POLITECNICO DI MILANO

Strongly connected components

Konstantinos Karouzos and Asimina Mertzani

Index

- Introduction
- Theoritical presentation of the algorithms of Tarjan, Nuutila and Pearce
- Implementation of the algorithms of Tarjan, Nuutila and Pearce
- Experimental Results
- Conclusions

Introduction

Main topic: Finding of Strongly Connected Components (SCC) of a directed graph

To reach this goal there will be presented the implementation and comparison of three different algorithms for finding the SCC of a directed graph:

- Tarjan’s Algorithms for SCC
- Nuutila’s Algorithms
- Pearce’s Algorithm

Strongly Connected Compontent Definition

A directed graph is said to be strongly connected if every vertex is reachable from every other vertex.

The strongly connected components of an arbitrary directed graph form a partition into subgraphs that are themselves strongly connected.

Main features of Tarjan’s Algorithm

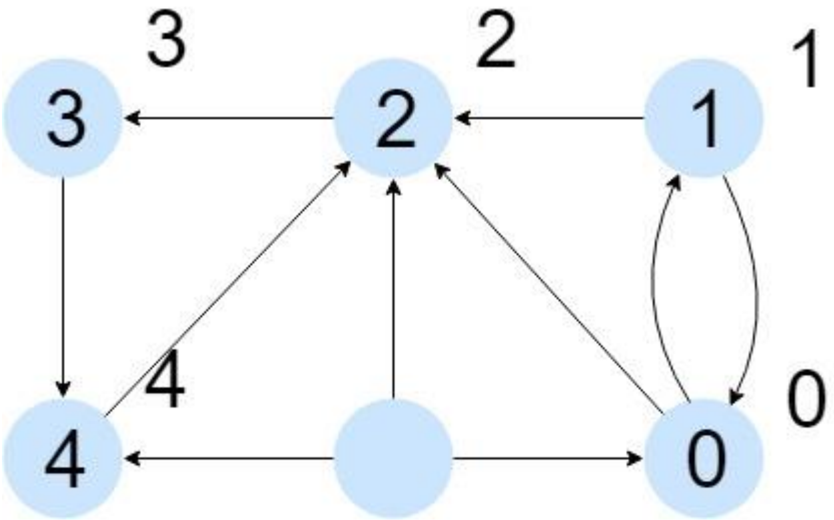
- Based upon Depth First Search Algorithm (DFS)
- Stores information concerning root of each element
- Usage of stack (control of valid nodes)

Tarjan's Algorithm

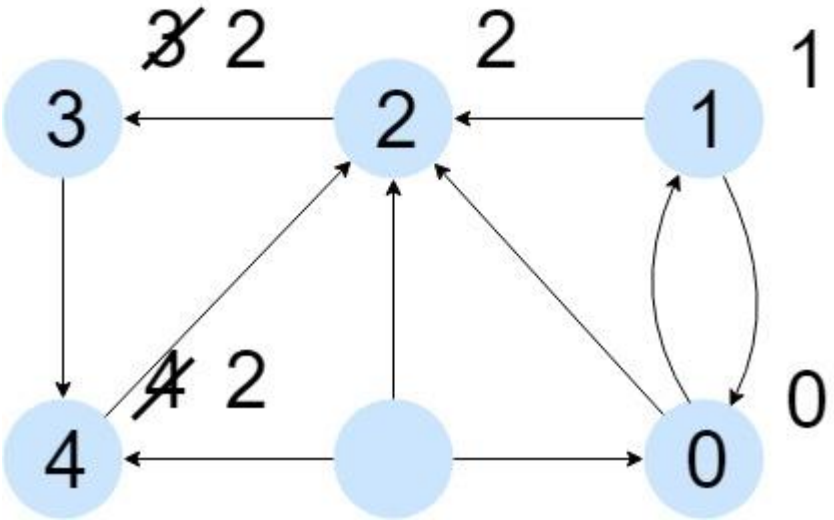
```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Figure 1: Tarjan's algorithm detects the strongly connected components of graph $G = (V, E)$.

Tarjan's Algorithm

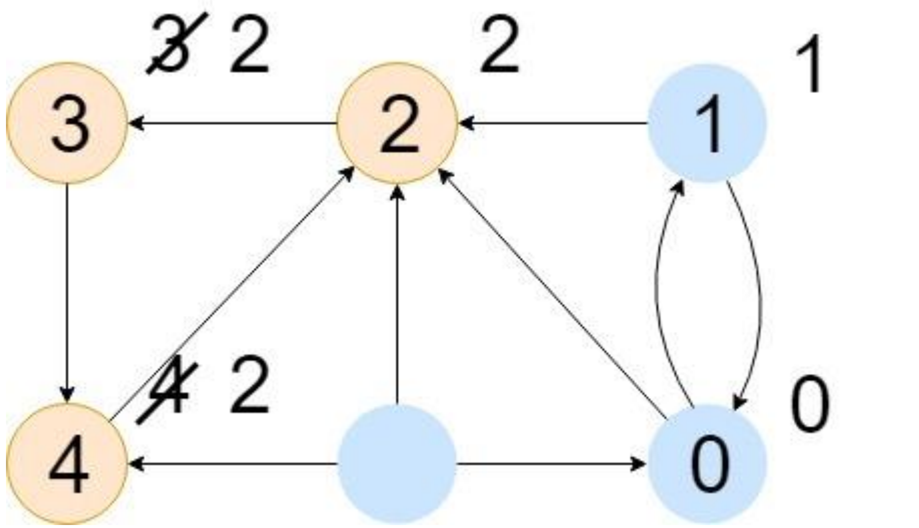


Stack:
0
1
2
3
4



Stack:
0
1

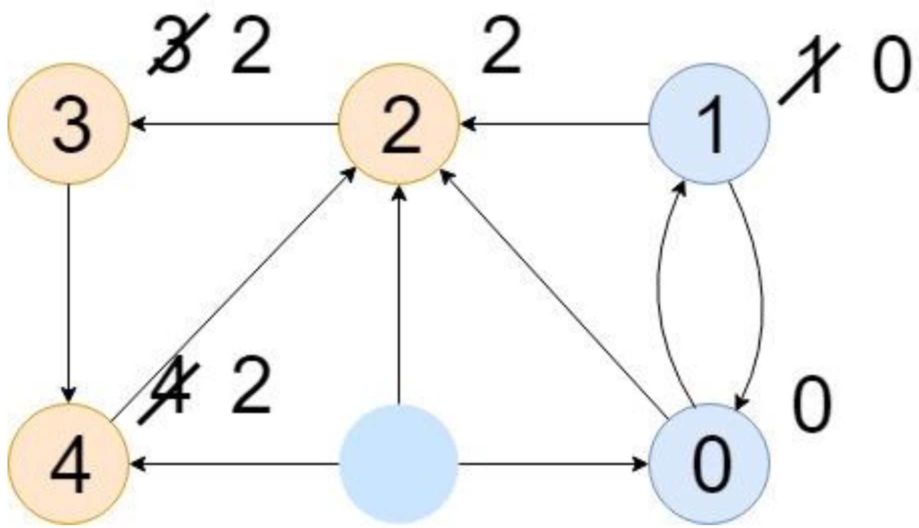
Tarjan's Algorithm



Stack:

0

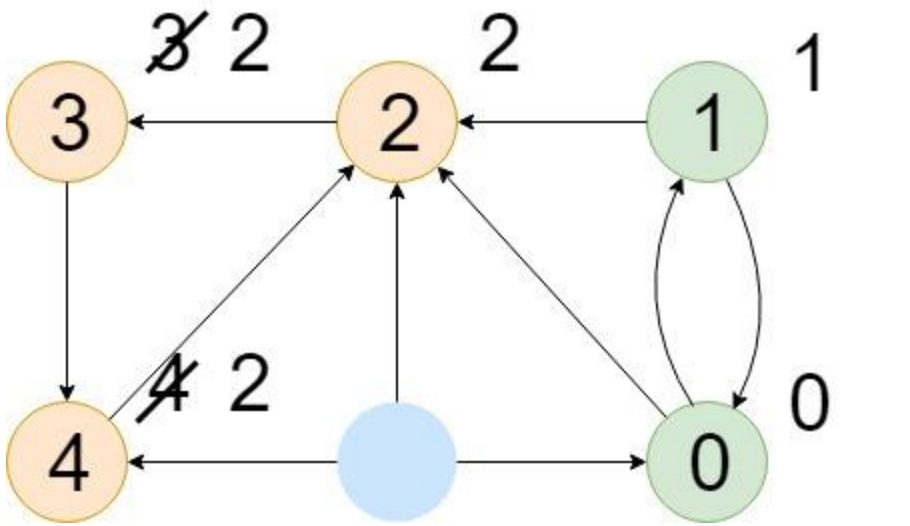
1



Stack:

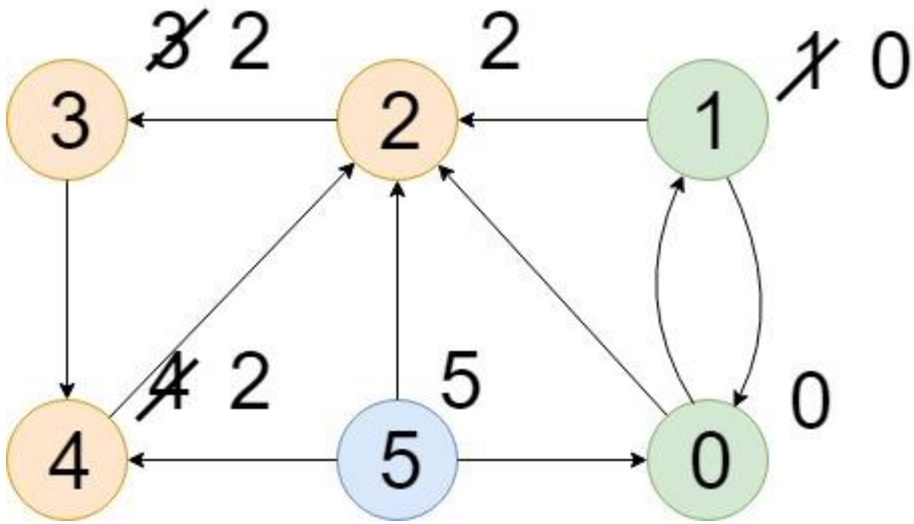
-

Tarjan’s Algorithm



Stack:

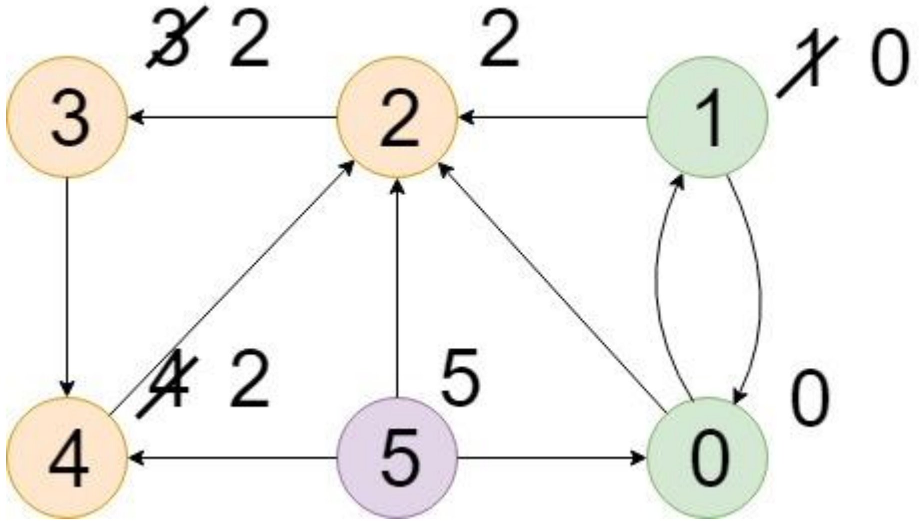
-



Stack:

5

Tarjan's Algorithm



Stack:

-

Main Features of Nuutila’s Algorithm

- 2 improved versions of Tarjan’s Algorithm
- Main idea: eliminate elements stored on the stack
- Improved runtime

1st algorithm: Stores only non root elements on the stack
2nd algorithm: Stores only final candidate roots on the stack

Comparison: Nuutila’s 1st – Tarjan’s

Tarjan’s Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Nuutila’s 1st Algorithm

```
procedure VISIT1(v);
begin
    root[v] := v; InComponent[v] := False;
    for each node w such that (v, w) ∈ 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 := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT1(v)
end.
```

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

1st algorithm: Stores only non root elements on the stack

Comparison: Nuutila's 1st – Tarjan's

Tarjan's Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Nuutila's 1st Algorithm

```
procedure VISIT1(v);
begin
    root[v] := v; InComponent[v] := False;
    for each node w such that (v, w) ∈ 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 := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT1(v)
end.
```

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

- Nodes stored on the stack when it is ensured that they are nonroote nodes

Comparison: Nuutila's 1st – Tarjan's

Tarjan's Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Nuutila's 1st Algorithm

```
procedure VISIT1(v);
begin
    root[v] := v; InComponent[v] := False;
    for each node w such that (v, w) ∈ 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 := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT1(v)
end.
```

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

- Removes from the stack nodes that have been visited after the root (since root is not on the stack)

Comparison: Nuutila’s 2nd – Tarjan’s

2nd algorithm: Stores only final candidate roots on the stack

Definition: Node x is **Final Candidate root** if x is the root of a node y, for some y node that have already processed all its neighbours.

Comparison: Nuutila’s 2nd – Tarjan’s

Tarjan’s Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Nuutila’s 2nd Algorithm

```
procedure VISIT2(v);
begin
    root[v] := v; InComponent[v] := False;
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT2(w);
        if not InComponent[root[w]] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        if TOP(stack) ≥ v then
            repeat
                w := POP(stack);
                InComponent[w] := True;
            until TOP(stack) < v;
        else InComponent[v] := True;
    else if root[v] is not on stack then PUSH(root[v], stack);
end;
begin/* Main program */
    Initialize stack to contain a value < any node in V;
    for each node v ∈ V do
        if v is not already visited then VISIT2(v)
end.
```

Figure 3: Algorithm 2 stores only final candidate root nodes on the stack.

- Nodes stored on the stack when it is ensured that they are final candidate root nodes

Comparison: Nuutila's 2nd – Tarjan's

Tarjan's Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Nuutila's 2nd Algorithm

```
procedure VISIT2(v);
begin
    root[v] := v; InComponent[v] := False;
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT2(w);
        if not InComponent[root[w]] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        if TOP(stack) ≥ v then
            repeat
                w := POP(stack);
                InComponent[w] := True;
            until TOP(stack) < v;
        else InComponent[v] := True;
    else if root[v] is not on stack then PUSH(root[v], stack);
end;
begin/* Main program */
    Initialize stack to contain a value < any node in V;
    for each node v ∈ V do
        if v is not already visited then VISIT2(v)
end.
```

Figure 3: Algorithm 2 stores only final candidate root nodes on the stack.

- Roots of trivial components (components consisting of a single node) are not stored on the stack
- They are just assigned to a component

Comparison: Nuutila’s 2nd – Tarjan’s

Tarjan’s Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Nuutila’s 2nd Algorithm

```
procedure VISIT2(v);
begin
    root[v] := v; InComponent[v] := False;
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT2(w);
        if not InComponent[root[w]] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        if TOP(stack) ≥ v then
            repeat
                w := POP(stack);
                InComponent[w] := True;
            until TOP(stack) < v;
        else InComponent[v] := True;
    else if root[v] is not on stack then PUSH(root[v], stack);
end;
begin/* Main program */
    Initialize stack to contain a value < any node in V;
    for each node v ∈ V do
        if v is not already visited then VISIT2(v)
end.
```

Figure 3: Algorithm 2 stores only final candidate root nodes on the stack.

- Removes from the stack nodes that have been visited after the root (since root is not on the stack)

Expectated results in runtime

First traversal: cannot be eliminated (cost: O(n+m))

Second traversal:

1. Tarjan’s Algorithm: All nodes on the stack (cost: O(n)).
2. 1st Nuutila’s Algorithm: Only non root elements on the stack (cost: O(n-s)).
3. 2nd Nuutila’s Algorithm: Only final candidate roots of non-trivial component on the stack (cost: O(x where 0<=x<=n-s).

n: number of nodes
m: number of edges
s: number of SCC

Expectated results in runtime

Dense graph (M≈NxN): not significant improvement

Sparse graph (M≈N) :

- both Nuutila’s Algorithms are expected to achieve better results
- especially the 2nd is expected to have even better results than the first.

Main Features of Pearce’s Algorithm

- Based upon DFS
- Decreased the bits of storage of Tarjan’s Algorithm
- Better results in memory usage

Comparison: Pearce's – Tarjan's

Tarjan's Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Pearce's Algorithm

```
for all v ∈ V do rindex[v] = 0
S = ∅ ; index = 1 ; c = |V| − 1
for all v ∈ 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 → w ∈ 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 ≠ ∅ ∧ rindex[v] ≤ 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)
```

Pearces Algorithm: Less arrays used

Comparison: Pearce’s – Tarjan’s

Tarjan’s Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Pearce’s Algorithm

```
for all v ∈ V do rindex[v] = 0
S = ∅ ; index = 1 ; c = |V| − 1
for all v ∈ 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 → w ∈ 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 ≠ ∅ ∧ rindex[v] ≤ 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)
```

- ridex[.] instead of id[.] + root[.]

Comparison: Pearce’s – Tarjan’s

Tarjan’s Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Pearce’s Algorithm

```
for all v ∈ V do rindex[v] = 0
S = ∅ ; index = 1 ; c = |V| − 1
for all v ∈ 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 → w ∈ 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 ≠ ∅ ∧ rindex[v] ≤ 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)
```

- ridex[.]=0 (initially) instead of IsVisited[.]

Comparison: Pearce's – Tarjan's

Tarjan's Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Pearce's Algorithm

```
for all v ∈ V do rindex[v] = 0
S = ∅ ; index = 1 ; c = |V| − 1
for all v ∈ 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 → w ∈ 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 ≠ ∅ ∧ rindex[v] ≤ 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)
```

- ridex[.] instead of InComponent[.]

Comparison: Pearce's – Tarjan's

Tarjan's Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Pearce's Algorithm

```
for all v ∈ V do rindex[v] = 0
S = ∅ ; index = 1 ; c = |V| − 1
for all v ∈ 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 → w ∈ 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 ≠ ∅ ∧ rindex[v] ≤ 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)
```

- Initializes: index= 1 (instead of index=0)
- Initializes: c= |n|-1 (instead of c=0)

Comparison: Pearce’s – Tarjan’s

Tarjan’s Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Pearce’s Algorithm

```
for all v ∈ V do rindex[v] = 0
S = ∅ ; index = 1 ; c = |V| − 1
for all v ∈ 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 → w ∈ 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 ≠ ∅ ∧ rindex[v] ≤ 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)
```

- Index is decreased when a node is assigned to a component
- C is decreased when a new component is found

Comparison: Pearce’s – Tarjan’s

Tarjan’s Algorithm

```
procedure VISIT(v);
begin
    root[v] := v; InComponent[v] := False;
    PUSH(v, stack);
    for each node w such that (v, w) ∈ E do begin
        if w is not already visited then VISIT(w);
        if not InComponent[w] then root[v] := MIN(root[v], root[w])
    end;
    if root[v] = v then
        repeat
            w := POP(stack);
            InComponent[w] := True;
        until w = v
end;
begin/* Main program */
    stack := ∅;
    for each node v ∈ V do
        if v is not already visited then VISIT(v)
end.
```

Pearce’s Algorithm

```
for all v ∈ V do rindex[v] = 0
S = ∅ ; index = 1 ; c = |V| − 1
for all v ∈ 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 → w ∈ 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 ≠ ∅ ∧ rindex[v] ≤ 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)
```

- Removes nodes from stack till finding a node with greater rindex

Expectated results in memory usage

1. Tarjan's Algorithm: $n(2+5w)$ bits of storage
2. Nuutila's Algorithms: $n(1+4w)$ bits of storage
3. Pearce's Algorithm: $n(1+3w)$ bits of storage

n: number of nodes
w: machine's word size

Random Graph Generation

- createG.cpp

- ./createG N M graph.txt

- Create a random graph for a given number of Vertices and Edges.

- Use std::rand and current time as seed for the random genarator.

- Implementing the Graph as

  typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::bidirectionalS> Graph;

- Store the graph as a .txt file in graphiz form.

Random Graph Generation

```cpp
void RandDirectedGraph(Graph &g, int N, int M);

int main(int argc, char * argv [])
{
  using namespace boost;

  std::string a1=argv[1];
  std::string a2=argv[2];
  int N =std::atoi(a1.c_str());
  int M =std::atoi(a2.c_str());

  Graph G(N);

  std::ofstream myfile (argv[3]);

  RandDirectedGraph(G, N, M);

  if (myfile.is_open())
  {
    write_graphviz(myfile, G);
    myfile.close();
  }
  return 0;
}


void RandDirectedGraph(Graph &g, int N, int M)
{
  // directed_graph is a subclass of adjacency_lis

  for (int i=0; i<M; i++){
    int k1= rand() % N;
    int k2= rand() % N;
    boost::add_edge(k1, k2, g);
  }
 }
```

Tarjan’s Algorithm Implementation

- An implementation of Tarjan’s Algorithm already exists in BGL

1. Defined in boost/graph/strong_components.hpp

2. The implementation is mainly composed by the definition of the tarjan_scc_visitor class, that extends dfs_visitor.

3. The class is equipped with discover_vertex and finish_vertex methods, that corresponds to the two phases of the DFS.

Tarjan's Algorithm Implementation

```
template <typename Graph>
void discover_vertex(typename graph_traits<Graph>::vertex_descriptor v,
                     const Graph&) {
  put(root, v, v);
  put(comp, v, (std::numeric_limits<comp_type>::max)());
  put(discover_time, v, dfs_time++);
  s.push(v);
}
template <typename Graph>
void finish_vertex(typename graph_traits<Graph>::vertex_descriptor v,
                   const Graph& g) {
  typename graph_traits<Graph>::vertex_descriptor w;
  typename graph_traits<Graph>::out_edge_iterator ei, ei_end;
  for (boost::tie(ei, ei_end) = out_edges(v, g); ei != ei_end; ++ei) {
    w = target(*ei, g);
    if (get(comp, w) == (std::numeric_limits<comp_type>::max)())
      put(root, v, this->min_discover_time(get(root,v), get(root,w)));
  }
  if (get(root, v) == v) {
    do {
      w = s.top(); s.pop();
      put(comp, w, c);
put(root, w, v);
    } while (w != v);
    ++c;
  }
}
```

File: boost/graph/strong_components.hpp
Lines 45-71

Nuutila's 1st Algorithm Implementation

- We modify the BGL implementation of Tarjan's Algorithm

1. We define nuutila_scc_visitor class

2. First traversal: the same

3. Second traversal: Stores only non-root nodes on the stack

Nuutila's 1st Algorithm Implementation

```cpp
template <typename Graph>
void finish_vertex(typename graph_traits<Graph>::vertex_descriptor v,
                   const Graph& g) {
  //Second traversal, stores only nonroot elements on the stack
  typename graph_traits<Graph>::vertex_descriptor w;
  typename graph_traits<Graph>::out_edge_iterator ei, ei_end;
  for (boost::tie(ei, ei_end) = out_edges(v, g); ei != ei_end; ++ei) {
    w = target(*ei, g);
    if (get(comp, w) == (std::numeric_limits<comp_type>::max)())
      put(root, v, this->min_discover_time(get(root,v), get(root,w)));
  }
  if (get(root, v) == v) {
    put(comp, v, c);
    if (! s.empty()){
      while (get(discover_time, s.top())>get(discover_time, v)){
        //Pop under the constrain that TOP(stack)>v
        w = s.top(); s.pop();
        put(comp, w, c);
        put(root, w, v);
        if (s.empty()) {
          break;
        }
      }
    }
    ++c;
  }else{
      // Push after control
      s.push(v);
  }
}
```

File: strcomp1.hpp
Lines: 58-87

Nuutila’s 2nd Algorithm Implementation

- We modify the BGL implementation of Tarjan’s Algorithm

1. We define nuutila2_scc_visitor class
2. Addition of vector inStack to instantly O(1) check if a node exists in stack s
3. Second traversal: Stores only final candidate root nodes on the stack

Nuutila's 2nd Algorithm Implementation

```
template <typename Graph>
void discover_vertex(typename graph_traits<Graph>::vertex_descriptor v,
                     const Graph&) {
  put(root, v, v);
  put(comp, v, (std::numeric_limits<comp_type>::max)());
  put(discover_time, v, dfs_time++);
  // Initialize a stack to contain a value < any node in Graph
  if (dfs_time==1) s.push(-1);
  //  Vector for instant check if v is in the Stack s
  inStack.resize(dfs_time);
  inStack[dfs_time]=false;
}
```

File: strcomp2.hpp
Lines: 50-61

Nuutila’s 2nd Algorithm Implementation

```
template <typename Graph>
void finish_vertex(typename graph_traits<Graph>::vertex_descriptor v,
                   const Graph& g) {
  //Only final candidate root nodes on the stack
  typename graph_traits<Graph>::vertex_descriptor w;
  typename graph_traits<Graph>::out_edge_iterator ei, ei_end;
  for (boost::tie(ei, ei_end) = out_edges(v, g); ei != ei_end; ++ei) {
    w = target(*ei, g);
    // if not InComponent[root[w]] then ... (before: if not InComponent[w]... )
    if (get(comp, get(root, w)) == (std::numeric_limits<comp_type>::max)())
      put(root, v, this->min_discover_time(get(root,v), get(root,w)));
  }
  if (get(root, v) == v) {
    //Check if s is not empty, to use .top()
    if (!s.empty()){
      //Removes if TOP(stack)>= v
      if (s.top()!=-1 && get(discover_time, s.top())>=get(discover_time, v)){
        do{
          w=s.top(); s.pop();
          // Update inStack
          inStack[get(discover_time, w)]=false;
          put(comp, w, c);
          if(s.empty()){
            break;
          }
        }while(s.top()!=-1 && get(discover_time, s.top())>=get(discover_time, v));
      }else{
        //roots of trivial components are not stored on the stack
        //ComponentMap is updated
        put(comp,v,c);
      }
    }
    ++c;
  }else if(!inStack[get(discover_time, get(root,v))]){
      //Stores on stack the final candidate root nodes
      s.push(get(root,v));
      //Update inStack
      inStack[get(discover_time, v)]=true;
  }
```

File: strcomp2.hpp
Lines: 63-102

Pearce's Algorithm Implementation

- As reported on Pearce's paper, a non-recursive implementation of this algorithm is required to obtain the reduced memory requirements in practice.

- So, we implement Algorithm 4 PEA_FIND_SCC3.

Pearce's Algorithm Implementation

- We implemented it based on Pearce's Java imperative implementation from the following link (Reference [6] of the paper): https://github.com/DavePearce/StronglyConnectedComponents/

Pearce’s Algorithm Implementation

```
Graph g;
std::stack<Vertex> vS, S;  // vS impements DFS stack
                           // S implements Backtracking stack
std::stack<int> iS;        // iS is the imperative version of recursive's call stack
std::vector<int> rindex;
std::vector<bool> root;
int index, c;
```

File: pearce1.hpp
Lines: 136-142

Two stacks (vS, S) are needed to implement DFS stack and BackTracking Stack.
Note: The memory usage of them is still vw (worst case) since a node cannot exists in both stacks in parallel.

Pearce’s Algorithm Implementation

```cpp
int scc () {
    //Initialization
    index=1;
    c= num_vertices(g)-1;
    std::pair<vertex_iter, vertex_iter> vp;
    //Loop over vertices
    for (vp = vertices(g); vp.first != vp.second; ++vp.first){
        if (rindex[*vp.first] == 0) {
            visit(*vp.first);
        }
    }
    // Number of SCC
    return num_vertices(g)-c-1;
}
```

File: pearce1.hpp
Lines: 26-39

Pearce’s Algorithm Implementation

```
void visitLoop(){
    Vertex v=vS.top();
    int i=iS.top();
    out_edge_iter ai, ai_end;
    // help is used to store out edges of v
    // Based on Pearce's Java implementation
    std::vector<out_edge_iter> help;
    for (tie(ai, ai_end) = out_edges(v, g); ai != ai_end; ++ai) {
        help.push_back(ai);
    }
    while (i <= out_degree(v,g)){
        if ( i>0 ){
            finishEdge(v, i-1);
        }
        if ( i< out_degree(v,g) && beginEdge(v,i)){
            return;
        }
        i=i+1;
    }
    finishVisiting(v);
}

void visit (Vertex v){
    beginVisiting(v);
    while (!vS.empty()){
        visitLoop();
    }
}
```

File: pearce1.hpp
Lines: 107-134

Pearce’s Algorithm Implementation

```
void finishVisiting(Vertex v){
    //Pop from "DFS" Stack
    vS.pop();
    iS.pop();
    if (root[v]){
        index=index-1;
        while (!S.empty() && (rindex[v] <= rindex [S.top()])){
            //Removes evrything from backtracking stack, same component as v
            Vertex w=S.top();
            S.pop();
            rindex[w]=c;
            index=index-1;
        }
        rindex[v]=c;
        //v's component members found and marked
        c=c-1;
    }else{
        //v is not a root, store it on backtracking stack
        S.push(v);
    }
}

void beginVisiting(Vertex v){
    //Store on stack
    vS.push(v);
    iS.push(0);
    root[v]=true;
    rindex[v]=index;
    index=index+1;
}
```

File: pearce1.hpp
Lines: 75-104

Pearce’s Algorithm Implementation

```cpp
void finishEdge(Vertex v, int k){
    //Create help, to find the k-th out-edge of v
    out_edge_iter ai, ai_end;
    std::vector<out_edge_iter> help;
    for (tie(ai, ai_end) = out_edges(v, g); ai != ai_end; ++ai) {
        help.push_back(ai);
    }
    Vertex w= target (*help[k],g) ;
    if (rindex[w]< rindex[v]){
        rindex[v]=rindex[w];
        root[v]=false;
    }
}

bool beginEdge(Vertex v, int k){
    //Create help, to find the k-th out-edge of v
    out_edge_iter ai, ai_end;
    std::vector<out_edge_iter> help;
    for (tie(ai, ai_end) = out_edges(v, g); ai != ai_end; ++ai) {
        help.push_back(ai);
    }
    Vertex w= target (*help[k],g) ;
    if (rindex[w]==0){
        iS.pop();
        iS.push(k+1);
        beginVisiting(w);
        return true;
    }else{
        return false;
    }
}
```

File: pearce1.hpp
Lines: 43-73

Testing the algorithms: Runtime results

Runtimes:

	Tarjan	*Nuutila 1*	*Nuutila 2*	*Pearce*
n ≈ 1000 e ≈ 1000 (g1000_1000.txt) (sparse graph)	Result: 683 Runtime: Real: 0m0.016s User: 0m0.014s Sys: 0m0.002s	Result: 683 Runtime: Real: 0m0.011s User: 0m0.008s Sys: 0m0.004s	Result: 683 Runtime: Real: 0m0.009s User: 0m0.005s Sys: 0m0.004s	Result: 683 Runtime: Real: 0m0.017s User: 0m0.017s Sys: 0m0.001s
n ≈ 1000 e ≈ 500 (ag1k_100k.txt) (sparse graph)	Result: 8 Runtime: Real: 0m0.019s User: 0m0.015s Sys: 0m0.004s	Result: 8 Runtime: Real: 0m0.019s User: 0m0.012s Sys: 0m0.007s	Result: 8 Runtime: Real: 0m0.016s User: 0m0.016s Sys: 0m0.000s	Result: 8 Runtime: Real: 0m0.035s User: 0m0.032s Sys: 0m0.003s
n ≈ 1000 e ≈ 100000 (ag1k_100k.txt) (dense graph)	Result: 1 Runtime: Real: 0m0.156s User: 0m0.157s Sys: 0m0.000s	Result: 1 Runtime: Real: 0m0.157s User: 0m0.153s Sys: 0m0.004s	Result: 1 Runtime: Real: 0m0.157s User: 0m0.148s Sys: 0m0.008s	Result: 8 Runtime: Real: 0m1.805s User: 0m1.797s Sys: 0m0.008s

Testing the algorithms: Runtime results

Runtimes:

	Tarjan	Nuutila 1	Nuutila 2	Pearce
n ≈ 1000 e ≈ 10500 (g10k_100k.txt) (sparse graph)	Result: 9973 Runtime: Real: 0m0.046s User: 0m0.047s Sys: 0m0.000s	Result: 9973 Runtime: Real: 0m0.043s User: 0m0.043s Sys: 0m0.000s	Result: 9973 Runtime: Real: 0m0.044s User: 0m0.036s Sys: 0m0.006s	Result: 9973 Runtime: Real: 0m0.078s User: 0m0.074s Sys: 0m0.004s
n ≈ 10000 e ≈ 1050000 (g10k_10500k.txt) (dense graph)	Result: 1 Runtime: Real: 0m1.597s User: 0m1.561s Sys: 0m0.036s	Result: 1 Runtime: Real: 0m1.585s User: 0m1.564s Sys: 0m0.020s	Result: 1 Runtime: Real: 0m1.624s User: 0m1.583s Sys: 0m0.040s	Result: 1 Runtime: Real: 0m19.72s User: 0m19.64s Sys: 0m0.080s
n ≈ 100000 e ≈ 100000 (g100k_100k.txt) (sparse graph)	Result: 99977 Runtime: Real: 0m0.382s User: 0m0.377s Sys: 0m0.004s	Result: 99977 Runtime: Real: 0m0.371s User: 0m0.359s Sys: 0m0.013s	Result: 99977 Runtime: Real: 0m0.413s User: 0m0.402s Sys: 0m0.012s	Result: 99977 Runtime: Real: 0m0.748s User: 0m0.739s Sys: 0m0.008s

Comments on the runtimes:

- Pearce’s runtimes are always worse (as expected).
- Between Tarjan and Nuutila Algorithms’ runtimes, **in dense graphs**, we do not expect much differences. This happens since in that case the second travelsal cannot be optimised (with the described by the algorithm improvements).
- Between Tarjan and Nuutila Algorithms’ runtimes, **in sparse graphs**, we expected Nuutila to achieve better runtimes than Tarjan. This does not happen (possible reason: implementation issues, not worst case graph).

Testing the algorithms: Memory Usage

We mesure the actual memory footprint using Valgrind.

```
--------------------------------------------------------------------------------
Command:            ./create1 10000 105000 g10k_105k.txt
Massif arguments:   (none)
ms_print arguments: massif.out.1282
--------------------------------------------------------------------------------


    MB
1.816^                                     : :  :
     |                                 ##:::::::::::::@:::::@:::::@:::::@:::
     |                                 # :::::::::::::@:::::@:::::@:::::@:::
     |                                @# :::::::::::::@:::::@:::::@:::::@:::
     |                             @@@@# :::::::::::::@:::::@:::::@:::::@::::
     |                          :@@@@ @# :::::::::::::@:::::@:::::@:::::@::::
     |                        @@@@@@@ @# :::::::::::::@:::::@:::::@:::::@::::
     |                     @@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@:::::
     |                   :@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@:::::
     |                @@@:@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@:::::
     |              @:@@@:@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@:::::
     |            :@@:@@@:@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@:::::
     |         @@@:@@:@@@:@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@:::::
     |      ::@@ @:@@:@@@:@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@:::::
     | :::::::@@ @:@@:@@@:@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@:::::
     | :  ::::@@ @:@@:@@@:@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@:::::
     | :  ::::@@ @:@@:@@@:@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@:::::
     | :  ::::@@ @:@@:@@@:@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@:::::
     | :  ::::@@ @:@@:@@@:@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@::::@
     | :  ::::@@ @:@@:@@@:@@@@@@@@@@@ @# :::::::::::::@:::::@:::::@:::::@::::@
   0 +----------------------------------------------------------------------->Mi
     0                                                                   142.6
```

Testing the algorithms: Memory Usage

Memory Usage:

	Tarjan	Nuutila 1	Nuutila 2	Pearce
n ≈ 1000 e ≈ 1000	316 KB	315 KB	317 KB	504KB
n ≈ 10000 e ≈ 100000	1.81 MB	1.81 MB	1.82 MB	3.66MB
n ≈ 100000 e ≈ 100000	7.32 MB	7.32 MB	7.33 MB	17.10MB
n ≈ 10000000 e ≈ 10000000	1.65 GB	-	-	3.43MB

Comments on the usage of memory:

- Between Tarjan's and Nuutila's Algorithms there is not observed much difference in memory usage. We expected this results since in our implementations we didn't focused on decreasing the memory usage in Nuutila's Algorithms.
- Pearce's Algorithm was expected to use less memory tan the other algorithms (since less vectors are used). This does not happen (possible reason: implementation issues, not worst case graph, usage of extra memory –help[.] for finding neighbours in Pearce's non-recursive version).
- We observed that Pearce's memory usage in increased with lower pace than Tarjan.