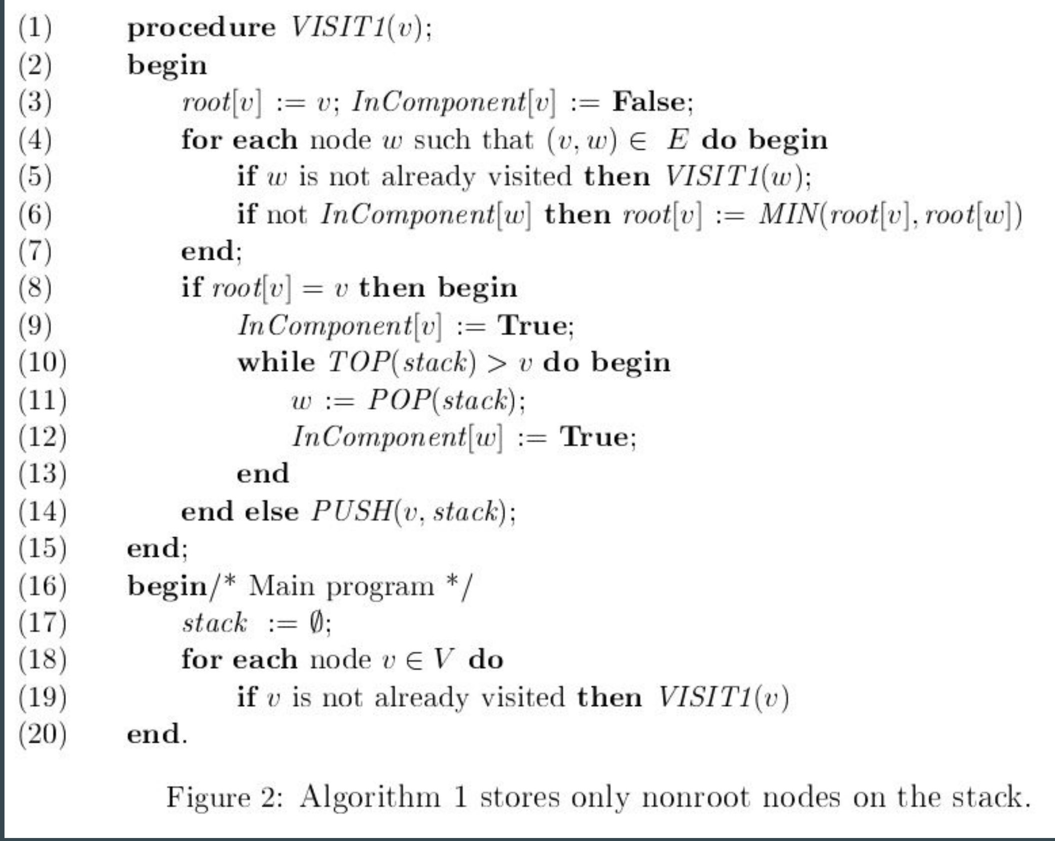
SCC:{1,3}

7

Nuutila

Needed structures:

- **Number** int[V] : the visiting order
- **InComponent** bool[V] : store if vertex is already in a component or not
- **Root** int[V] : store the root of the component where the vertex is
- **Stack** : store vertices until they are not assigned to a component, avoid pushing if knows it will be popped immediately



```
procedure $VISIT1(v)$;
begin
    $root[v]$ := $v$; $InComponent[v]$ := False;
    for each node $w$ such that $(v, w) \in E$ do begin
        if $w$ is not already visited then $VISIT1(w)$;
        if not $InComponent[w]$ then $root[v] := MIN(root[v], root[w])$
    end;
    if $root[v] = v$ then begin
        $InComponent[v]$ := True;
        while $TOP(stack) > v$ do begin
            $w := POP(stack)$;
            $InComponent[w]$ := True;
        end
    end else $PUSH(v, stack)$;
end;
begin/* Main program */
    $stack := \emptyset$;
    for each node $v \in V$ do
        if $v$ is not already visited then $VISIT1(v)$
end.
```

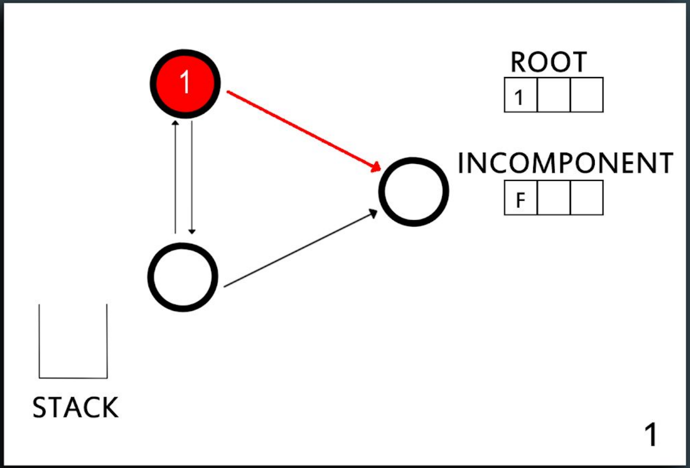
Figure 2: Algorithm 1 stores only nonroot nodes on the stack.

Time complexity : O(V + E)

Space complexity :

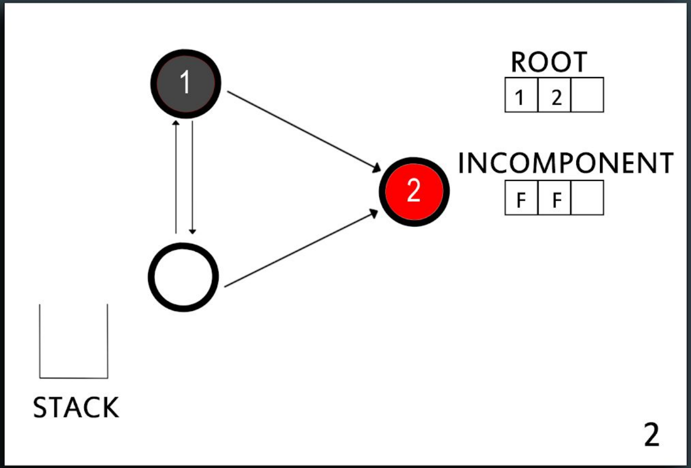
- O(V)
- v * (1 + 4w) (specified in Pearce's paper)

Nuutila Example 1/2



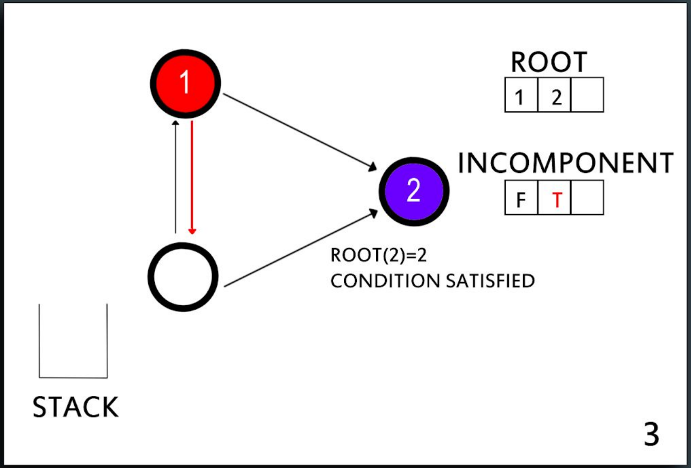
ROOT		
1		

INCOMPONENT		
F		



ROOT		
1	2	

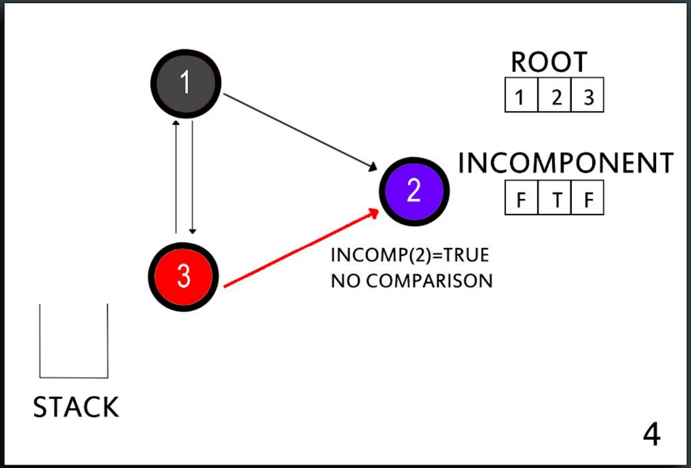
INCOMPONENT		
F	F	



ROOT		
1	2	

INCOMPONENT		
F	T	

ROOT(2)=2
CONDITION SATISFIED

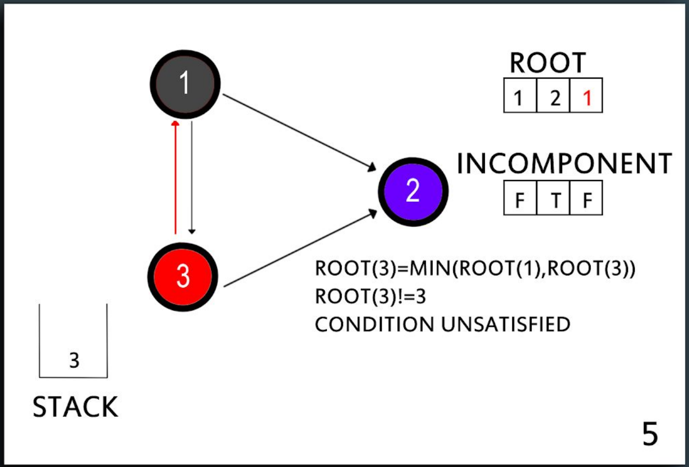


ROOT		
1	2	3

INCOMPONENT		
F	T	F

INCOMP(2)=TRUE
NO COMPARISON

Nuutila Example 2/2



ROOT

1	2	1

INCOMPONENT

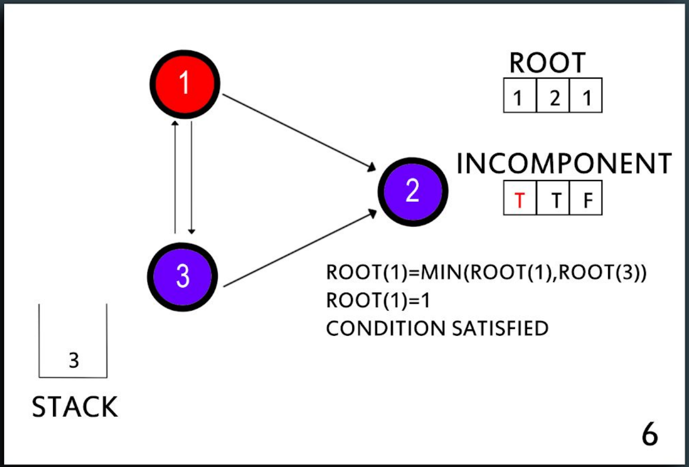
F	T	F

ROOT(3)=MIN(ROOT(1),ROOT(3))
ROOT(3)!=3
CONDITION UNSATISFIED

3

STACK

5



ROOT

1	2	1

INCOMPONENT

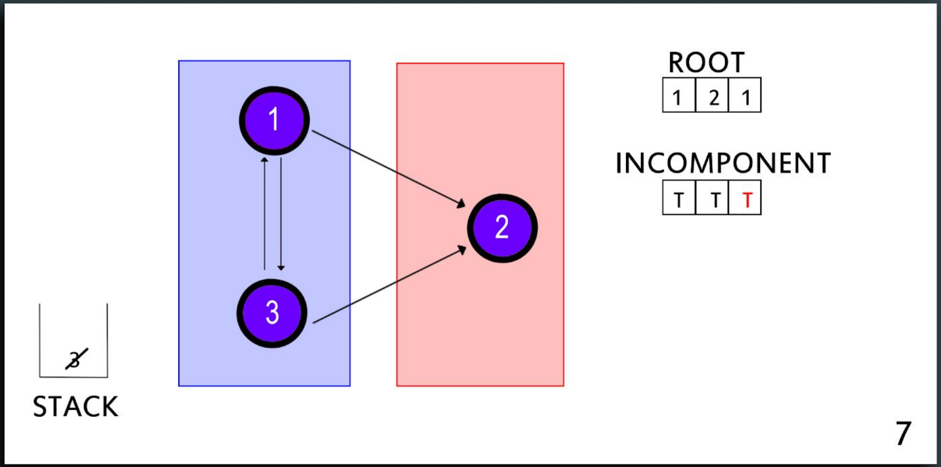
T	T	F

ROOT(1)=MIN(ROOT(1),ROOT(3))
ROOT(1)=1
CONDITION SATISFIED

3

STACK

6



ROOT

1	2	1

INCOMPONENT

T	T	T

~~3~~

STACK

7

Pearce

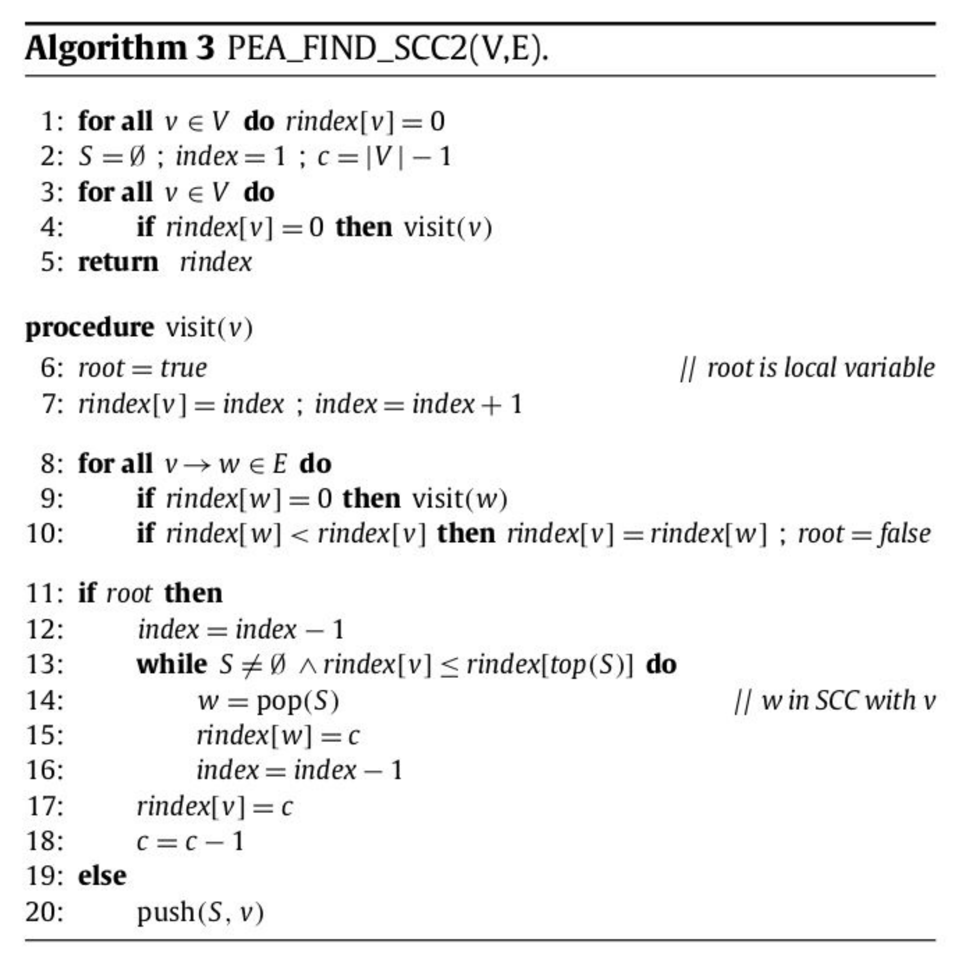
Needed structures :

- **Rindex** int[V] : used to check if not visited yet (0) and also the component the vertex belongs to.
- **Stack** : store vertices until not popped to be assigned to a component. Last one not assigned

Time Complexity : O (V + E)

Space Complexity (worst case) :

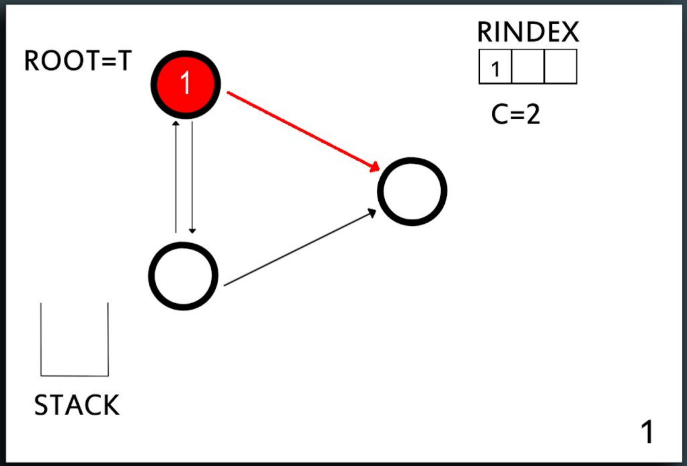
- Non-recursive version : v * (1 + 3w)

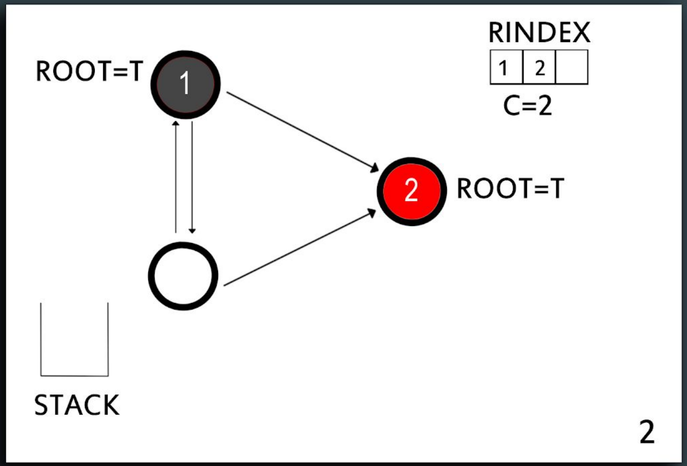


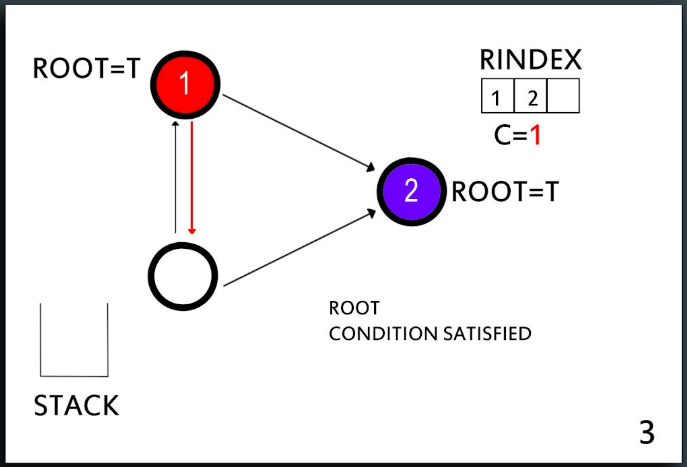
Algorithm 3 PEA_FIND_SCC2(V,E).

```
for all $v \in V$ do $rindex[v] = 0$
$S = \emptyset$ ; $index = 1$ ; $c = |V| - 1$
for all $v \in V$ do
    if $rindex[v] = 0$ then visit($v$)
return $rindex$

procedure visit($v$)
$root = true$                                        // root is local variable
$rindex[v] = index$ ; $index = index + 1$

for all $v \to w \in E$ do
    if $rindex[w] = 0$ then visit($w$)
    if $rindex[w] < rindex[v]$ then $rindex[v] = rindex[w]$ ; $root = false$

if $root$ then
    $index = index - 1$
    while $S \neq \emptyset \wedge rindex[v] \leq rindex[top(S)]$ do
        $w = $ pop($S$)                                  // w in SCC with v
        $rindex[w] = c$
        $index = index - 1$
    $rindex[v] = c$
    $c = c - 1$
else
    push($S$, $v$)
```

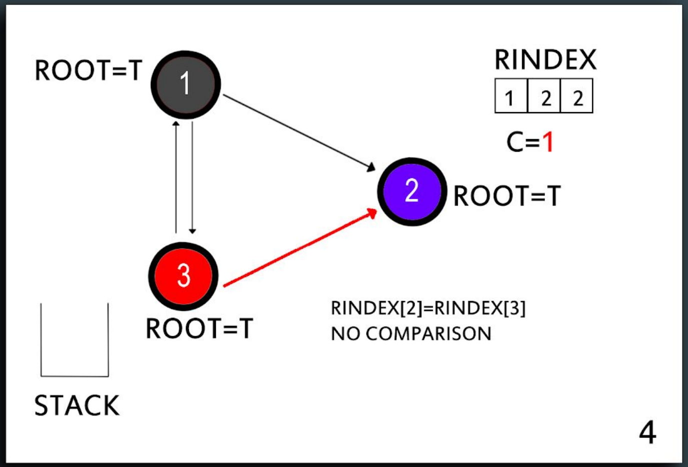
Pearce Example 1/2





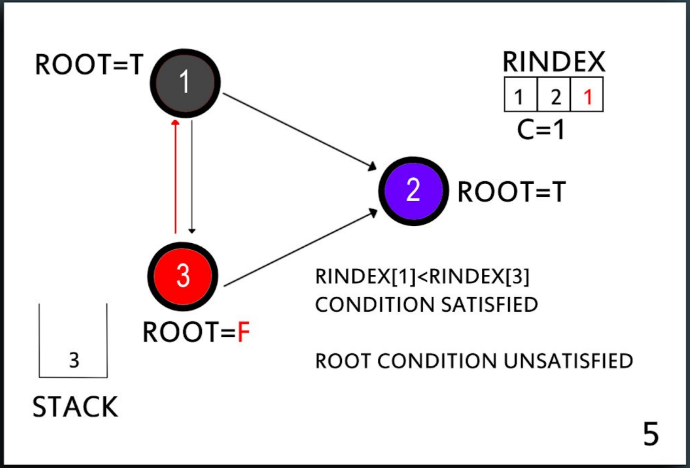


ROOT
CONDITION SATISFIED



RINDEX[2]=RINDEX[3]
NO COMPARISON

Pearce Example 2/2



ROOT=T

ROOT=T

ROOT=F

RINDEX

1	2	1

C=1

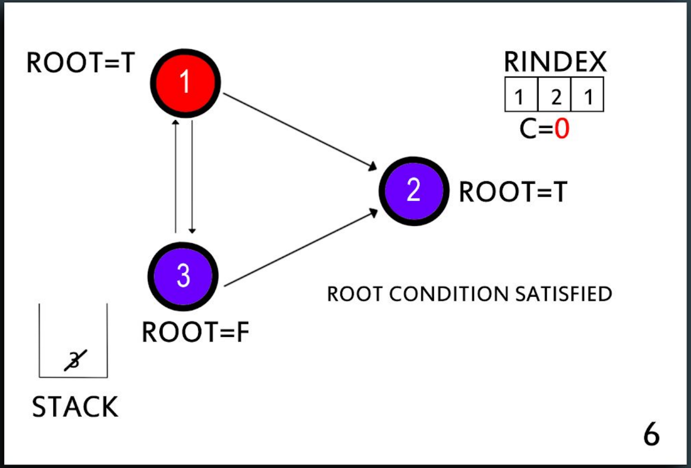
RINDEX[1]<RINDEX[3]
CONDITION SATISFIED

ROOT CONDITION UNSATISFIED

STACK

3

5



ROOT=T

ROOT=T

ROOT=F

RINDEX

1	2	1

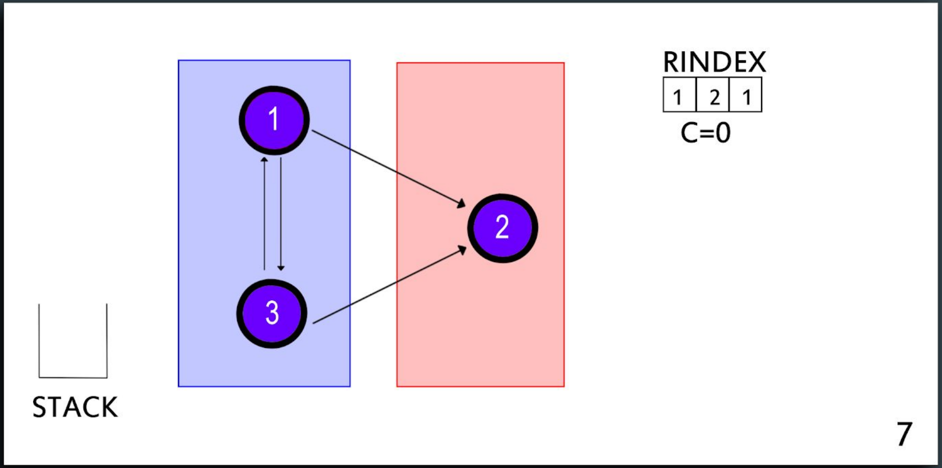
C=0

ROOT CONDITION SATISFIED

STACK

~~3~~

6



RINDEX

1	2	1

C=0

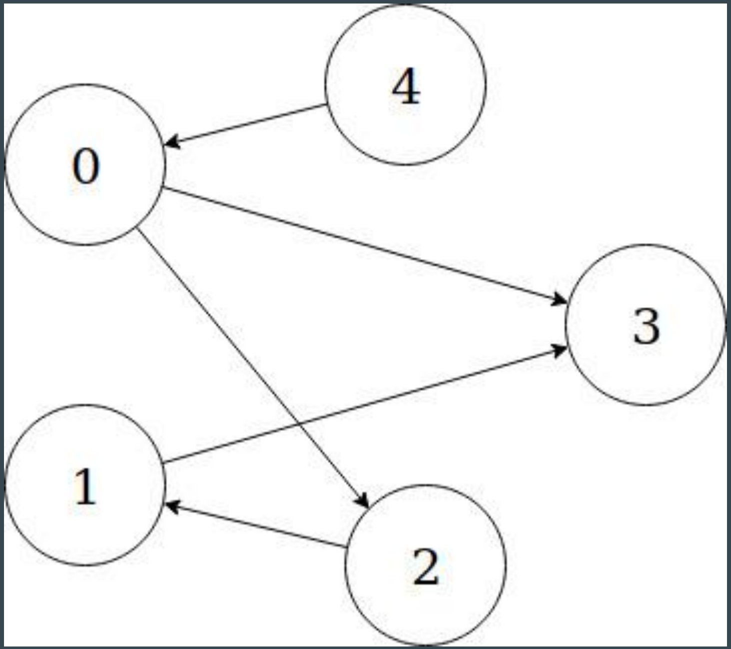
STACK

7

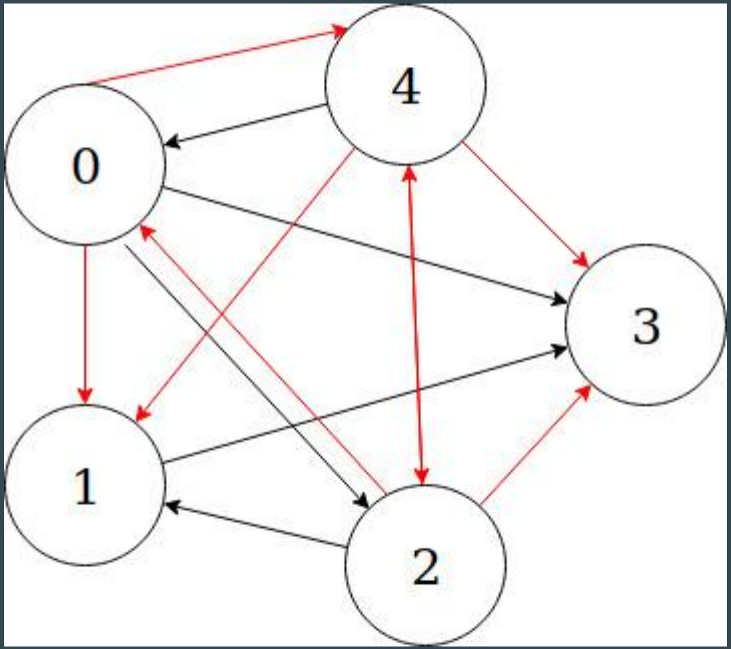
Differences

- **Space** efficiency :
    - **Stack** usage :
        - **Tarjan's** pushes on the stack the vertices before checking if it's necessary or not, pushing in the worst case all the vertices on the stack (long chain of vertices).
        - **Nuutila's** avoids pushing on the stack nodes belonging to trivial components (single node components), achieving better performance when the graphs are sparse and pushes on the stack only during the backtracking phase, when it's sure it's not part of a trivial component. This way in case of a long chain no vertices at all are pushed on the stack.
        - **Pearce's** further improves Nuutila's approach using fewer auxiliary structures, rindex is used for multiple purposes ("already visited?", "in component?")..
- **Time** complexity remains the same due to the **Depth First Search** all three the algorithms perform.

A practical usage of Nuutila’s - Transitive Closure



Original Graph



Transitive closure of the original

A **transitive closure** of a graph is the one in which there exists an edge starting from vertex i to vertex j if and only if vertex j “is reachable” from vertex i in the original graph, that is there is a (single or multi) hop path from vertex i to j.

Transitive Closure

The main idea of Nuutila is to compute the **successor nodes,**, that is the ones reachable via a path, only for the final candidate roots.

Taken a SCC called C, the set of successors **succ[C]** contains all the vertices in C and the candidate roots' successors of SCC adjacent to C, that is reachable from a vertex v in C.

These are the benefits respect to other algorithms for transitive closure based on SCC:

- All the outgoing edges are scanned only once (during the Depth-First Search) thanks to usage of suitable data structures
- The successors set is computed only for the final roots and not for all the vertices in the graph

Our implementations

- **Languages** :
  - Algorithms implementations : **C++** (Boost Graph Library)
  - Visualization and results manipulation : **Python** (jupyter notebooks, pandas, matplotlib, plotly)
  - Testing automation and benchmarking : **Bash** scripts and some **C++** helper functions
- **Tools** :
  - Time measures : **boost::timer::auto_cpu_timer**
  - Memory profiling :
    - **Valgrind** (massif)
    - **Heaptrack**
- **We have implemented :**
  - Tarjan's STRONGCONNECT
  - Nuutila's :
    - Visit1
    - Transitive Closure example
  - Pearce's :
    - PEA_FIND_SCC2 (recursive)
    - PEA_FIND_SCC3 (non-recursive)

Our implementations

TarjanClass, **NuutilaClass**, **PearceClass (recursive)** that wrap all the needed structures and expose a method (**tarjan_scc(), nuutila_scc(), pearce_scc()**) returning a pointer to the structure used to associate components and vertices (components / root / rindex)

To check the correctness of our implementation we have compared them against the results given by the **boost::strong_components** function.

We have also implemented in the **PearceNR** class**,** the **non recursive** version of the algorithm proposed by Pearce, and in the **TransitiveClosure** class, the example given as an application of Nuutila’s algorithm for computing the transitive closure of a given graph.

Modifications to the proposed algorithms

In order to effectively implement them, we had to add some data structures:

- Tarjan :
    - **sm** (std::vector<bool>(V)) : used to check if a vertex is in the stack or not in linear time
        - std::vector<bool>(V) uses 1 bit per element, instead bool[V] uses 1 Byte per element, wasting 8 times more space than a vector
    - **ancestor** (std::vector<bool>(V)) : used to check if a vertex is an ancestor of the current one in linear time
    - **comp** (std::vector<int>(V)) : used to store the component each vertex belongs to (not strictly specified)
- Nuutila:
    - No additions
- Pearce:
    - No additions

Timing results

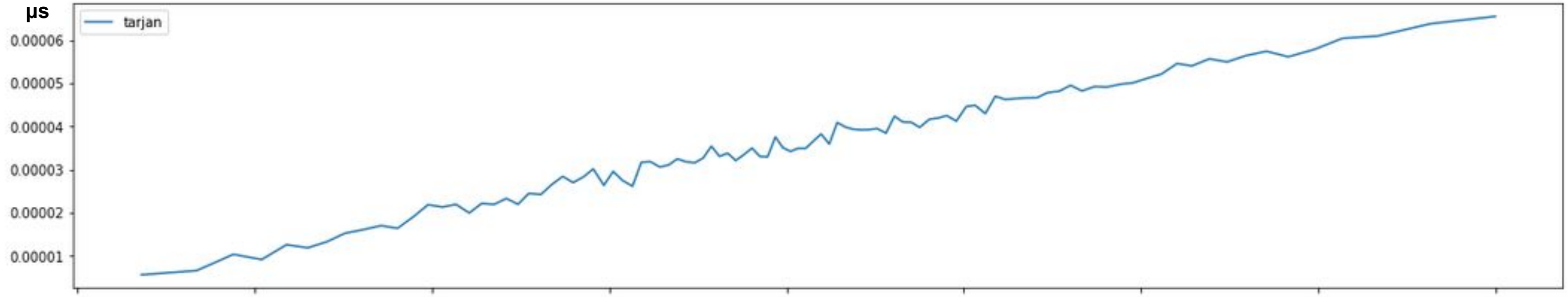
Generated graphs with 0 to 1000 vertices and 0 to 1000 edges using **boost::erdos_renyi_iterator** (setS as outgoing edge container in order to avoid repeated edges)

We then have :

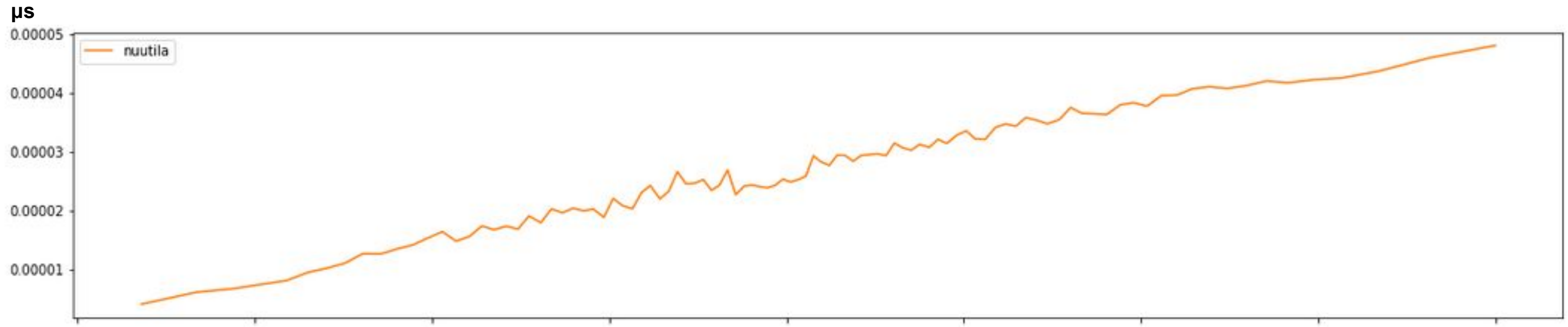
1. run the executable over the generated graphs (multiple runs)
2. collected the results of **boost::timer::auto_cpu_time**
3. averaged them
4. Plotted both in (V+E,t) space and in (V,E,t)

And as can be seen from the plots we have achieved a behaviour according to the specified theoretical complexity.

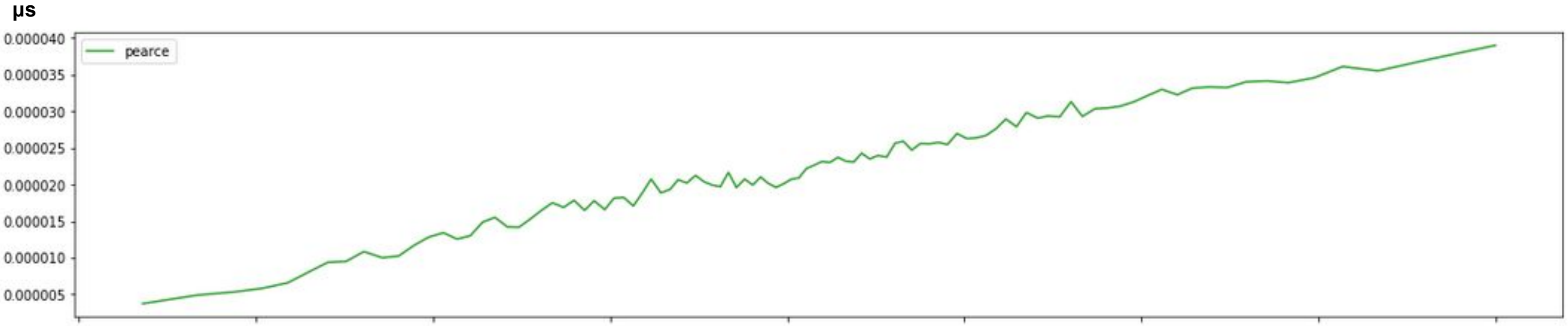
Tarjan



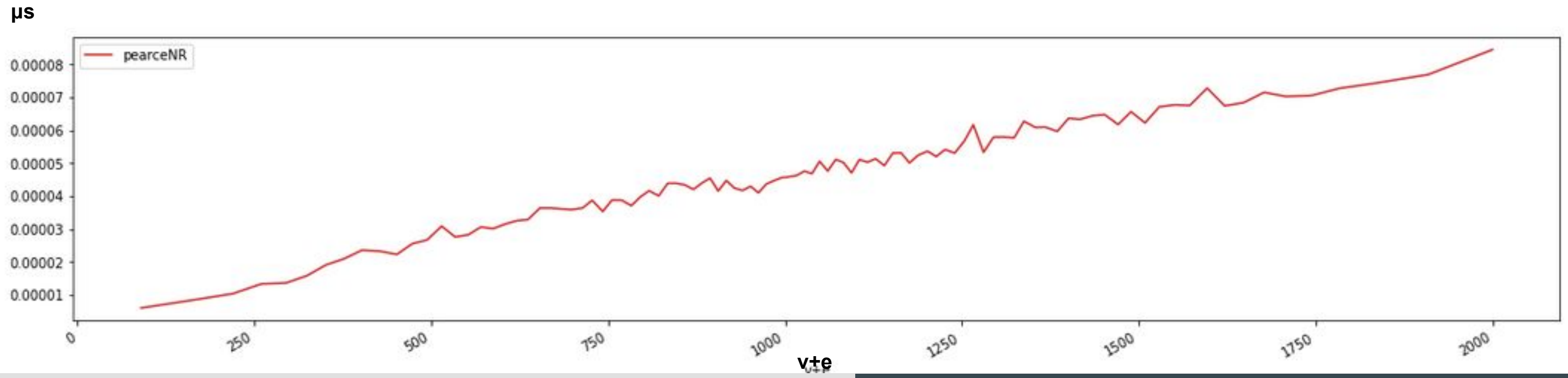
Nuutila

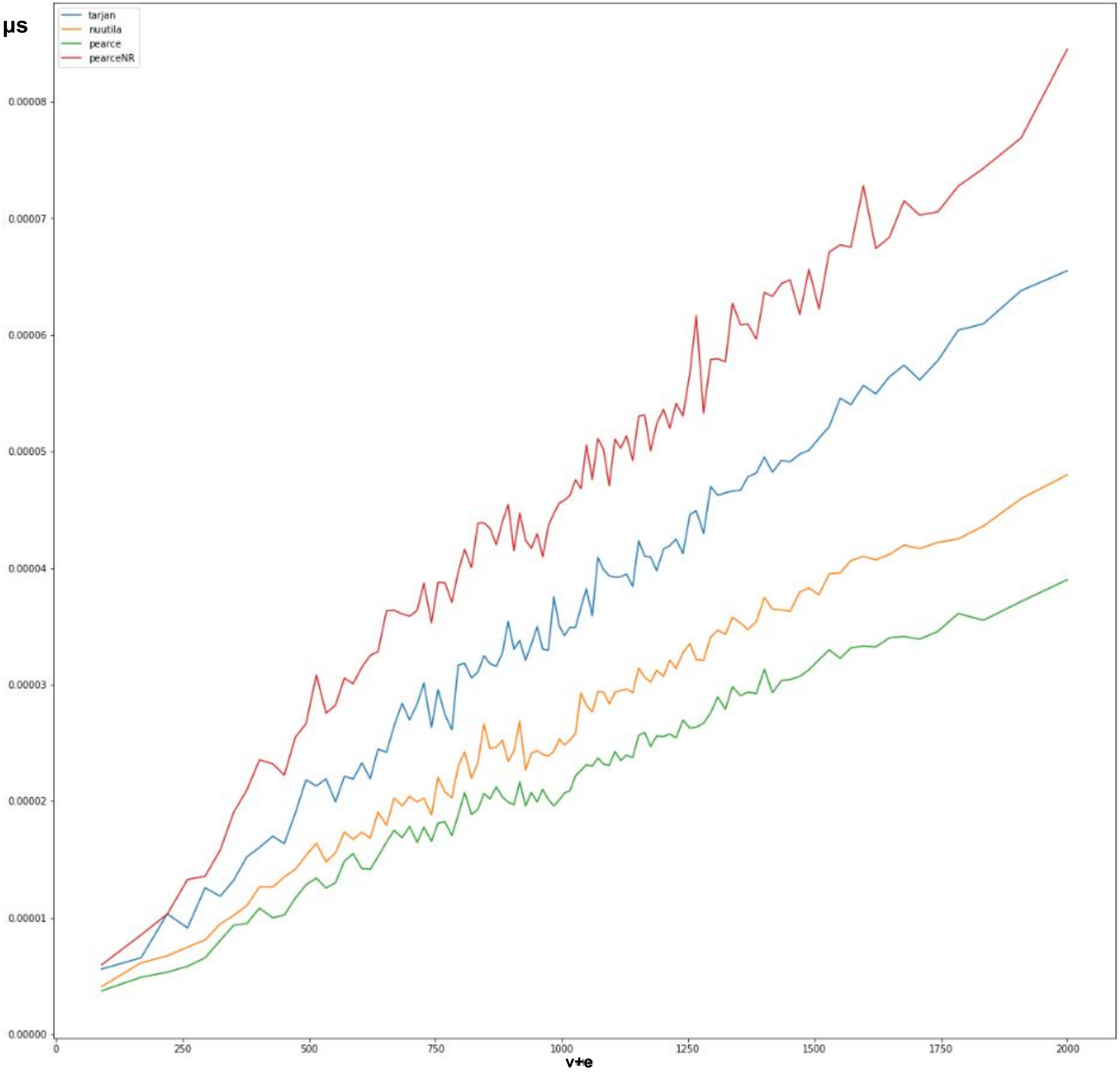


Pearce



Pearce non recursive





PearceNR

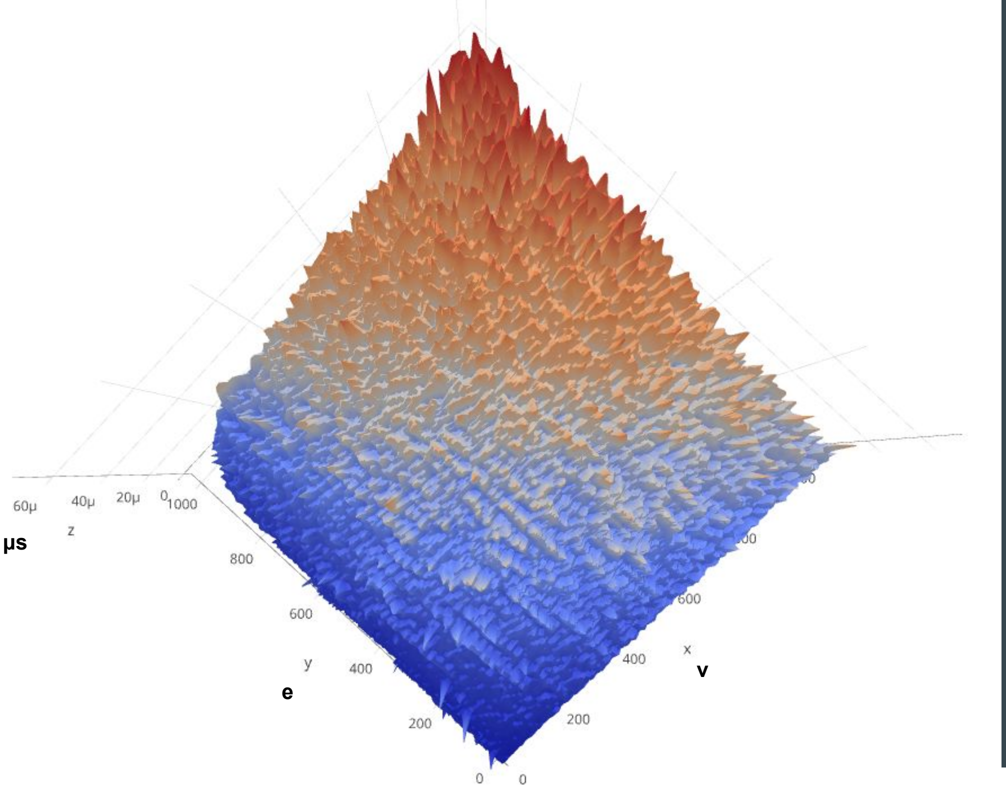
Tarjan

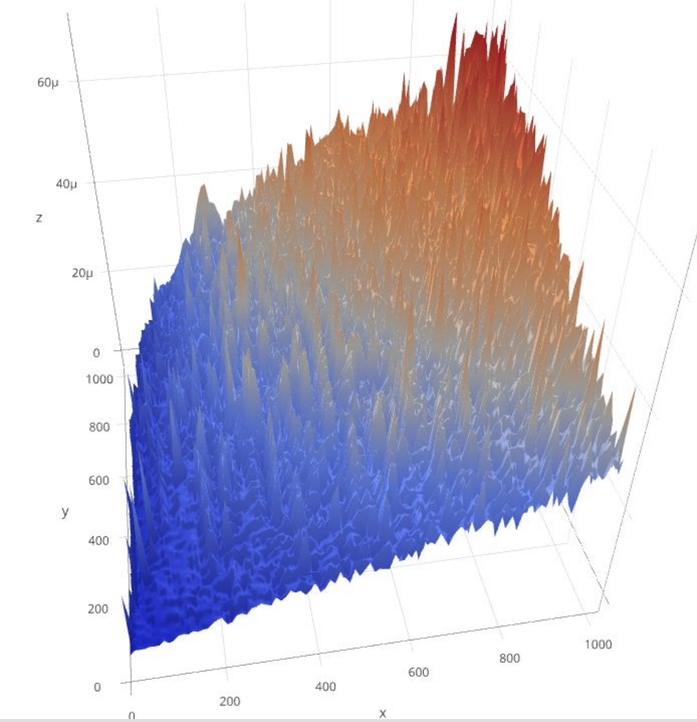
Nuutila

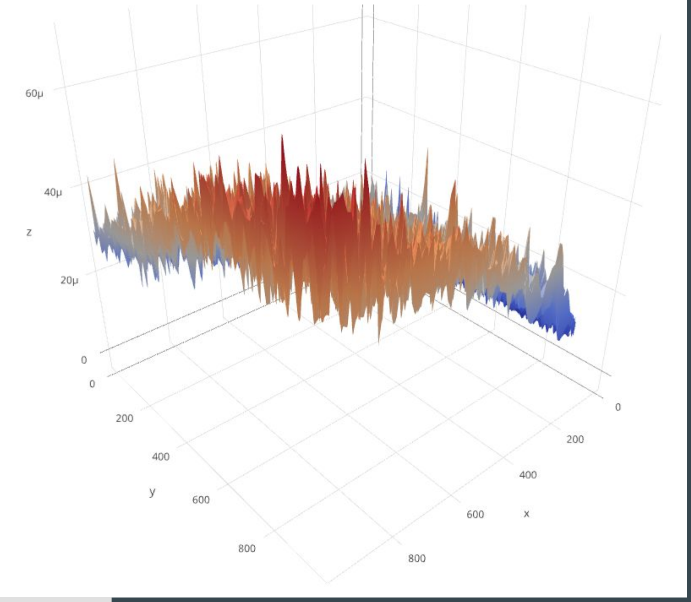
Pearce

(V,E,t) space

- Tarjan
- Nuutila
- Pearce
- Non recursive Pearce







Memory Analysis

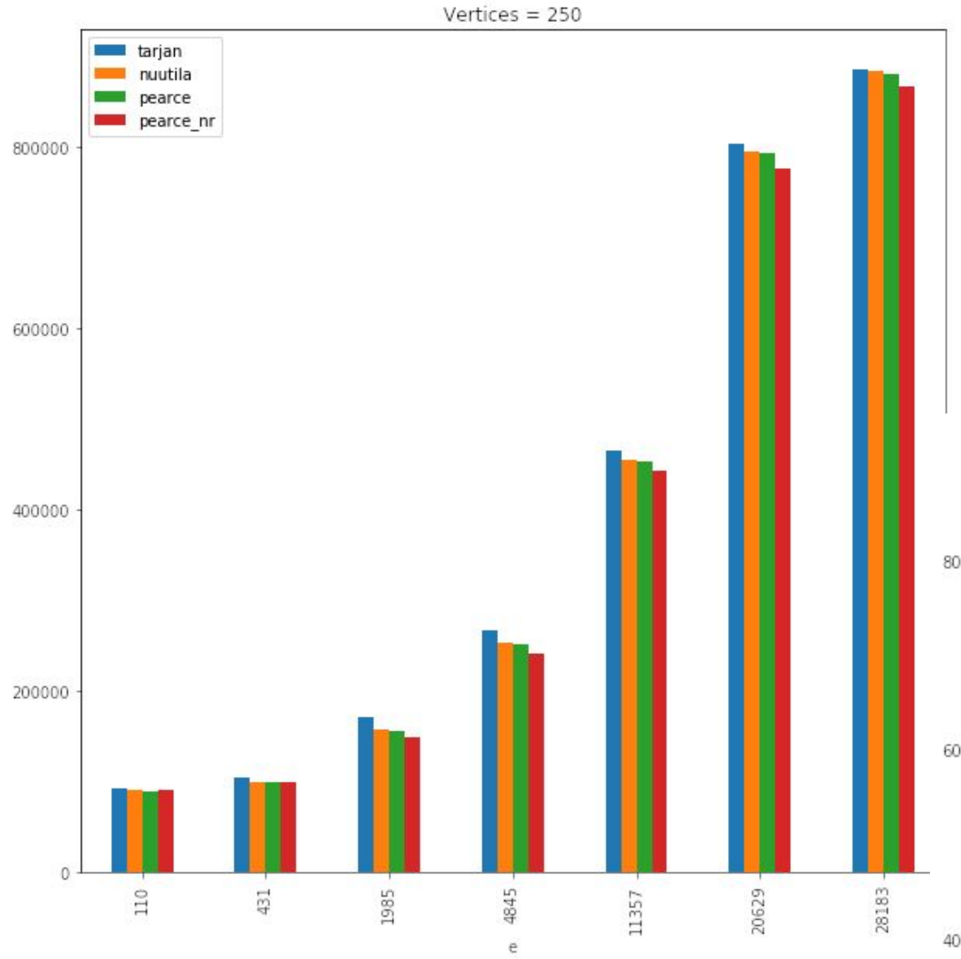
Our aim was to confirm, generally speaking, what's reported by the papers, so that Tarjan's algorithm is less efficient than Nuutila's and, in turn, the latter is less efficient than Pearce's.

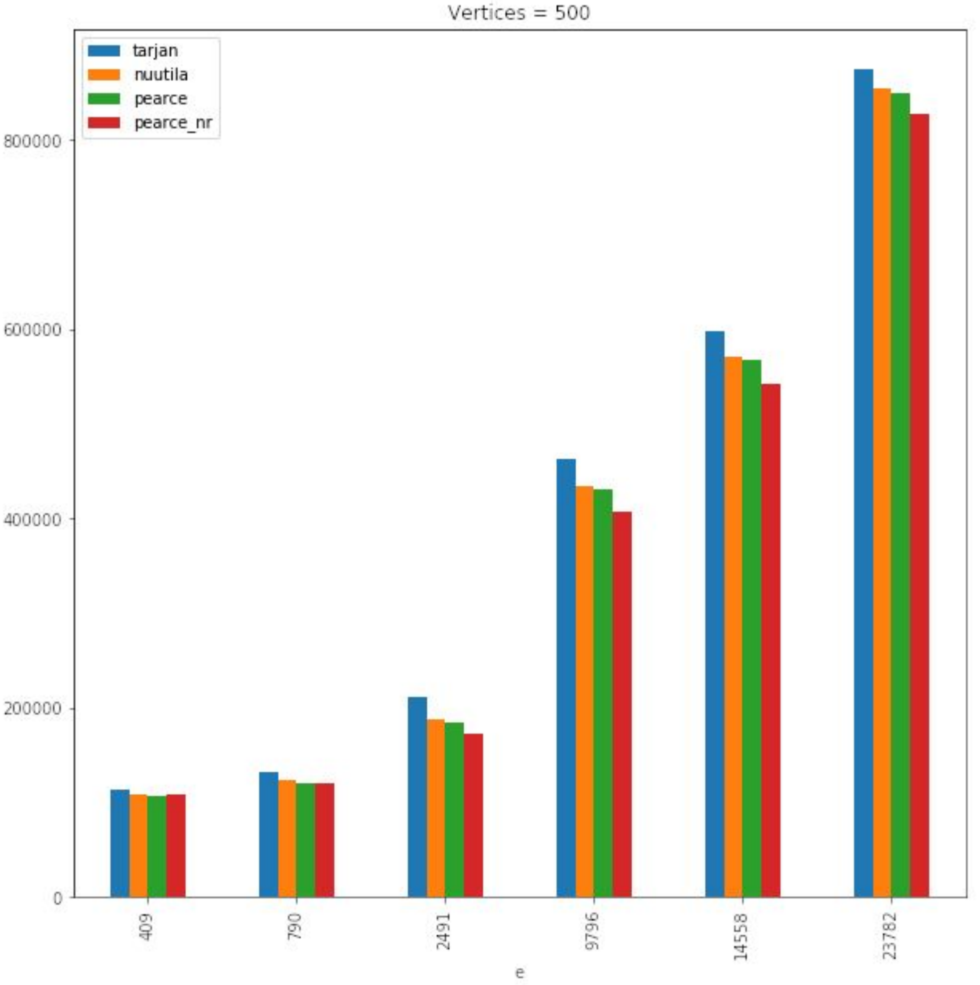
Despite the similar characteristics, Nuutila's is less efficient because it uses more data structures to achieve the same results of Pearce's.

At last, we tested how the iterative approach of Pearce impacts on the memory allocation w.r.t. to the other algorithms, expecting that, when a huge amount of recursive calls occur, it saves a lot of memory in terms of stack occupancy, nullifying the overhead introduced by the usage of more data structures.

What we did:

- We used Valgrind's Massif tool to measure heap and stack allocations' peak for all the four algorithms, feeding the same test graphs. In order to have a general result, we have tested with graphs that differ by an increasing (step fashion) number of vertex and different number of edges, for the same number of vertex.

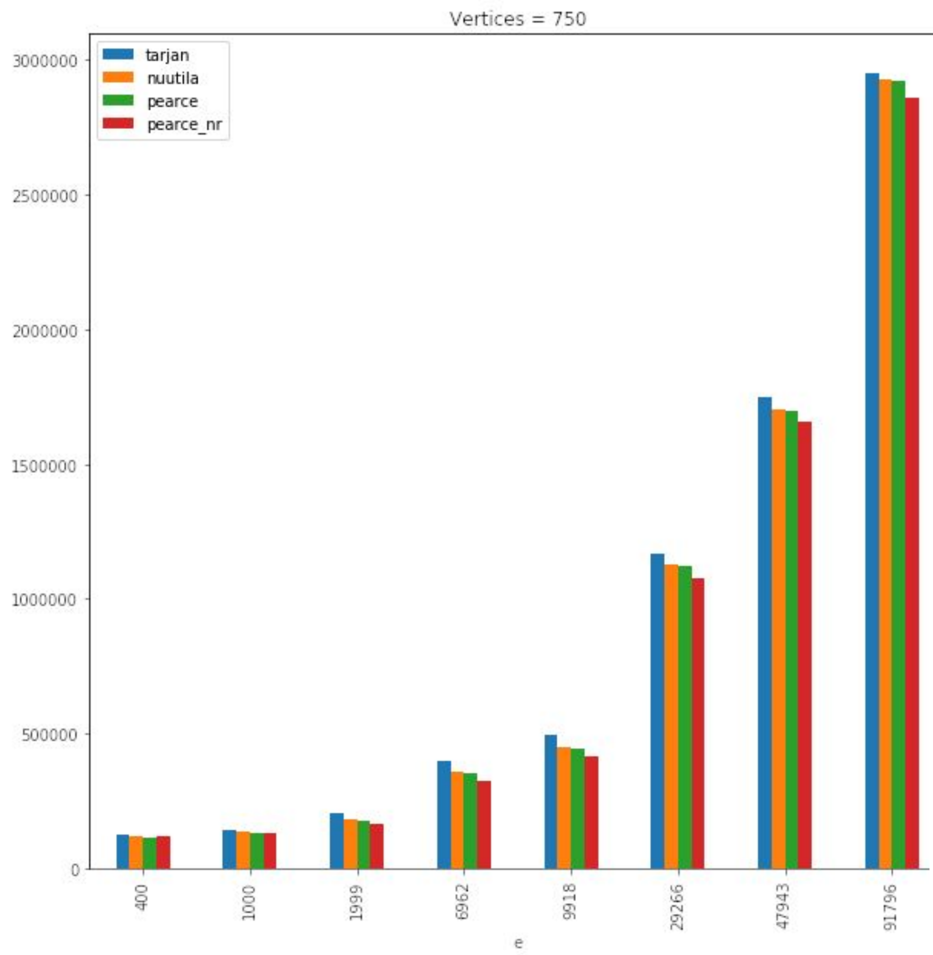




Vertex Number:

250 (Above)

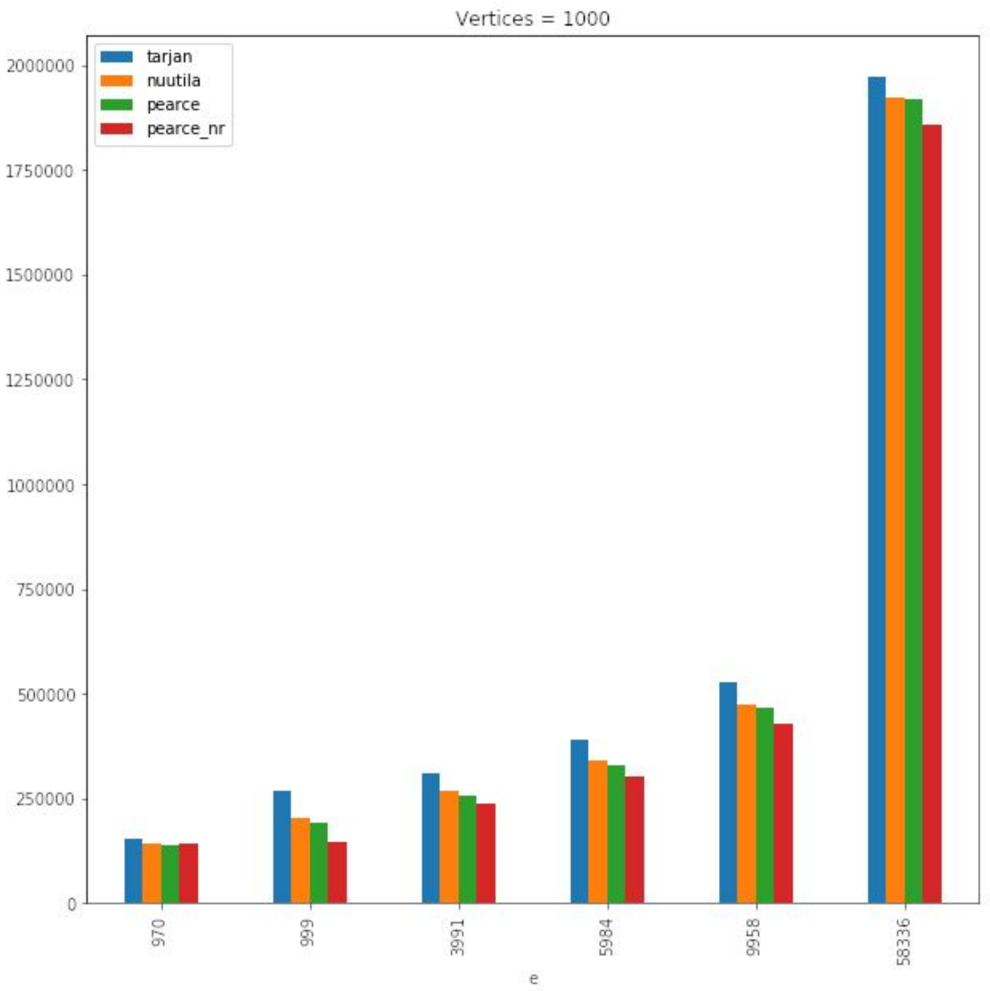
500 (Side)

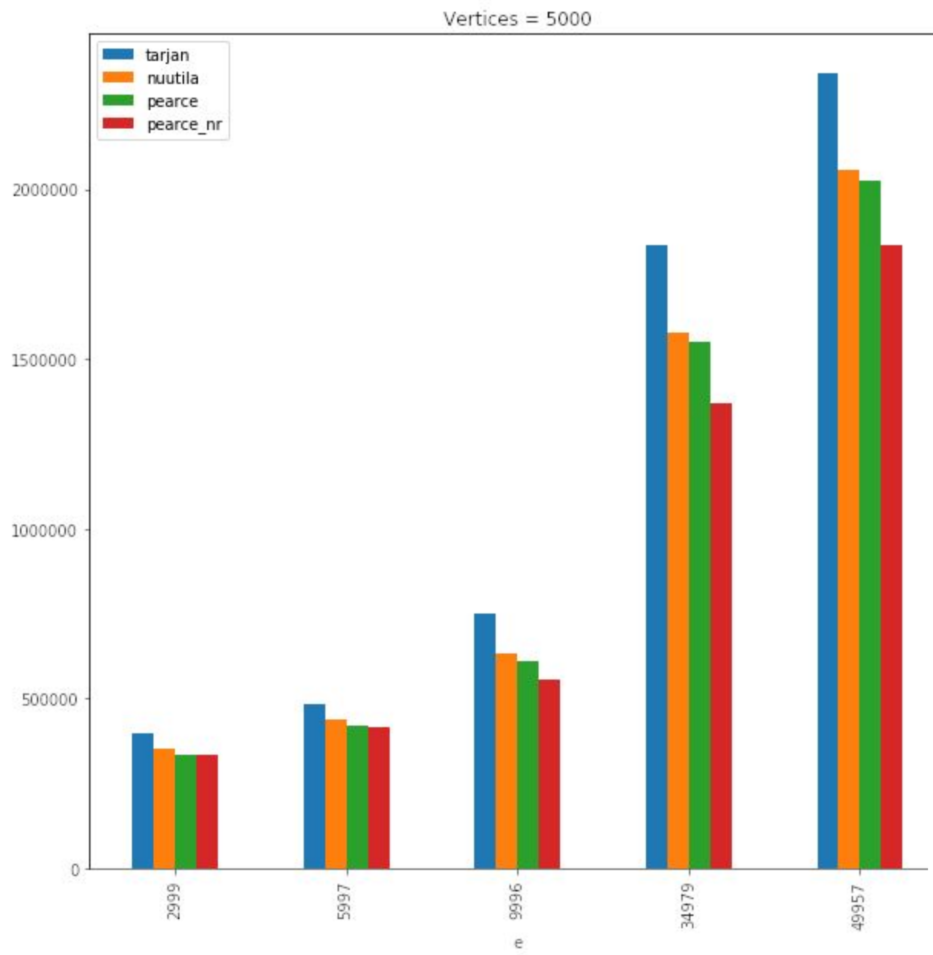


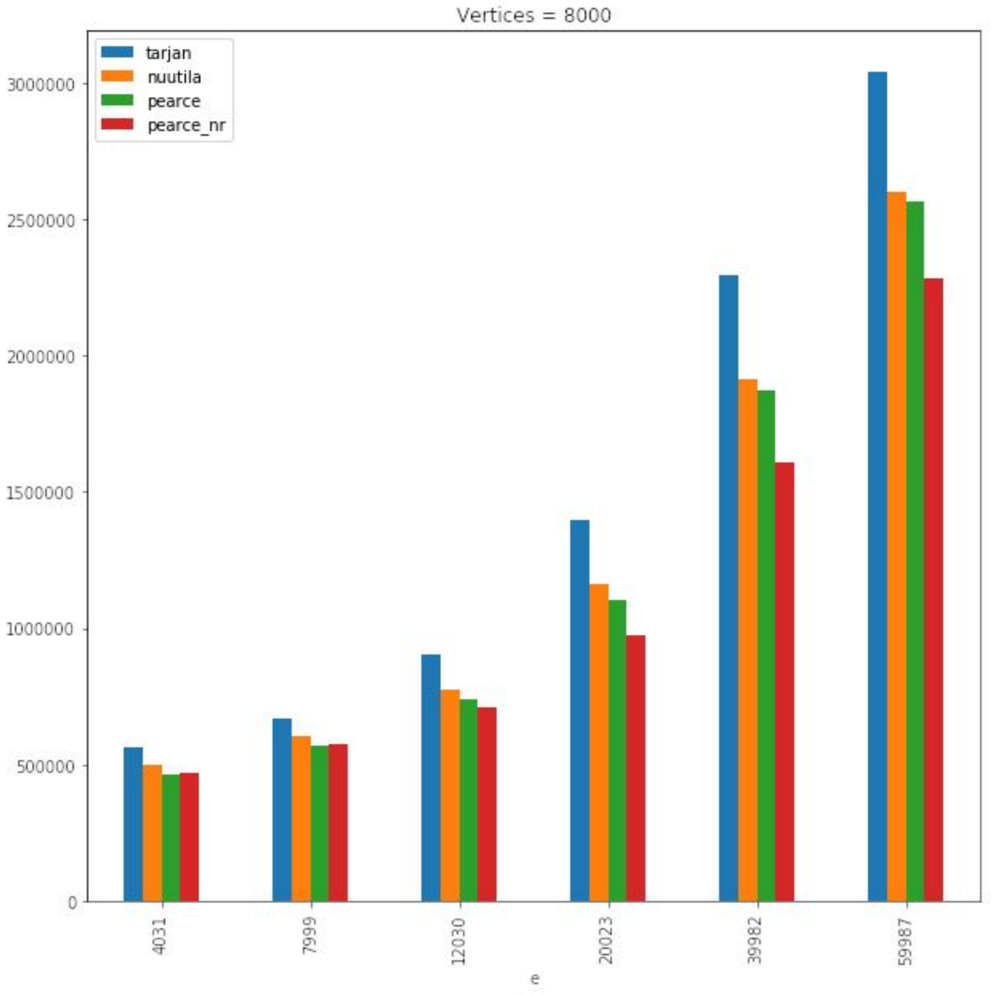
Vertex Number:

750 (Above)

1000 (Side)



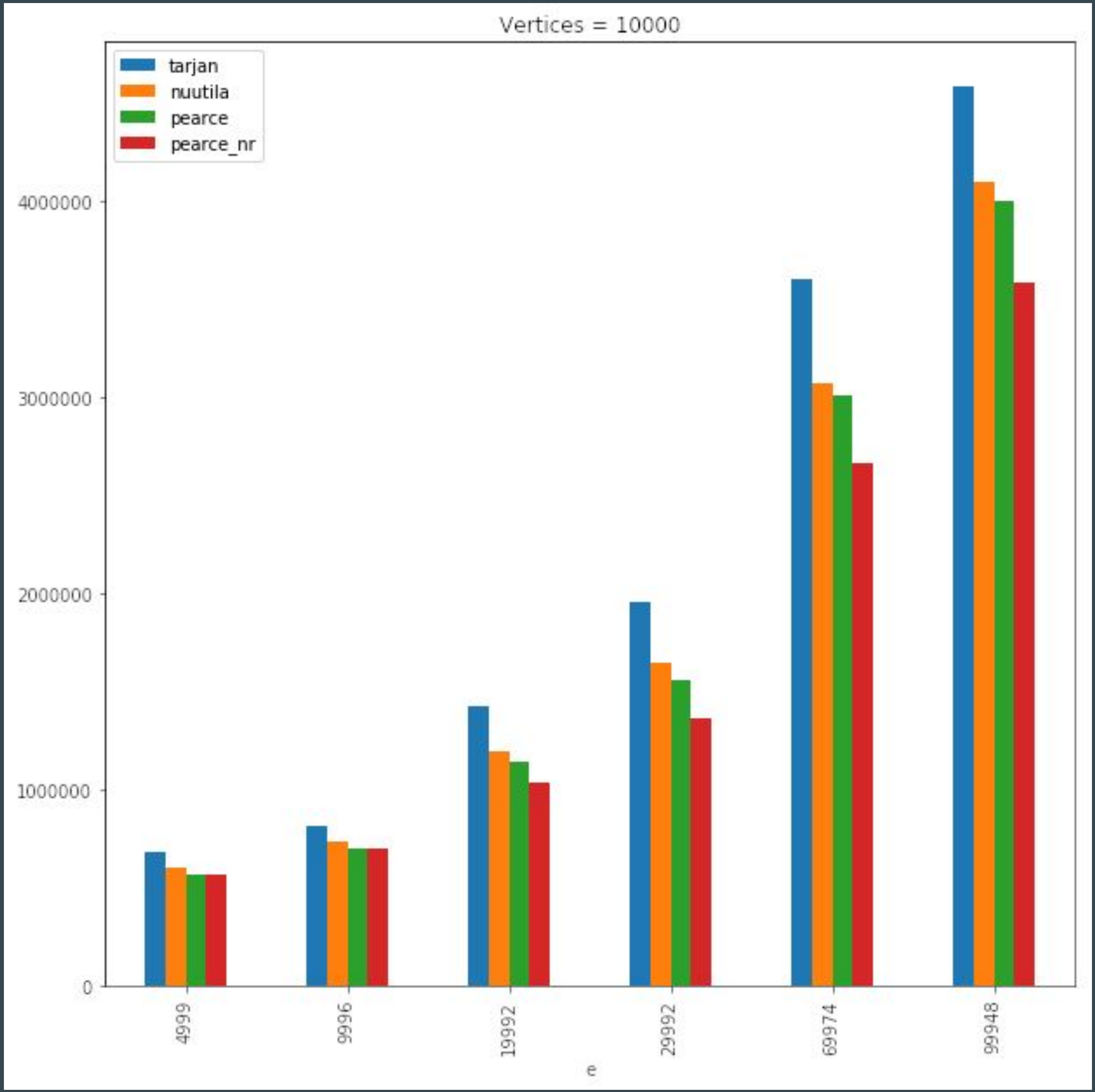




Vertex Number:

5000 (Above)

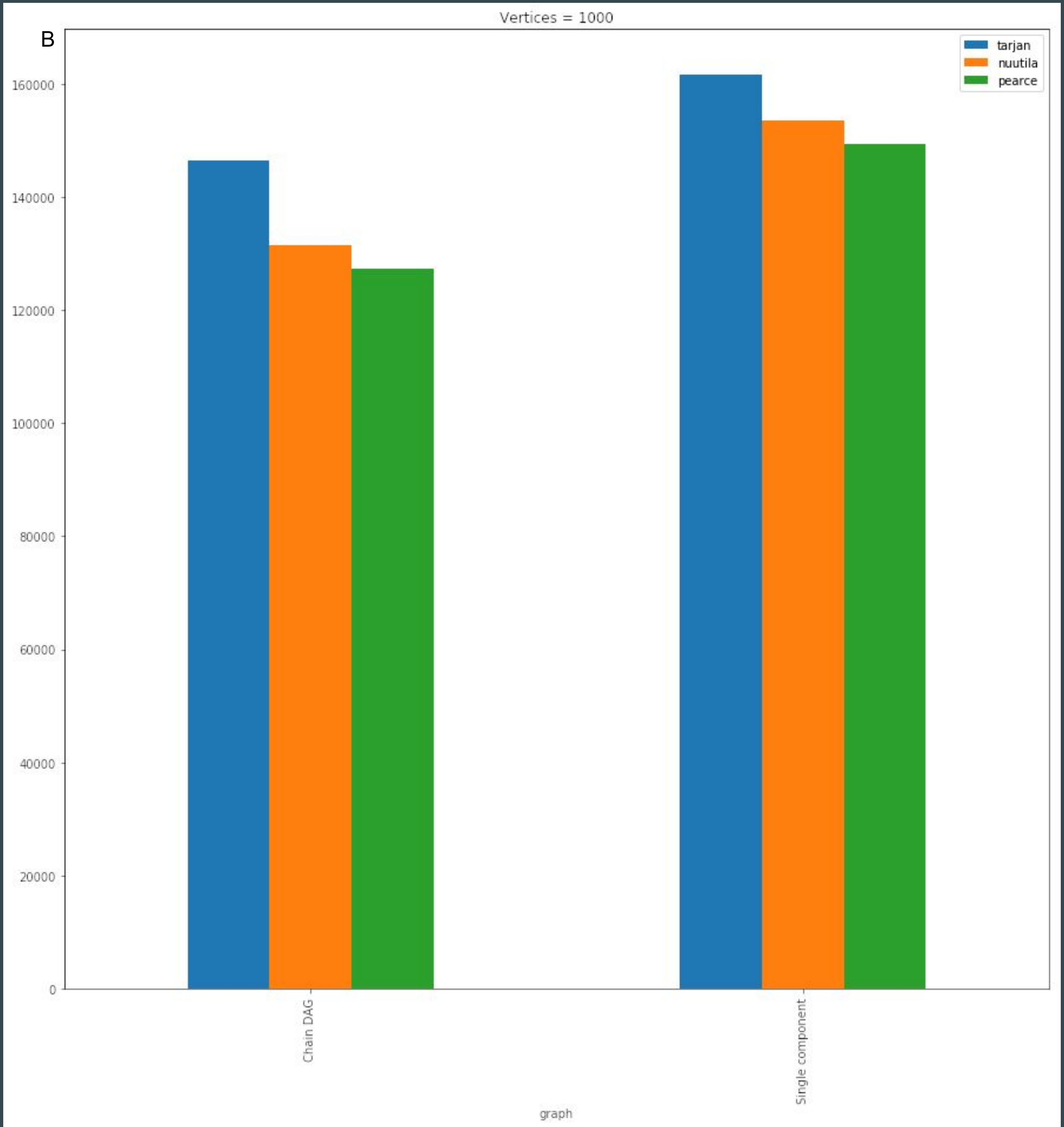
8000 (Side)

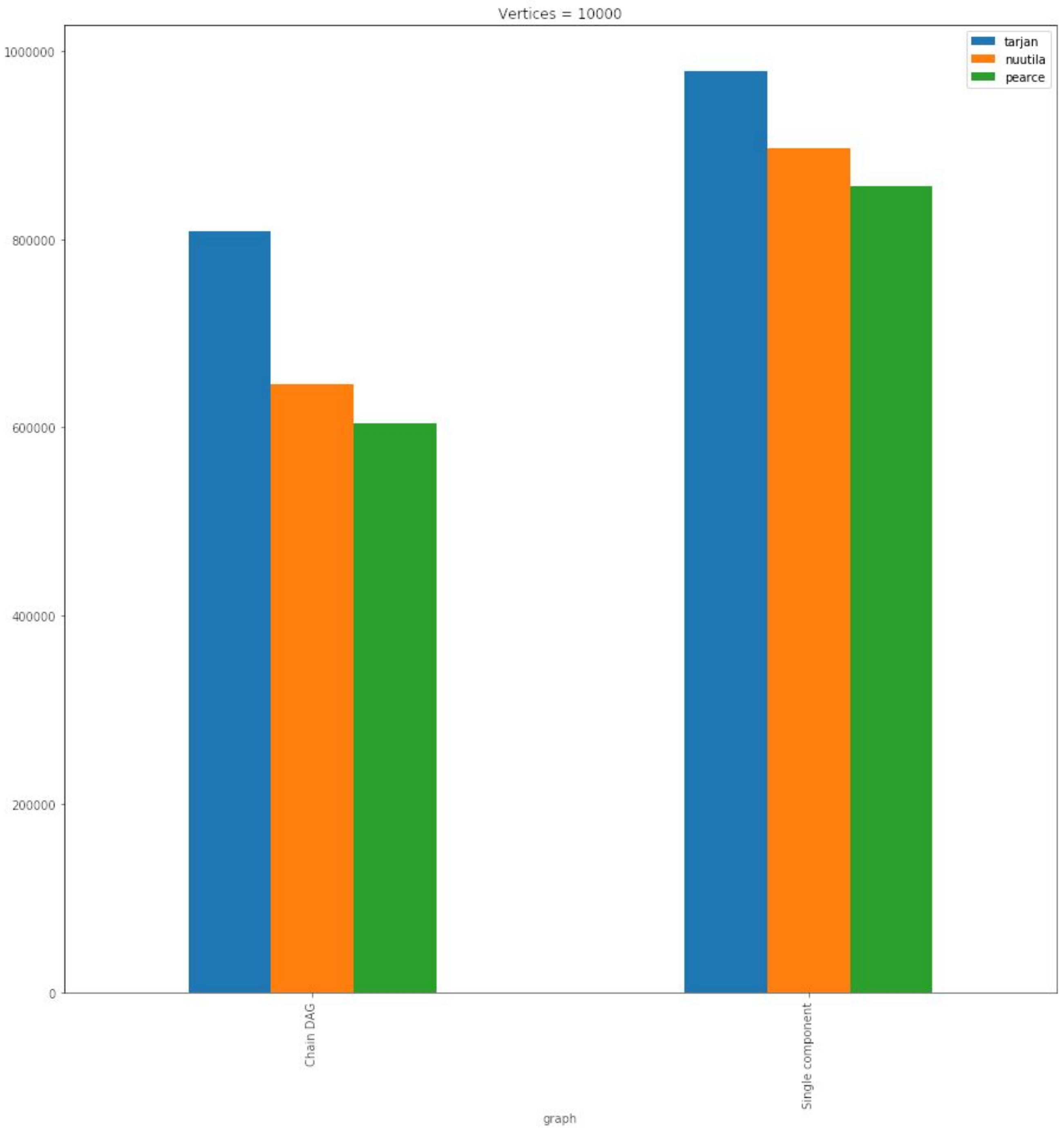

Vertices = 10000
tarjan
nuutila
pearce
pearce_nr
4000000
3000000
2000000
1000000
0
4999
9996
19992
29992
69974
99948
e

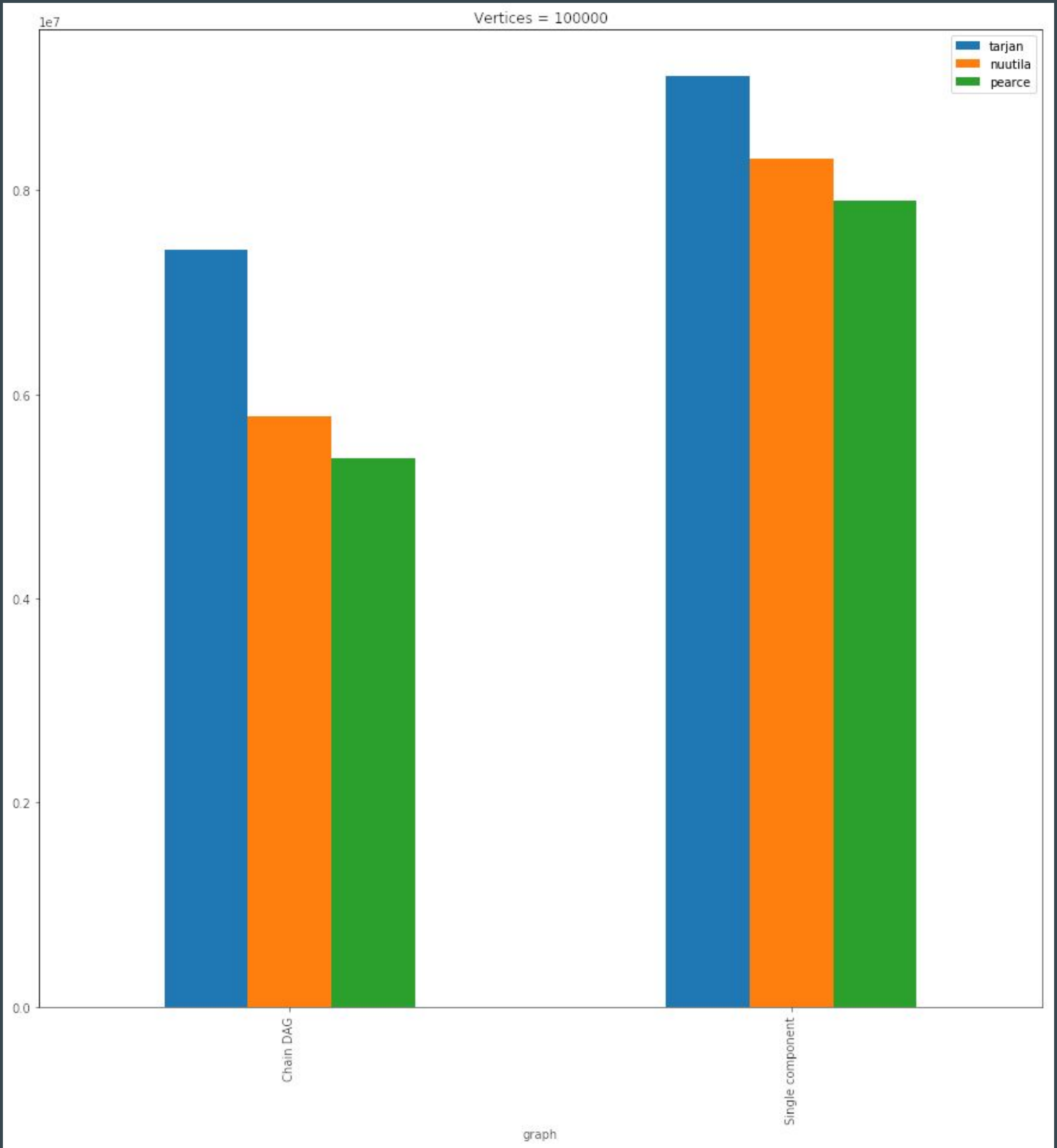
Further Memory Analysis

In order to assess if our implementations showed the same benefits described in Nuutila’s and Pearce’s paper, s.t. no “trivial” SCC’s vertex is pushed on the stack, we evaluated the heap peaks to measure the gap between Tarjan and the others in two special test cases, at the two ends of the spectrum:

- Chain DAG : where all SCC are trivial
- Single Component Graph : all vertex in the same SCC


B
Vertices = 1000
tarjan
nuutila
pearce
160000
140000
120000
100000
80000
60000
40000
20000
0
Chain DAG
Single component
graph


Vertices = 10000
1000000
800000
600000
400000
200000
0
tarjan
nuutila
pearce
Chain DAG
Single component
graph

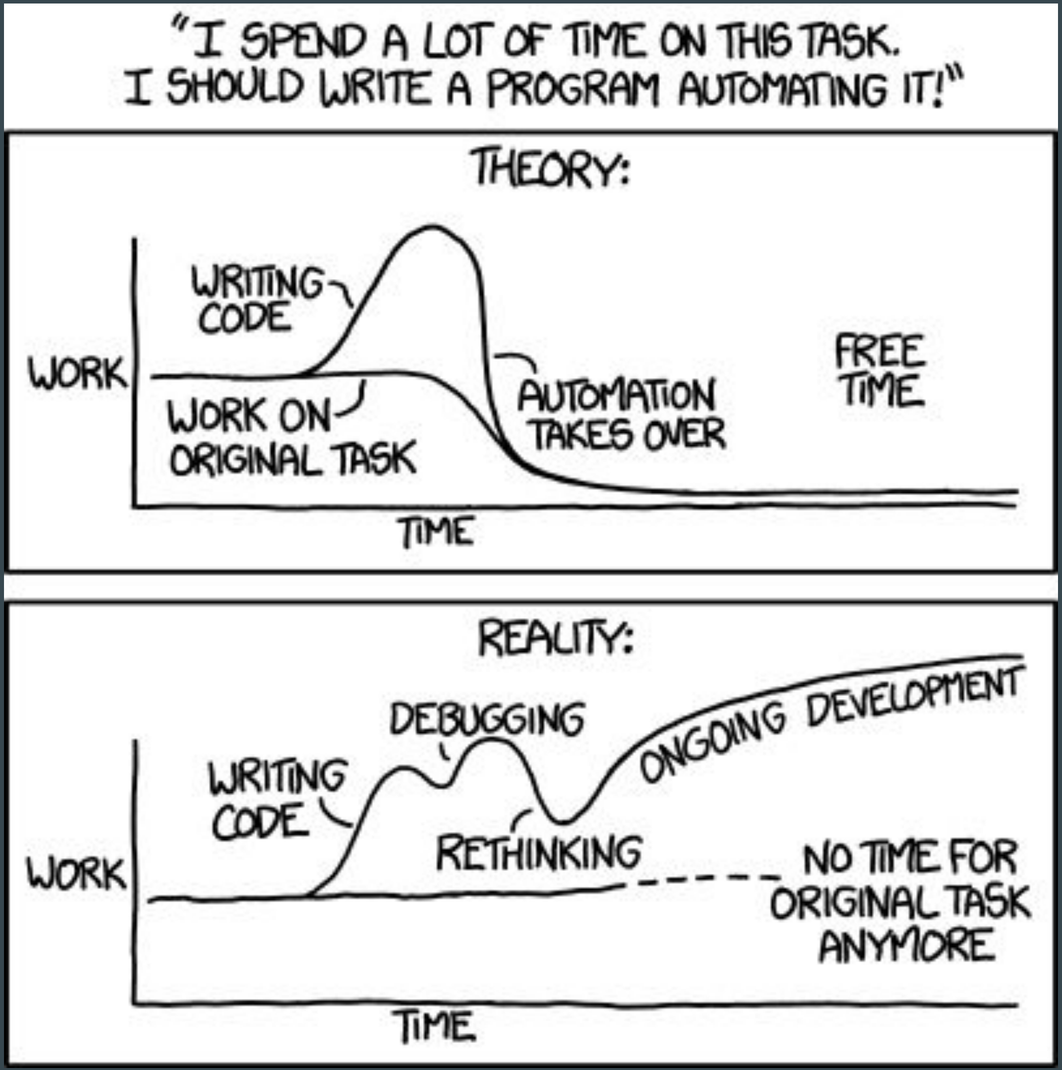

Vertices = 100000
1e7
0.8
0.6
0.4
0.2
0.0
tarjan
nuutila
pearce
Chain DAG
Single component
graph

Conclusions

We have :

- Implemented the algorithms
- Verified that our implementation respects time and space complexity
- Shown empirically the benefits between the 3 iteration of the algorithm

THANK YOU FOR YOUR ATTENTION!


"I SPEND A LOT OF TIME ON THIS TASK.
I SHOULD WRITE A PROGRAM AUTOMATING IT!"
THEORY:
WRITING CODE
WORK
WORK ON ORIGINAL TASK
AUTOMATION TAKES OVER
FREE TIME
TIME
REALITY:
DEBUGGING
WRITING CODE
ONGOING DEVELOPMENT
RETHINKING
WORK
NO TIME FOR ORIGINAL TASK ANYMORE
TIME