A Method to Evaluate CFG Comparison Algorithms

Patrick P.F. Chan
Christian Collberg

Research problem

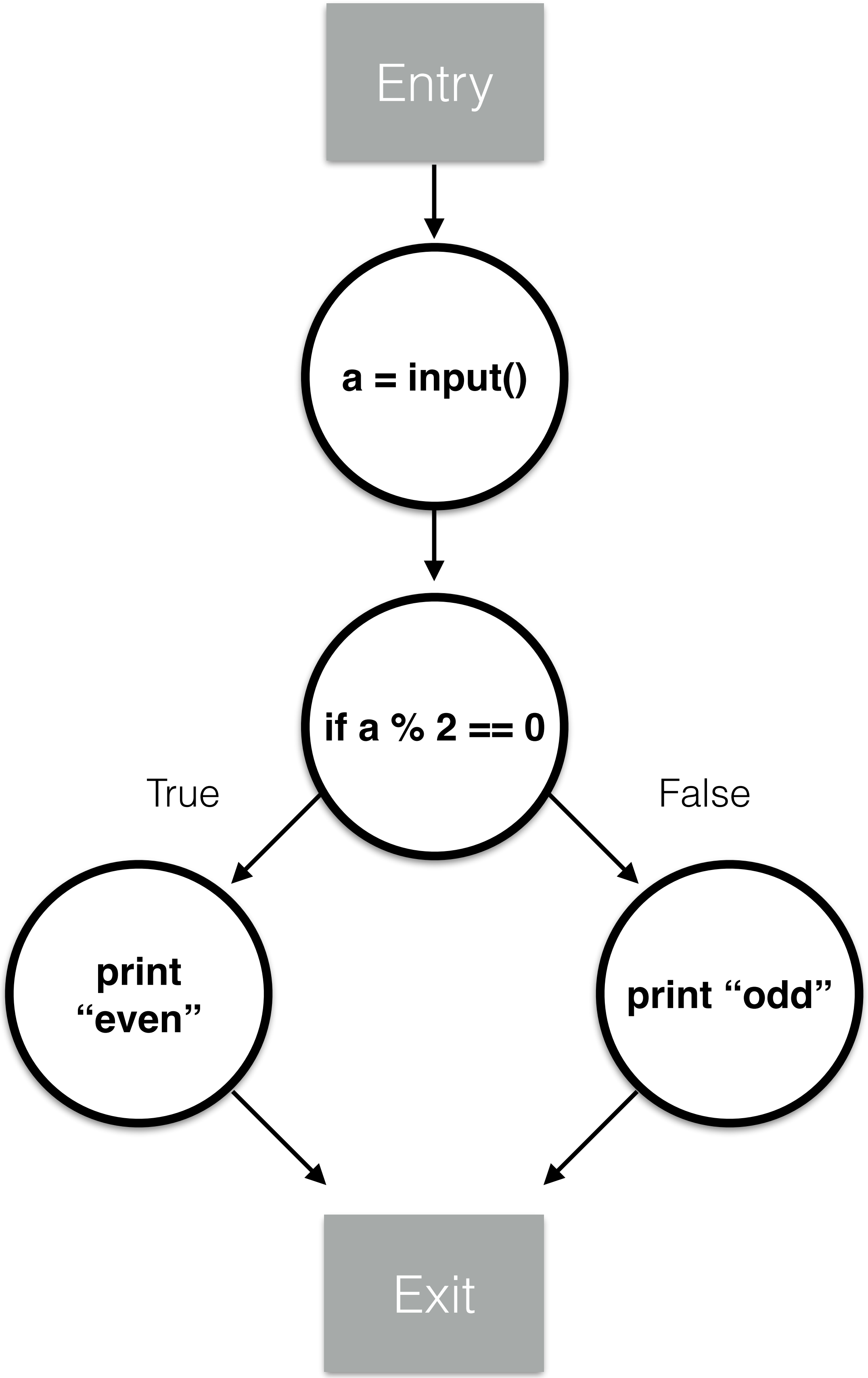
- Which CFG similarity algorithm is better?
- I come up with a new algorithm, how does it compare to the existing ones?
- Is there a systematic way to compare CFG similarity algorithms?

Research outcomes

- A methodology to evaluate and compare CFG similarity algorithms
- Comparison results of four CFG similarity algorithms
- A survey of existing CFG similarity algorithms
- A publicly available evaluation framework

What is CFG?

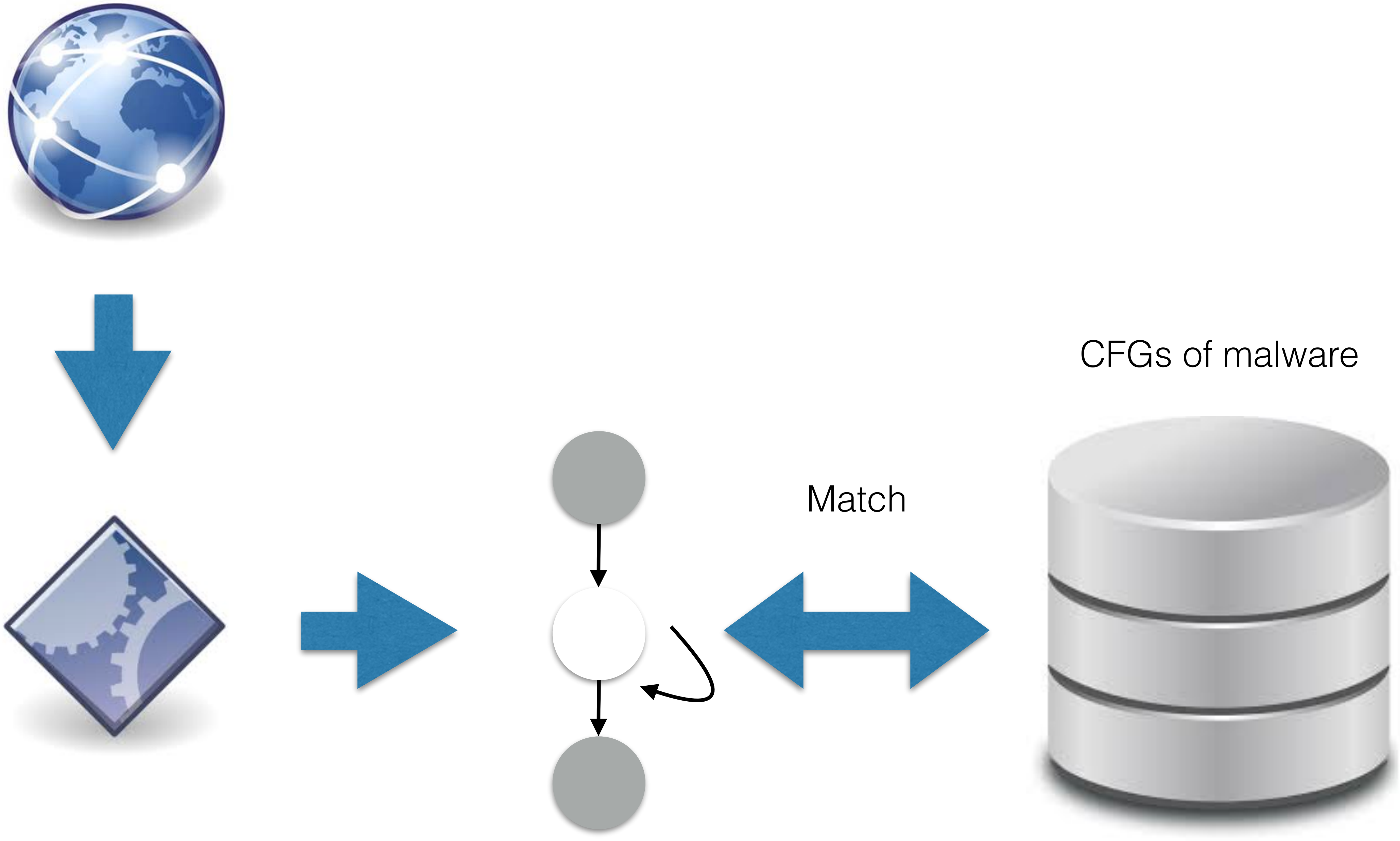
- CFG stands for **c**ontrol-**f**low **g**raph
- A CFG represents all possible execution paths of a function
- And thus, it encodes its behavior


Entry
a = input()
if a % 2 == 0
True
False
print “even”
print “odd”
Exit

Why do we compare CFGs?

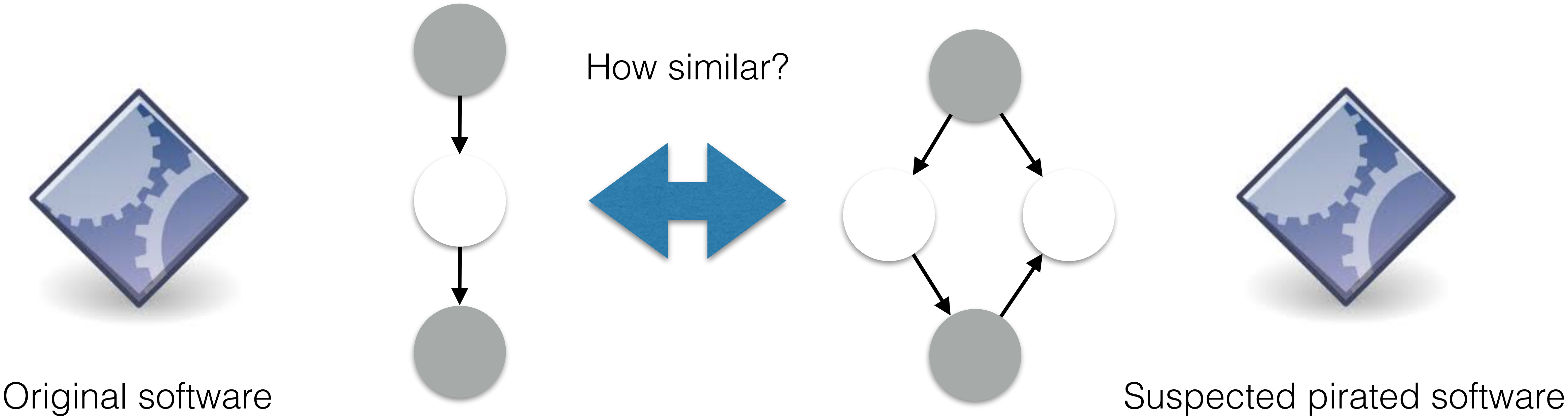
Why do we compare CFGs?

- Malware detection / classification



Why do we compare CFGs?

- Software theft detection



Original software

How similar?

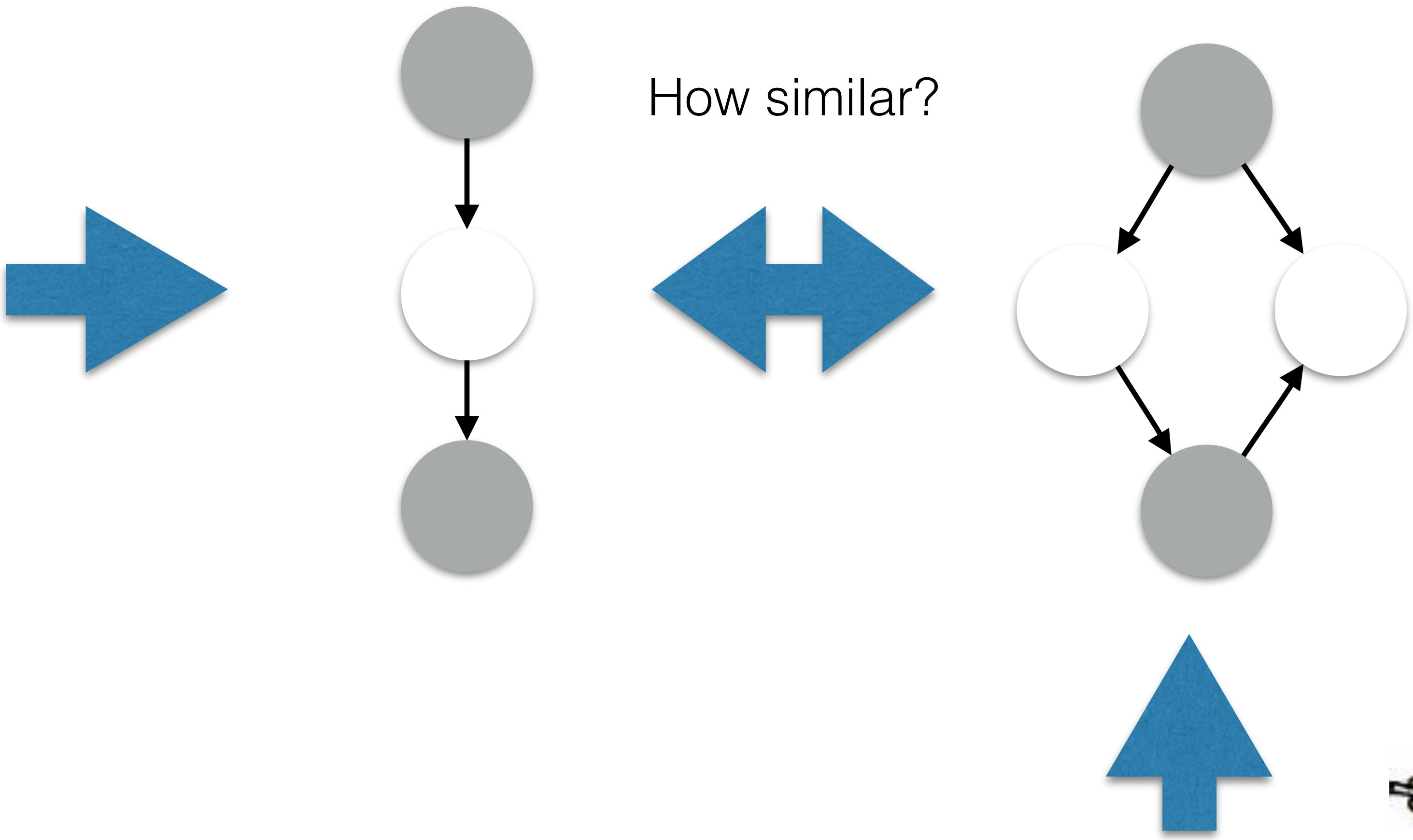
Suspected pirated software

Why do we compare CFGs?

- Programming assignments grading

```
#include <iostream>
using namespace std;
int main() {
  int value1 = 3000000000;
  int value2, value3;
  char middleInitial = "G";
  boolean 4me=true;
  double capacity=14.89;
  string name = 'Michelle';

  cout << "value1 = " << value1 << endl;
  value2 = capacity;
  cout << "value2 = " << value2 << endl;
  cout << "middleInitial = " << middleInitial << endl;
  13=value3;
  cout << "value3 = " << value3 << endl;
  cout << "4me = " << 4me << endl;
  cout << "name = " << name << endl;
  return 0;
}
```

Assignment Submission



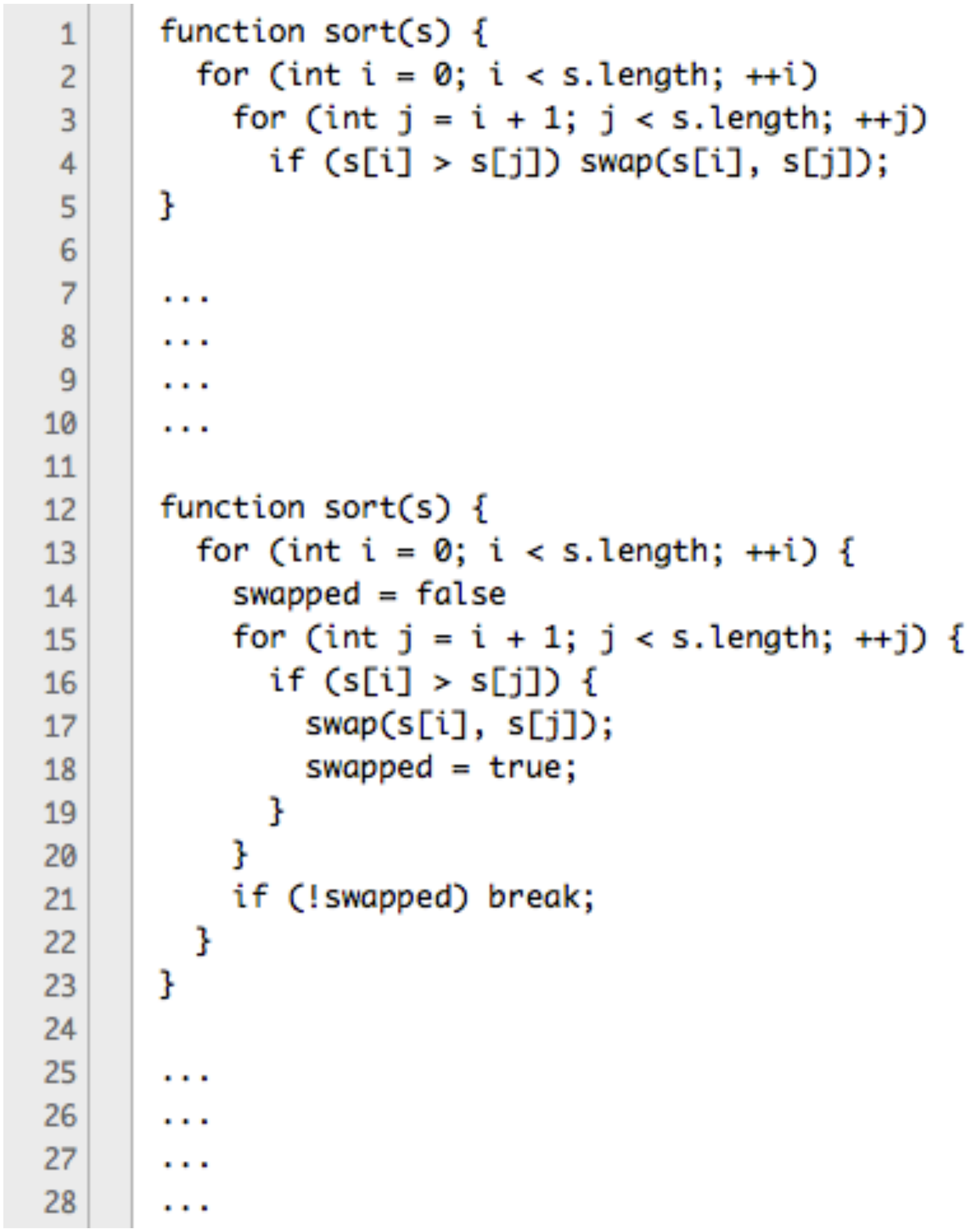
How similar?

```
Program Check_Group
  use crystallographic_symmetry, only: Space_Group_Type, set_spacegroup
  use reflections_utilities, only: Hkl_Absent
  use Symmetry_Tables, only: spgr_info, Set_Spgr_Info

  ....... ! Read reflections, apply criteria of "goodness" for checking,
          ! set indices i1,i2 for search in space group tables ...
  ....... ! omitted for simplicity
  call Set_Spgr_Info()
  m=0
  do_group: do i=i1,i2
    hms=adjustl(spgr_info(i)%HM)
    hall=spgr_info(i)%hall
    if( hms(1:1) /= "P" .and. .not. check_cent ) cycle do_group ! Skip centred groups
    call set_spacegroup(hall,Spacegroup,Force_Hall="y")
    do j=1,nhkl
      if(good(j) == 0) cycle  !Skip reflections that are not good (overlap) for checking
      absent=Hkl_Absent(hkl(:,j), Spacegroup)
      if(absent .and. intensity(j) > threshold) cycle do_group  !Group not allowed
    end do
    ! Passing here means that all reflections are allowed in the group -> Possible group!
    m=m+1
    num_group(m)=i
  end do  do_group
  write(unit=*,fmt=*) " => LIST OF POSSIBLE SPACE GROUPS, a total of ",m," groups are possible"
  write(unit=*,fmt=*) "     ------------------------------------------------------"
  write(unit=*,fmt=*) "     Number(IT)      Hermann-Mauguin Symbol     Hall Symbol"
  write(unit=*,fmt=*) "     ------------------------------------------------------"
  do i=1,m
    j=num_group(i)
    hms=adjustl(spgr_info(j)%HM)
    hall=spgr_info(j)%hall
    numg=spgr_info(j)%N
    write(unit=*,fmt="(i10,4a)") numg,"              ",hms,"        ",hall
  end do
  ...............
```

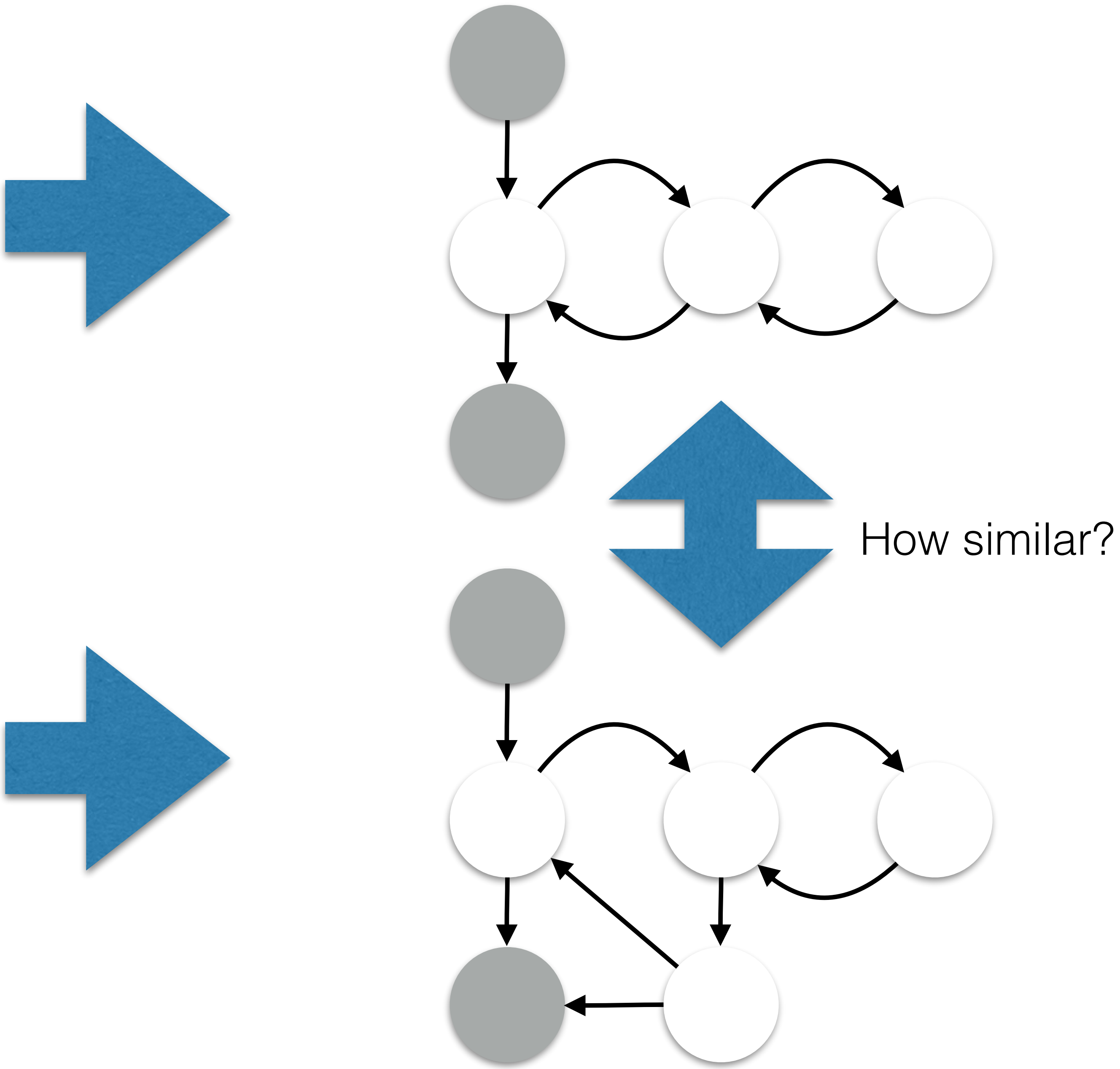
Solution

Why do we compare CFGs?

- Code clones detection

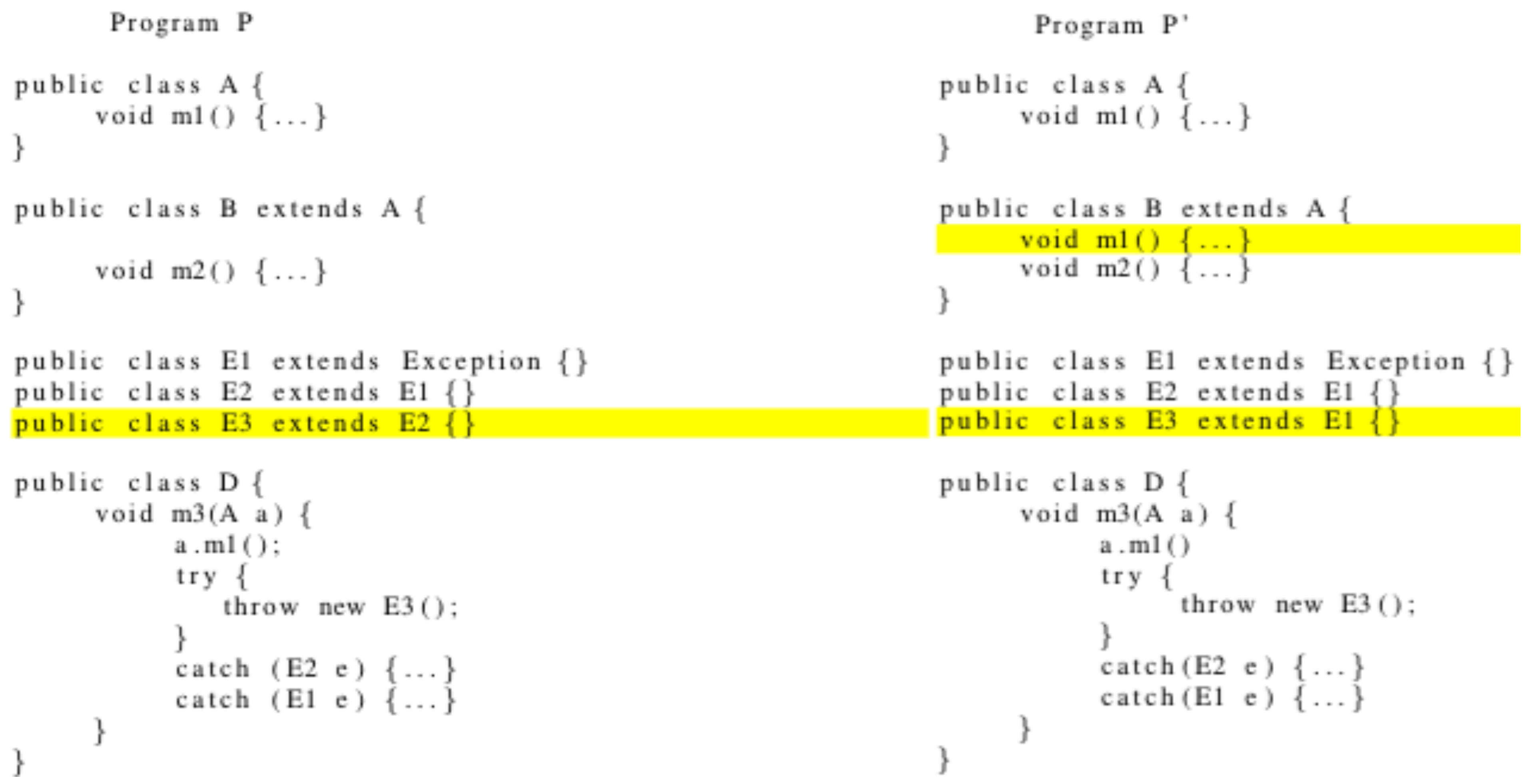


```
function sort(s) {
  for (int i = 0; i < s.length; ++i)
    for (int j = i + 1; j < s.length; ++j)
      if (s[i] > s[j]) swap(s[i], s[j]);
}

...
...
...
...

function sort(s) {
  for (int i = 0; i < s.length; ++i) {
    swapped = false
    for (int j = i + 1; j < s.length; ++j) {
      if (s[i] > s[j]) {
        swap(s[i], s[j]);
        swapped = true;
      }
    }
    if (!swapped) break;
  }
}

...
...
...
...
```



Why do we compare CFGs?

- Detection of changes between different versions of a program



Program P

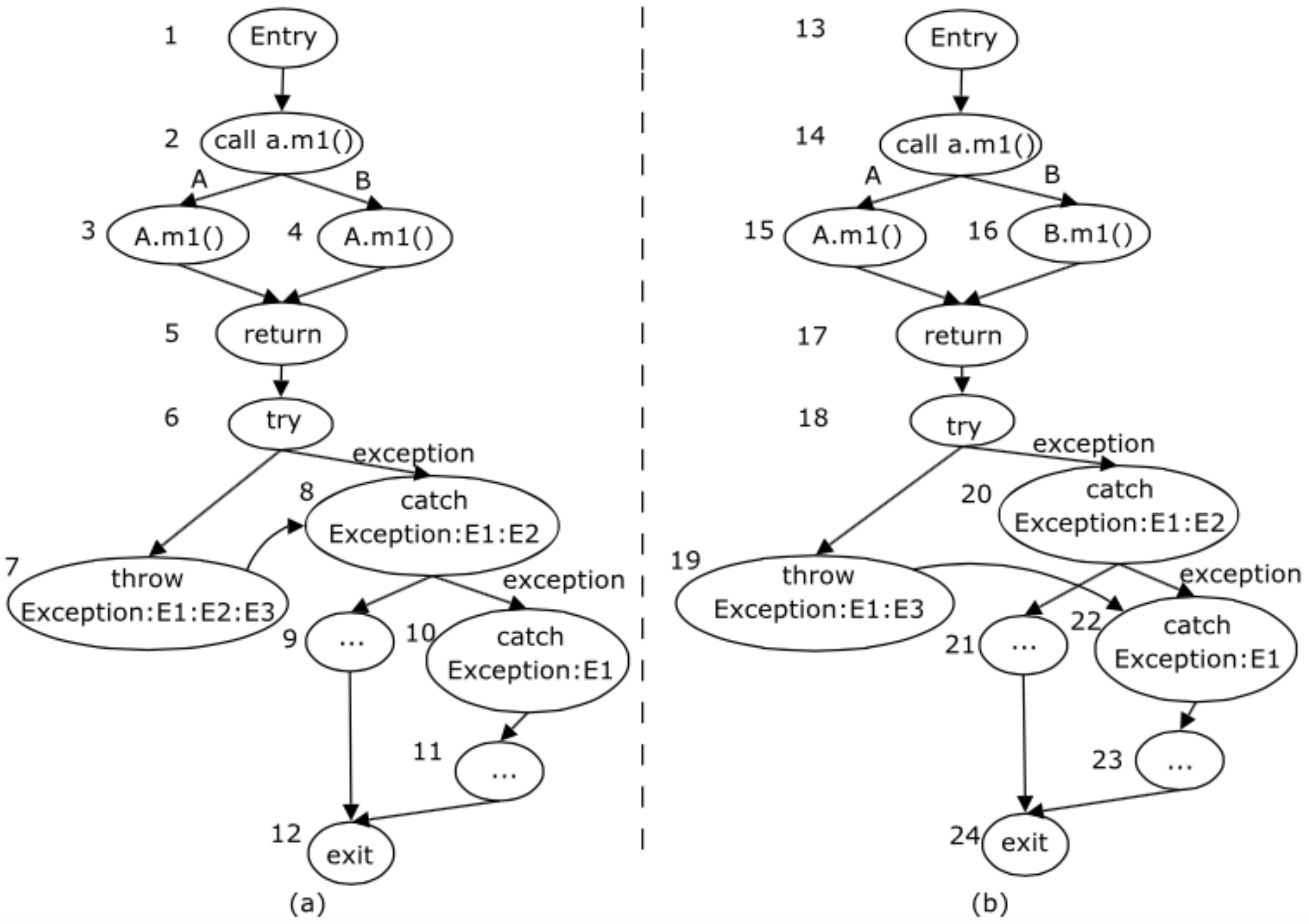
```
public class A {
      void m1() {...}
}

public class B extends A {

      void m2() {...}
}

public class E1 extends Exception {}
public class E2 extends E1 {}
public class E3 extends E2 {}

public class D {
      void m3(A a) {
            a.m1();
            try {
                  throw new E3();
            }
            catch (E2 e) {...}
            catch (E1 e) {...}
      }
}
```

Program P'

```
public class A {
      void m1() {...}
}

public class B extends A {
      void m1() {...}
      void m2() {...}
}

public class E1 extends Exception {}
public class E2 extends E1 {}
public class E3 extends E1 {}

public class D {
      void m3(A a) {
            a.m1()
            try {
                  throw new E3();
            }
            catch(E2 e) {...}
            catch(E1 e) {...}
      }
}
```

Why do we compare CFGs?

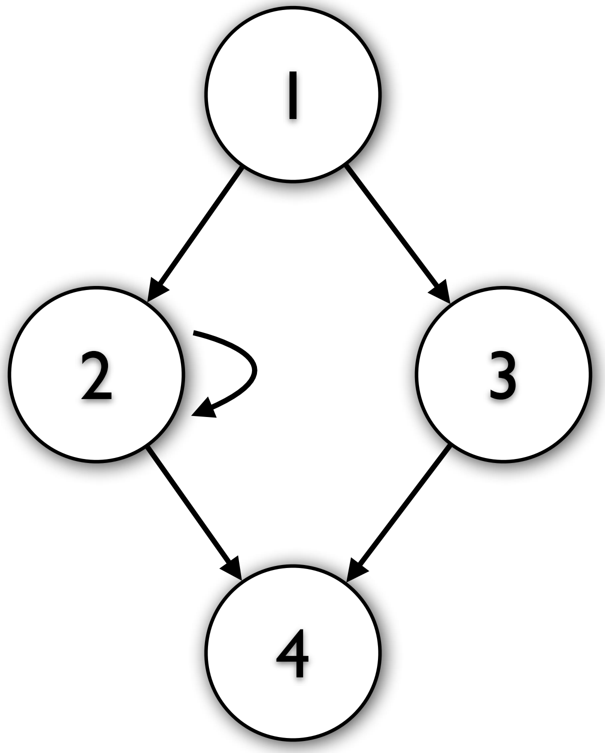
- Detection of changes between different versions of a program



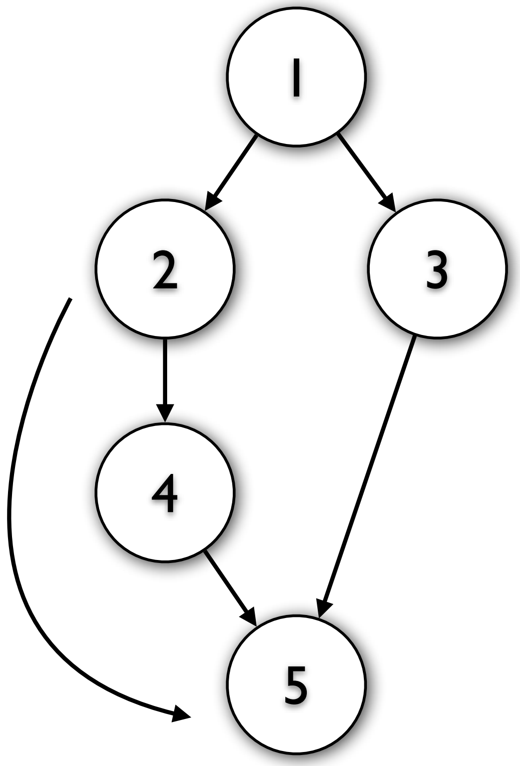
Match the nodes of the enhanced CFGs

This leads to many algorithms to compare CFGs…

Let’s use two existing algorithms to compare these two CFGs



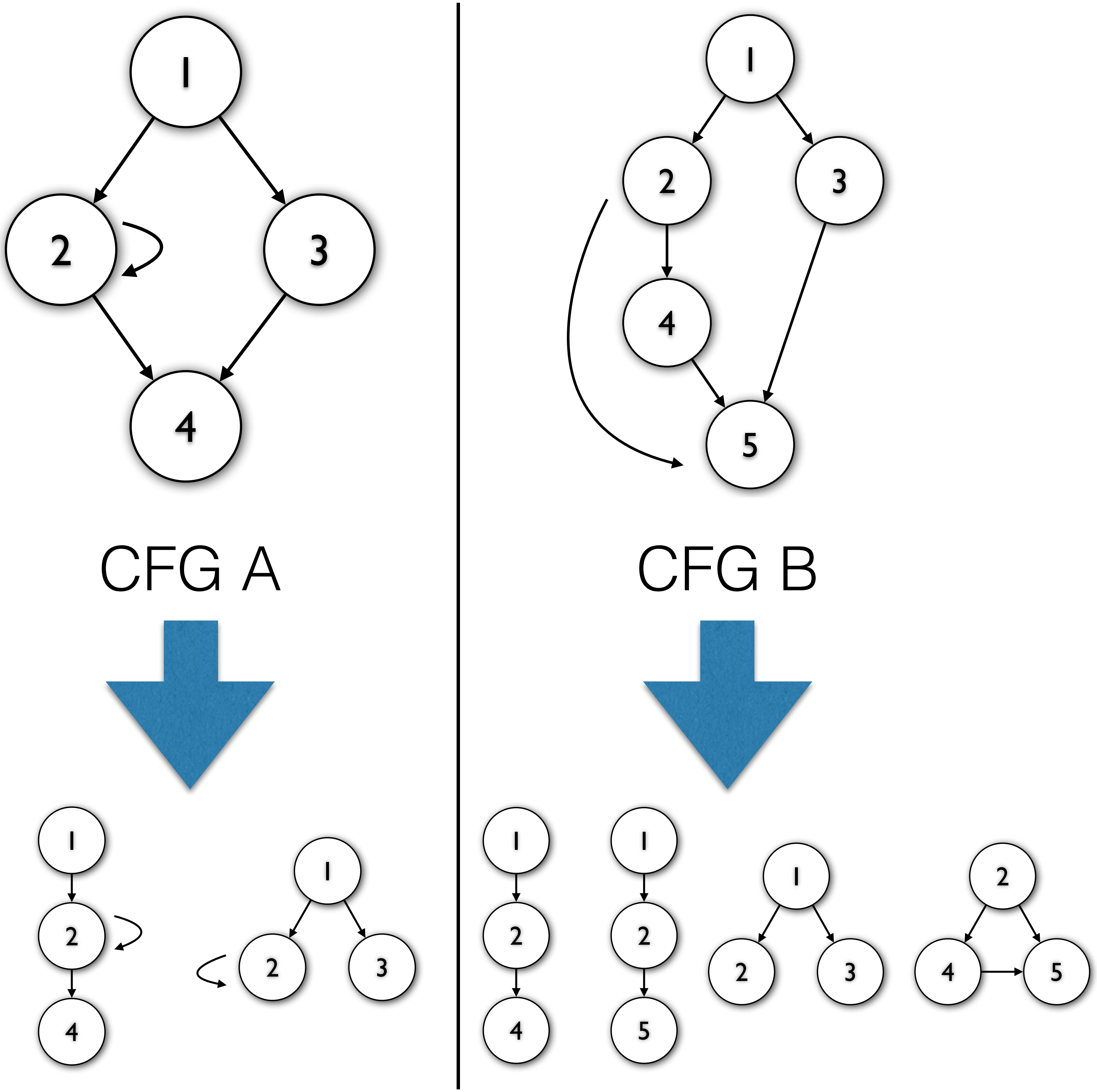
CFG A



CFG B

Algorithm 1 from Kruegel et al.

- Extract subgraphs that have k nodes (k-subgraphs) from CFGs and match them



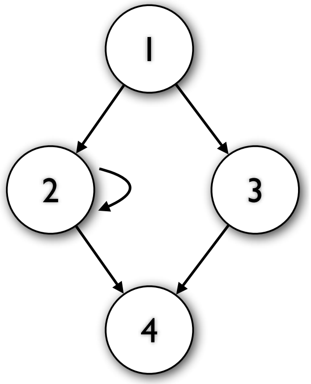
CFG A

CFG B

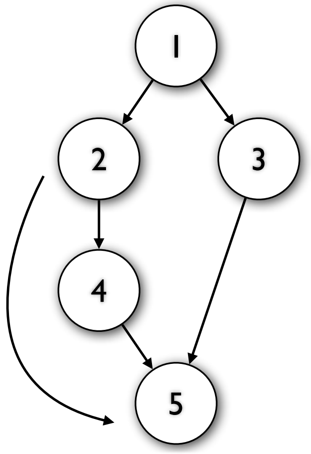
No match!

Algorithm 2 from Hu et al.

- Approximates the minimum number of edit operations needed to transform one graph into another graph



CFG A



CFG B

Cost of matching nodes

Cost of deleting nodes in CFG A

Cost of matching node 1 of CFG A to node 1 of CFG B

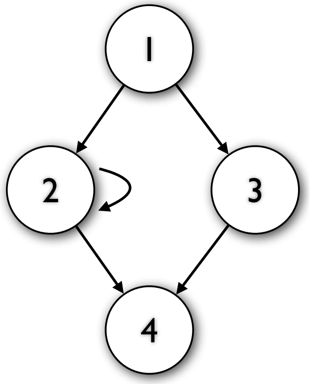
$$\begin{pmatrix} 0 & 1 & 2 & 2 & 5 & 3 & \infty & \infty & \infty \\ 2 & 1 & 2 & 2 & 3 & \infty & 5 & \infty & \infty \\ 2 & 1 & 0 & 0 & 3 & \infty & \infty & 3 & \infty \\ 4 & 3 & 2 & 2 & 1 & \infty & \infty & \infty & 3 \\ 3 & \infty & \infty & \infty & \infty & 0 & 0 & 0 & 0 \\ \infty & 4 & \infty & \infty & \infty & 0 & 0 & 0 & 0 \\ \infty & \infty & 3 & \infty & \infty & 0 & 0 & 0 & 0 \\ \infty & \infty & \infty & 3 & \infty & 0 & 0 & 0 & 0 \\ \infty & \infty & \infty & \infty & 4 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Cost of deleting node 4 of CFG B

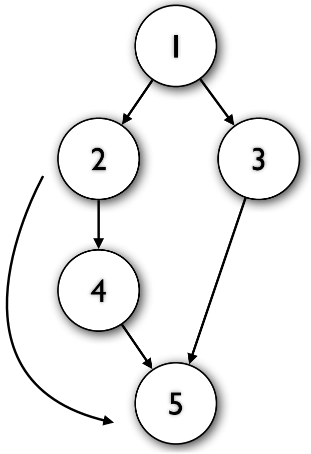
Cost of deleting node 1 of CFG B

Cost of deleting nodes in CFG B

Cost of matching dummy nodes



CFG A



CFG B

$$\begin{pmatrix}
0 & 1 & 2 & 2 & 5 & 3 & \infty & \infty & \infty \\
2 & 1 & 2 & 2 & 3 & \infty & 5 & \infty & \infty \\
2 & 1 & 0 & 0 & 3 & \infty & \infty & 3 & \infty \\
4 & 3 & 2 & 2 & 1 & \infty & \infty & \infty & 3 \\
3 & \infty & \infty & \infty & \infty & 0 & 0 & 0 & 0 \\
\infty & 4 & \infty & \infty & \infty & 0 & 0 & 0 & 0 \\
\infty & \infty & 3 & \infty & \infty & 0 & 0 & 0 & 0 \\
\infty & \infty & \infty & 3 & \infty & 0 & 0 & 0 & 0 \\
\infty & \infty & \infty & \infty & 4 & 0 & 0 & 0 & 0
\end{pmatrix}$$

Total cost = 5

And there are many other algorithms…

- Algorithm from Vujosˇevic´-Janicˇic´ et al. iteratively builds a similarity matrix between the nodes of the two CFGs, based on the similarity of their neighbor

- Algorithm from Sokolsky et al. models the control flow graphs using *Labeled Transition Systems* (LTS)

But which one is the best?

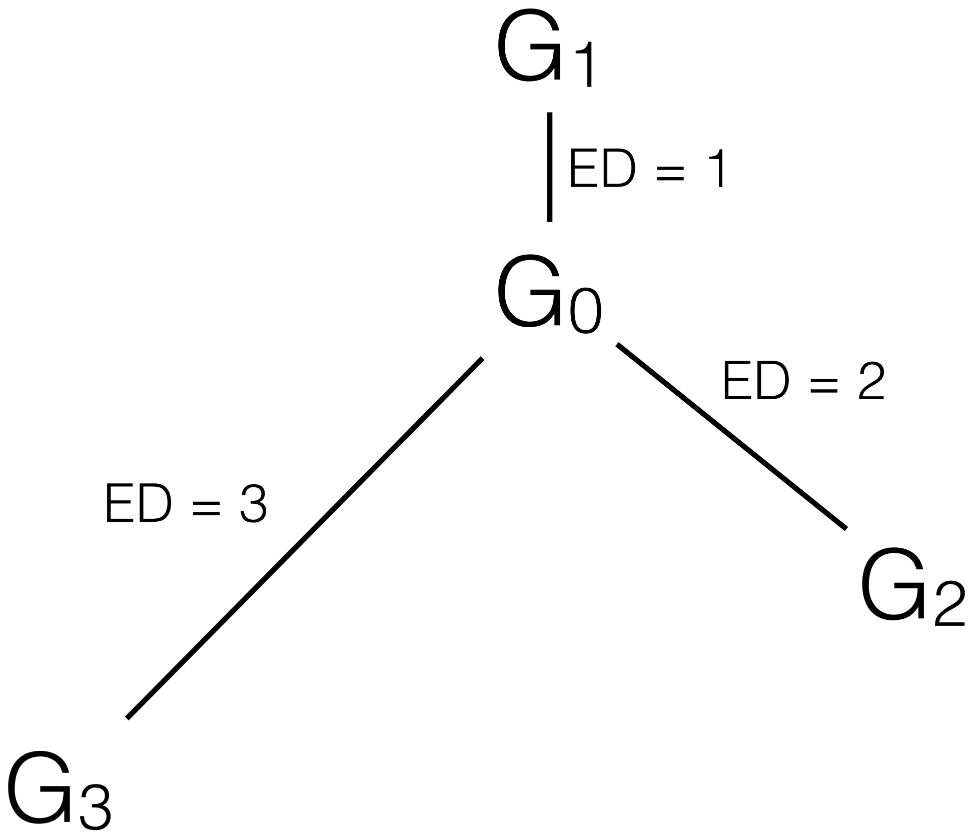
Evaluation of CFG similarity algorithms

- Start by generating CFGs G_1, G_2,...,G_i with increasing edit distances with respect to a seed CFG G_0
    - i.e. $ED(G_0,G_i) = i$
- Use the algorithm under evaluation to rank the CFGs such that the higher is the similarity score between G_i and G_0 given by that algorithm, the higher G_i is ranked
- Get a “goodness score” for the algorithm by comparing the ranking it produces to the ground truth $\langle G_1, G_2, G_3,\ldots\rangle$, using ranking correlation algorithms such as sortedness or Pearson correlation

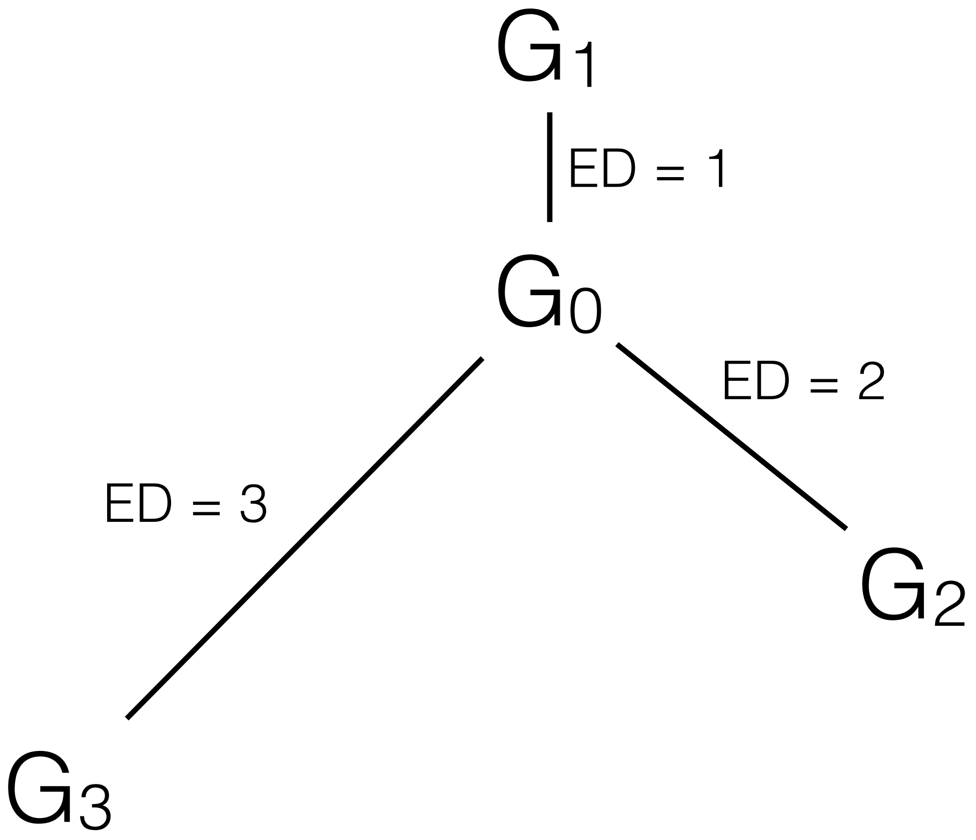
Example

G_0

Example

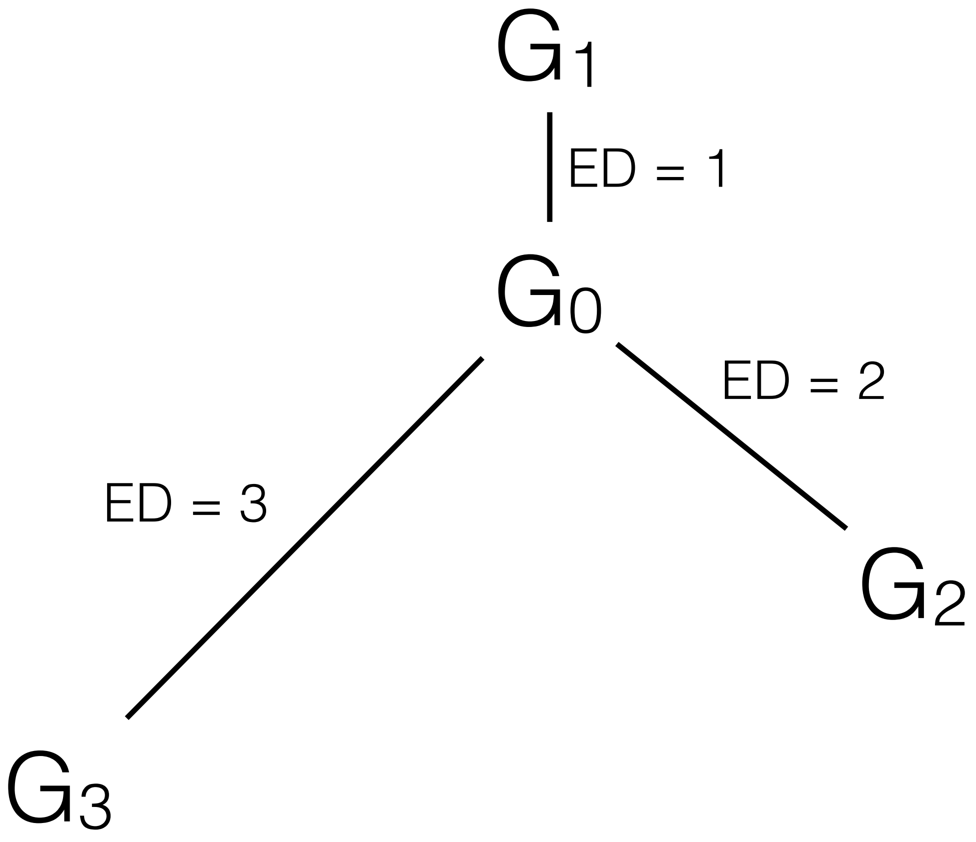


Example

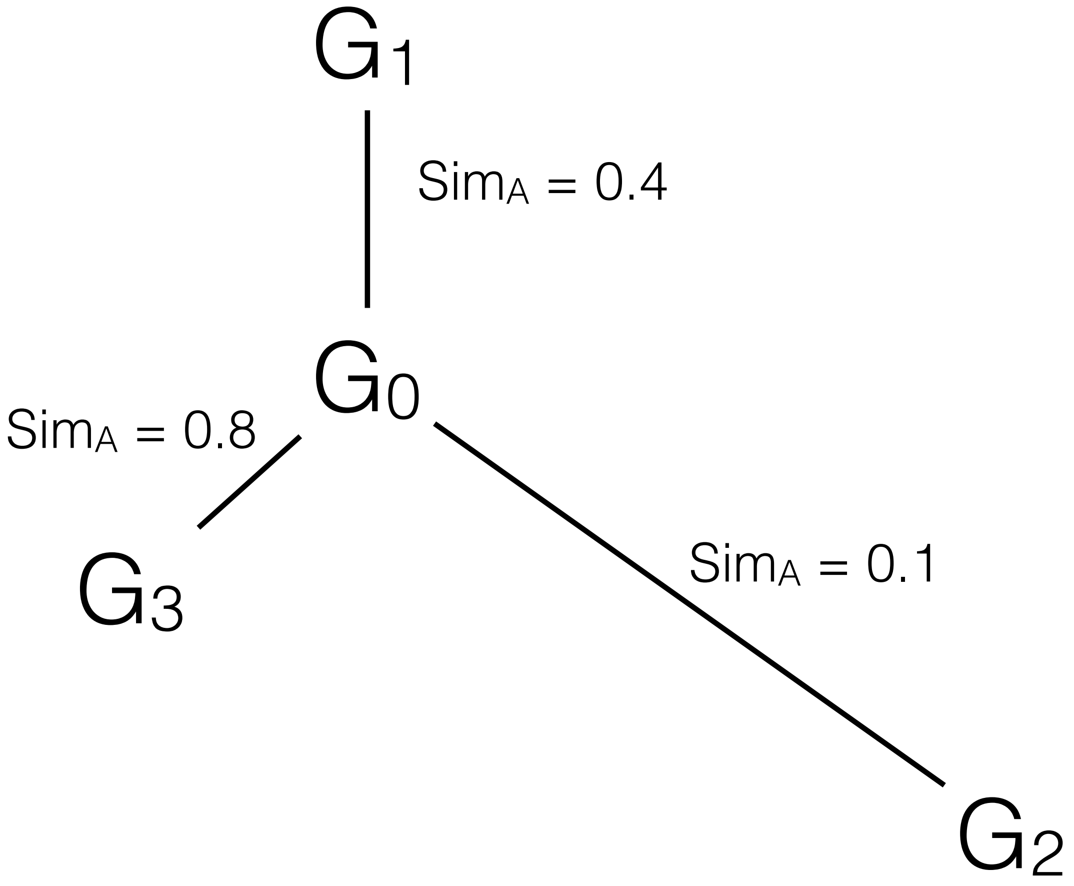


Ranking: $\langle G_1, G_2, G_3 \rangle$

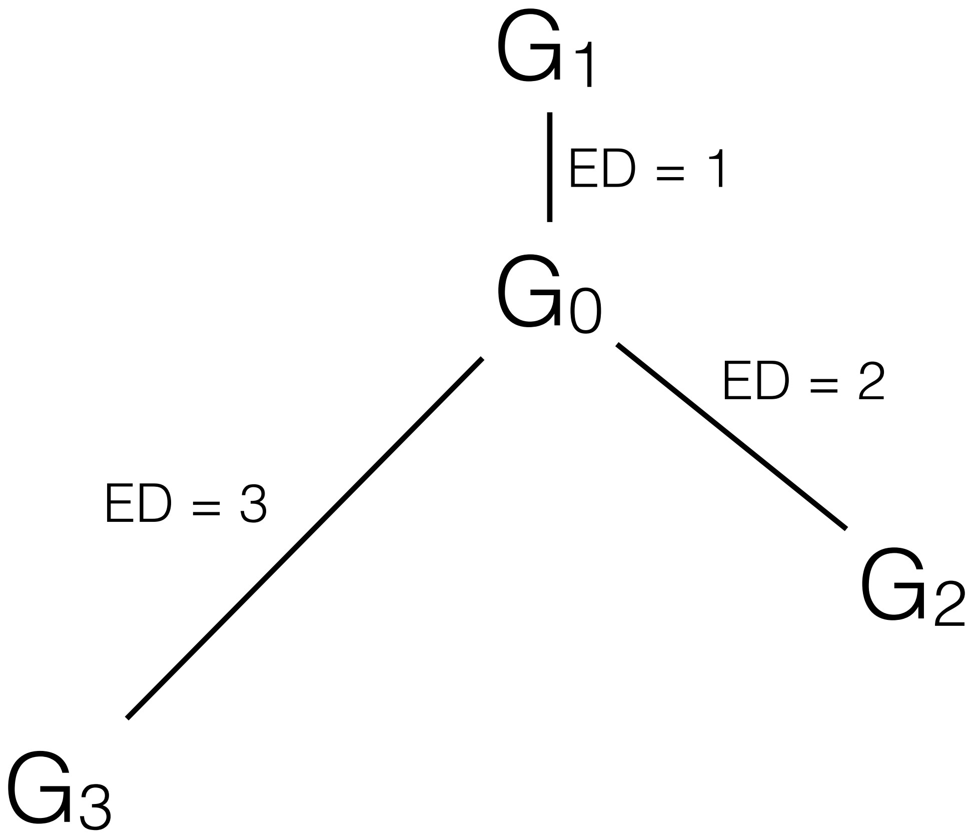
Example



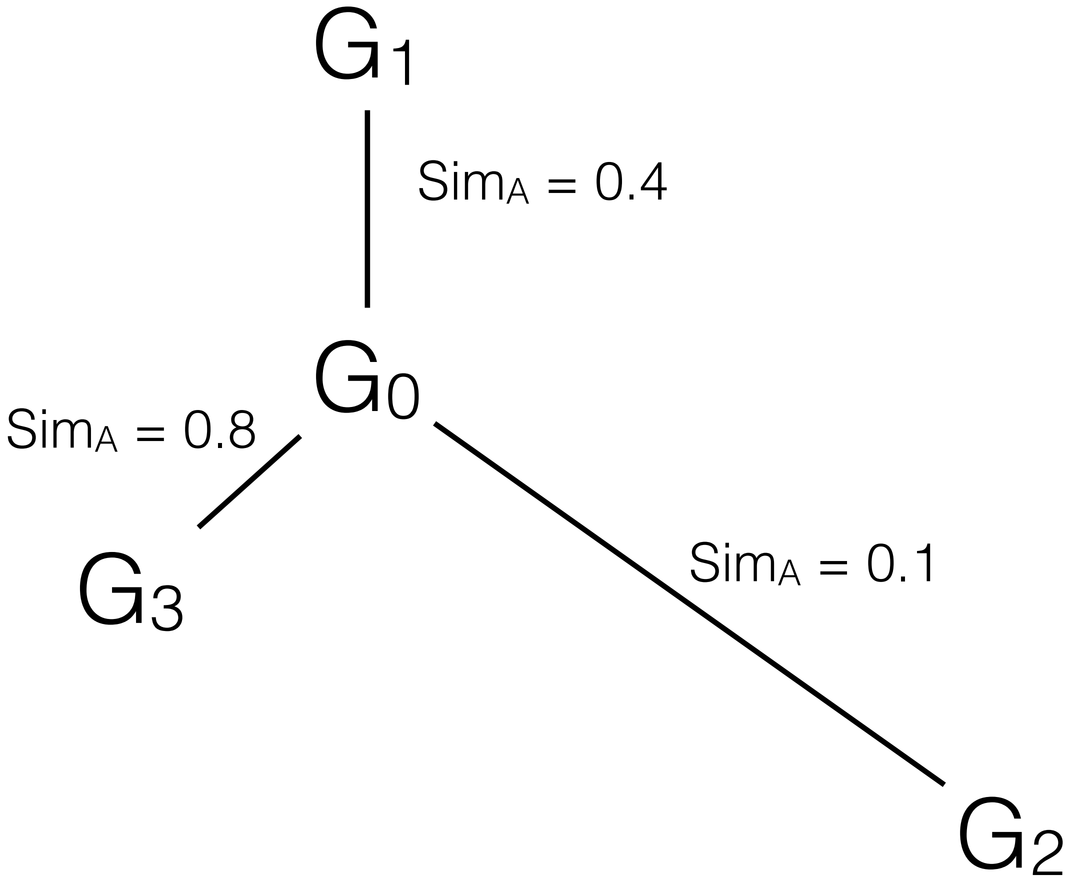
Ranking: ⟨G_1, G_2, G_3⟩



Example

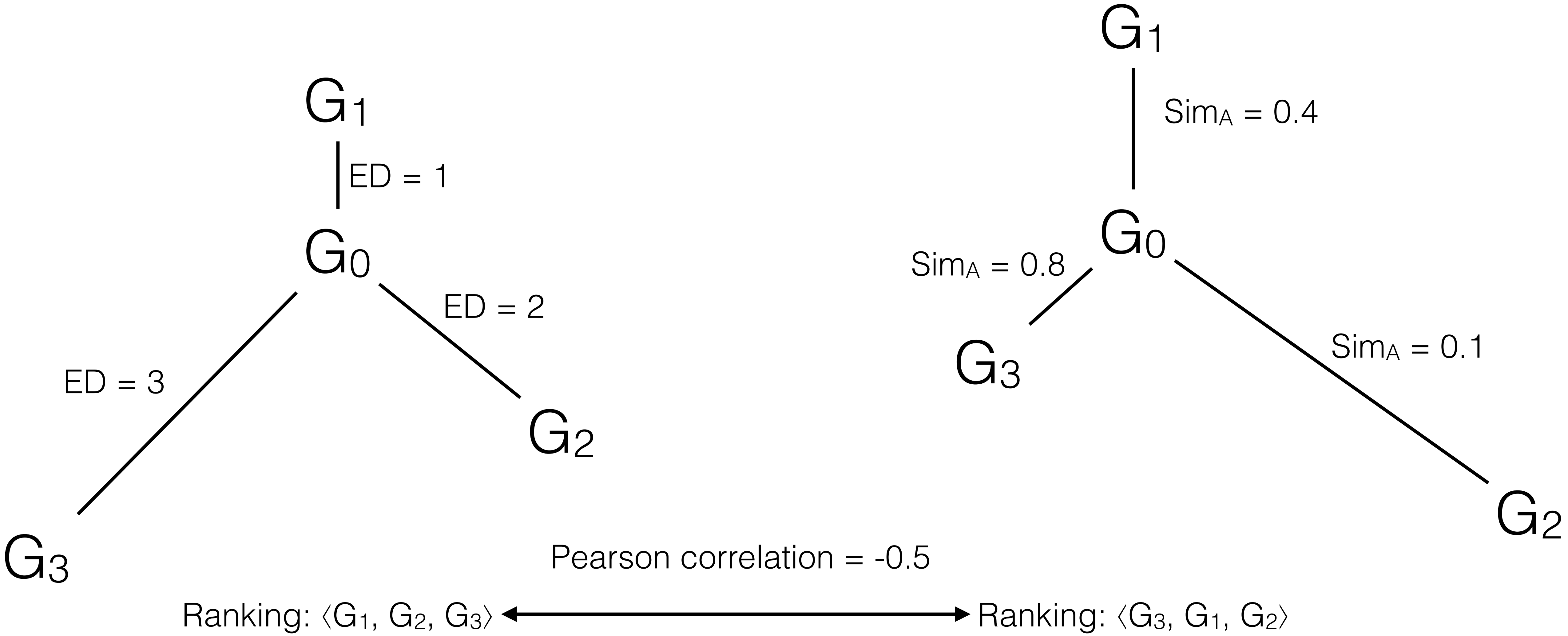


Ranking: $\langle G_1, G_2, G_3 \rangle$



Ranking: $\langle G_3, G_1, G_2 \rangle$

Example



Pearson correlation = -0.5

Ranking: $\langle G_1, G_2, G_3 \rangle$ ⟷ Ranking: $\langle G_3, G_1, G_2 \rangle$

Two questions remain…

1. What is the definition of the edit distance between two CFGs?

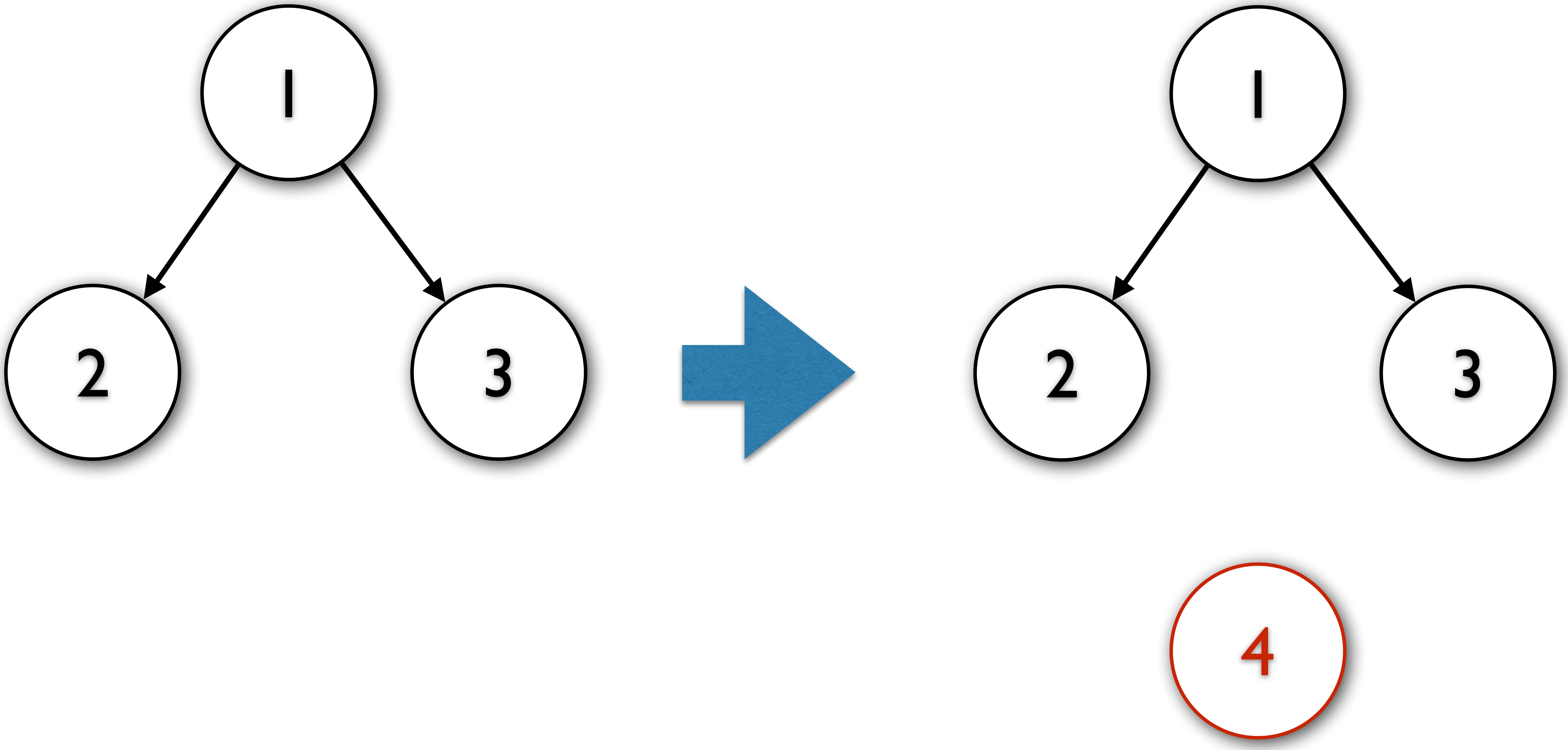
2. How to generate those CFGs such that they have increasing edit distances with the seed CFG G_0?

What is the definition of the edit distance between two CFGs?

- The Graph Edit Distance is a function ED : (G_i, G_j) → N that computes the smallest number of edit operations needed to transform Gi into Gj.
- There are four possible edit operations

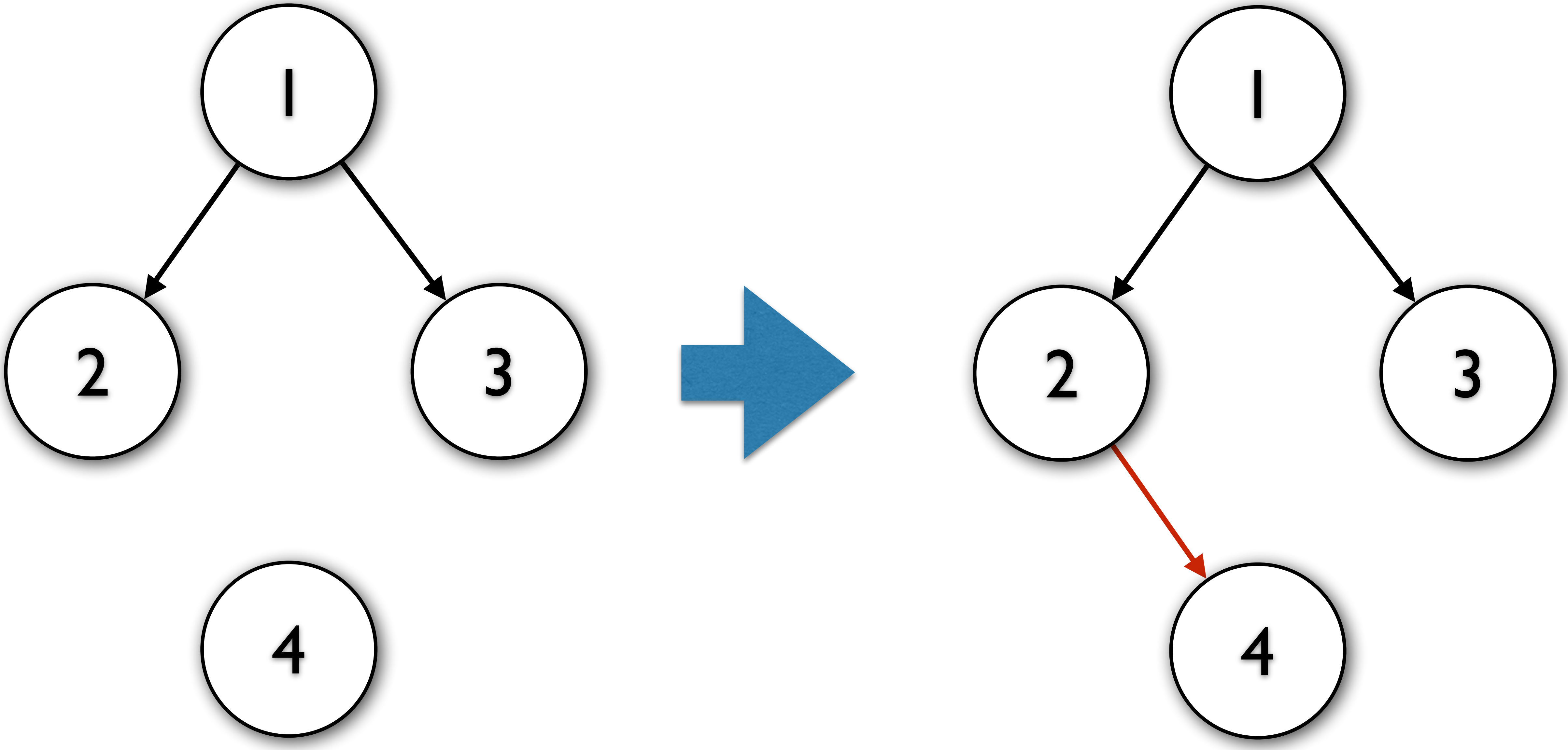
What is the definition of the edit distance between two CFGs?

- Add a zero-degree node



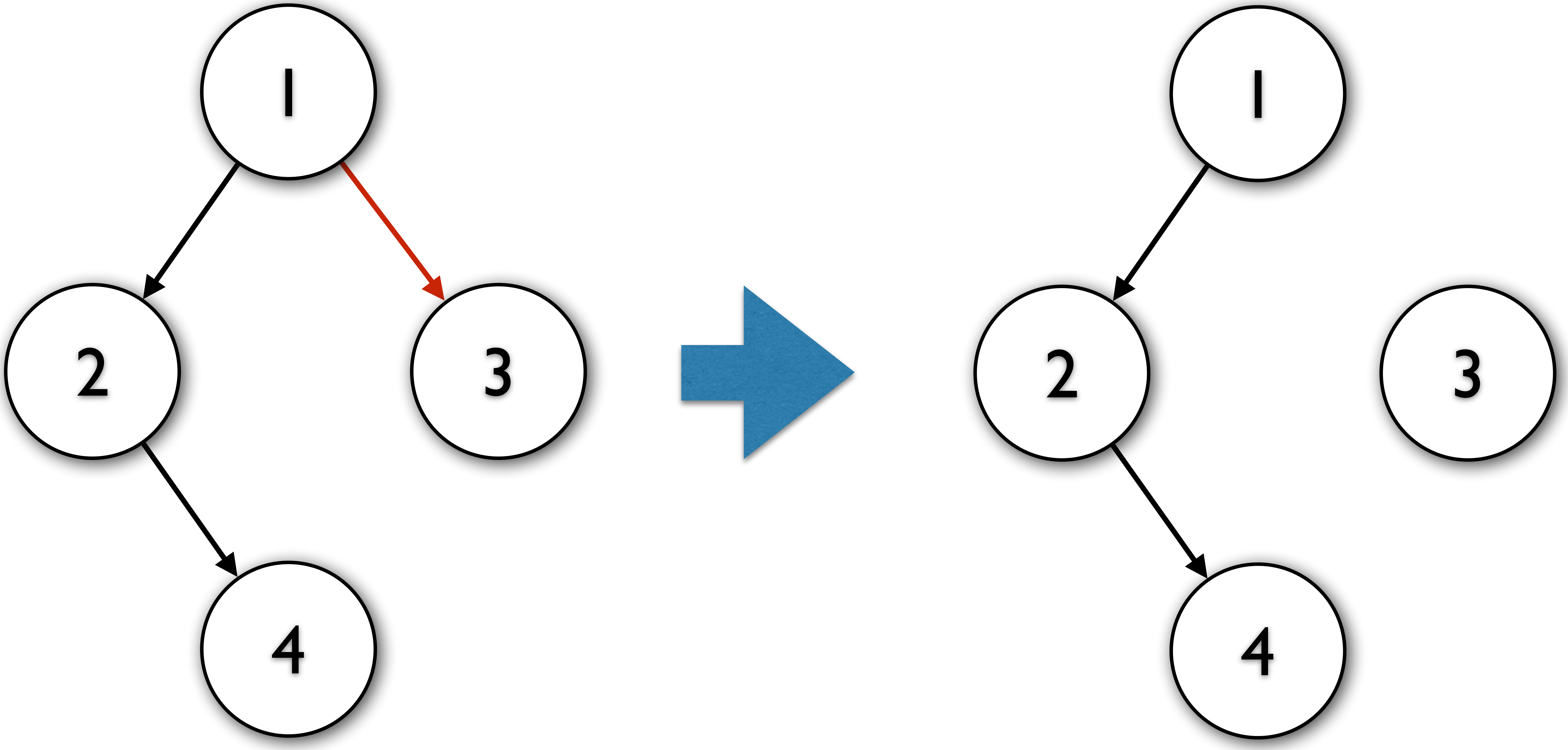
What is the definition of the edit distance between two CFGs?

- Add an edge between two existing nodes



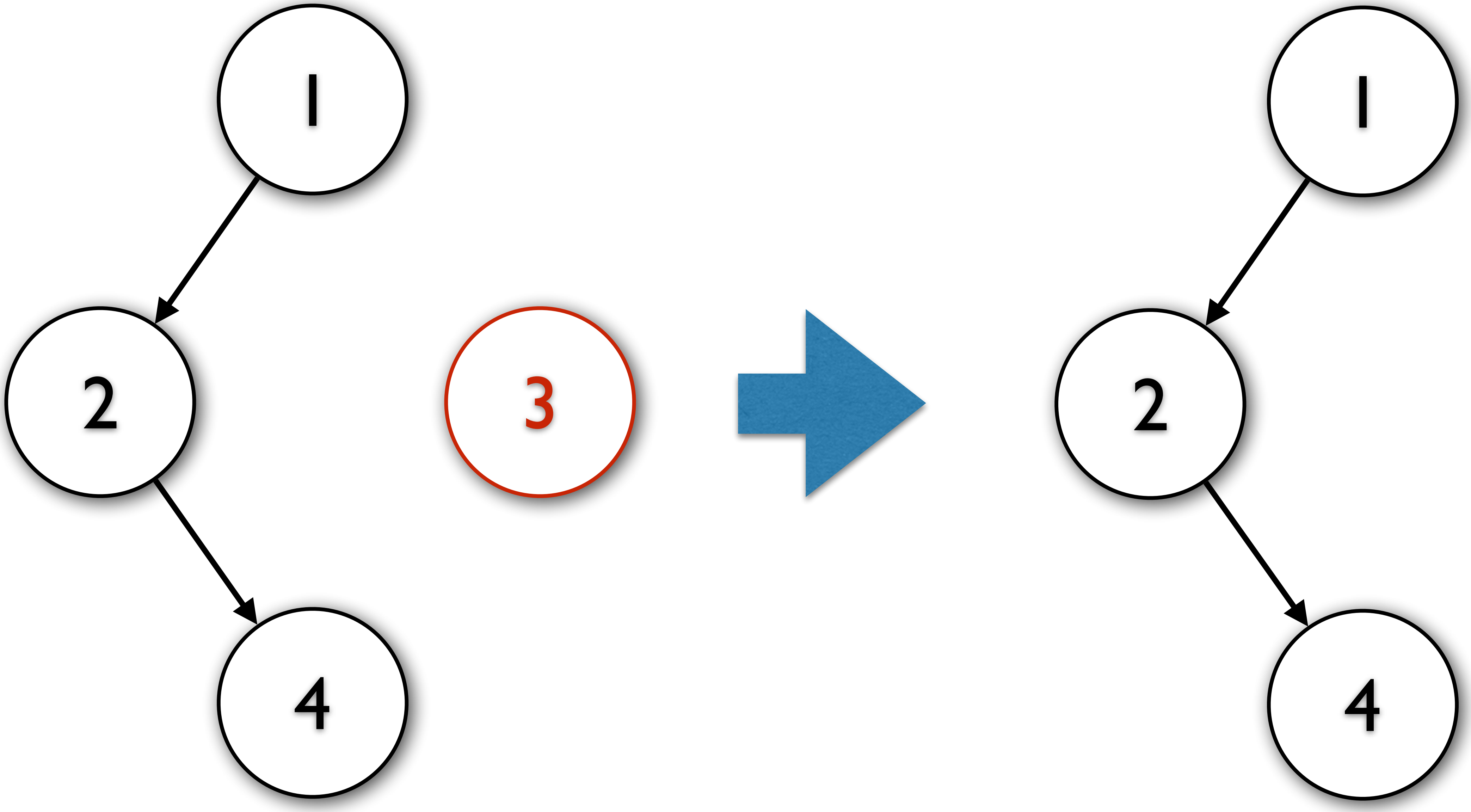
What is the definition of the edit distance between two CFGs?

- Delete an edge between two existing nodes

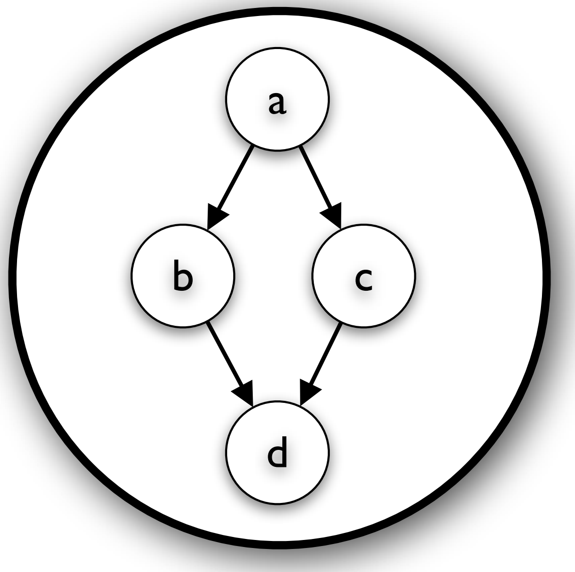


What is the definition of the edit distance between two CFGs?

- Delete a zero-degree node

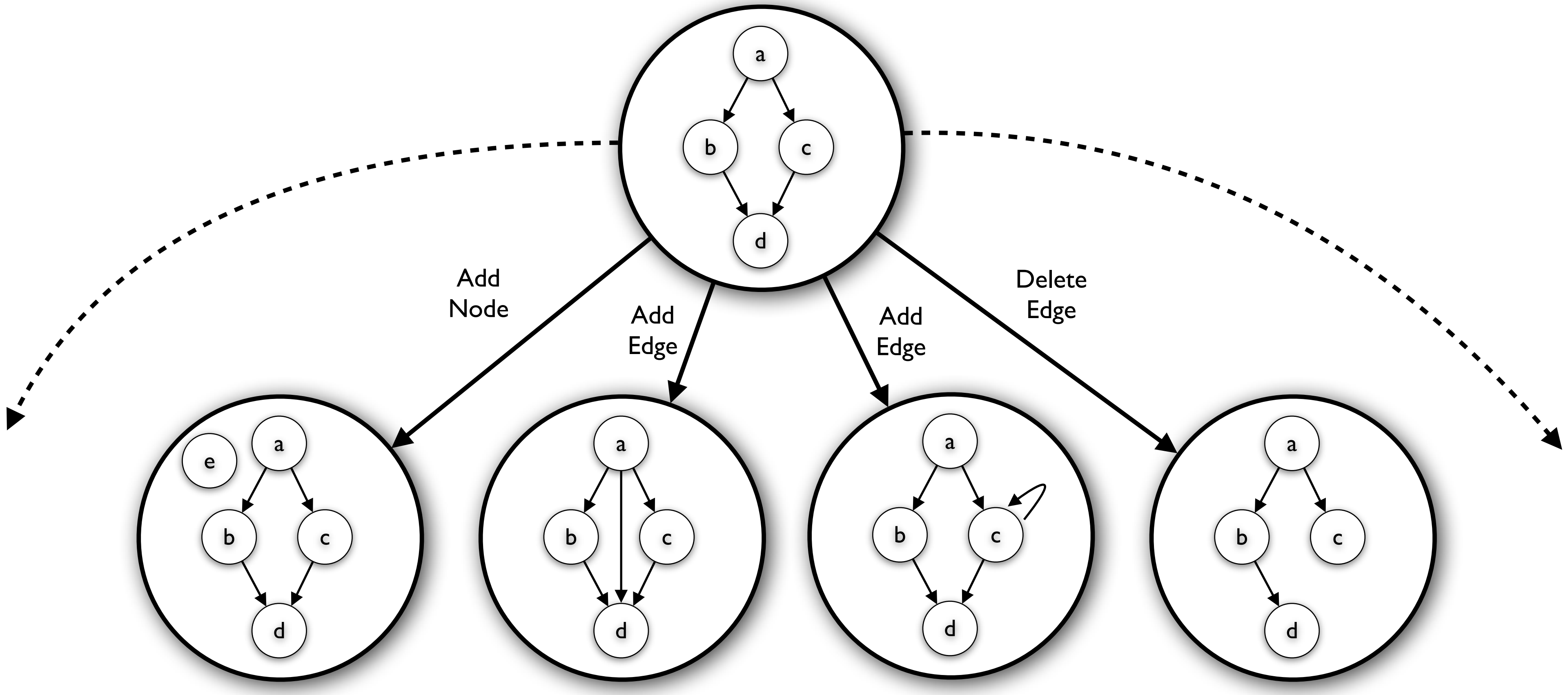


How to generate those CFGs such that they have increasing edit distances with the seed CFG G0?



G_0

How to generate those CFGs such that they have increasing edit distances with the seed CFG G0?

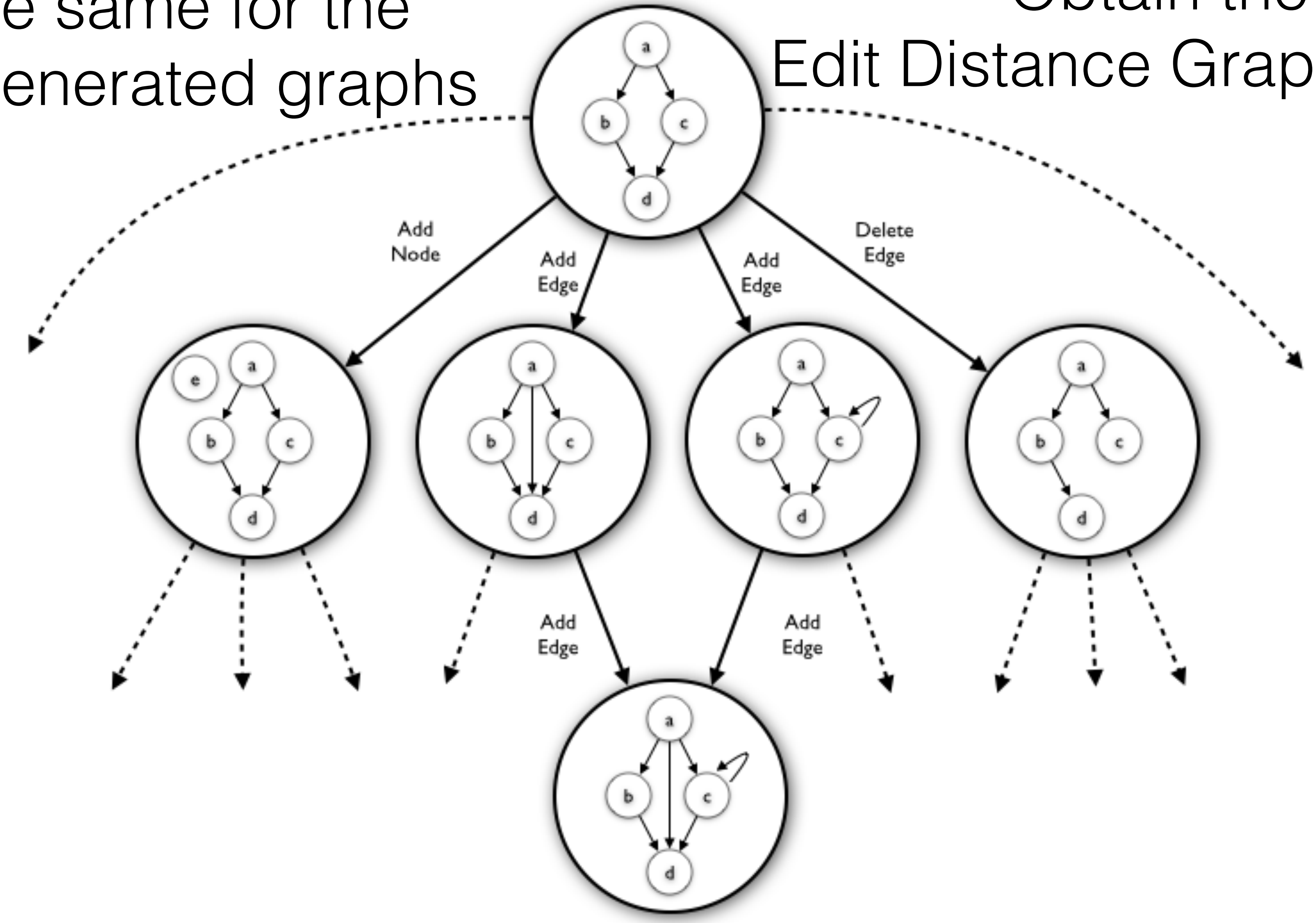


For every possible edit operation that can be applied to G_0, apply that and generate a new graph

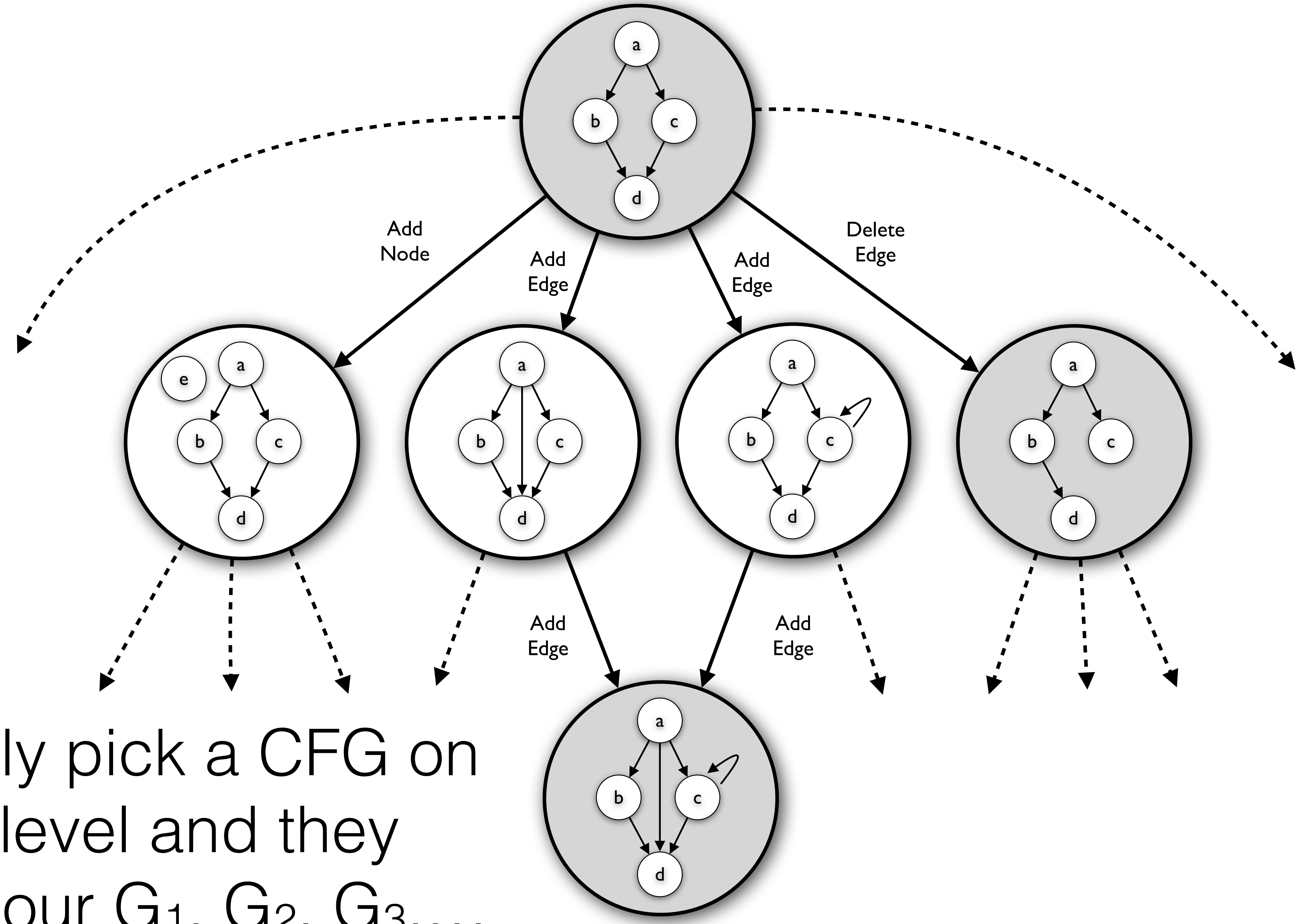
How to generate those CFGs such that they have increasing edit distances with the seed CFG G0?

Do the same for the newly generated graphs

Obtain the Edit Distance Graph (EDG)



How to generate those CFGs such that they have increasing edit distances with the seed CFG G0?

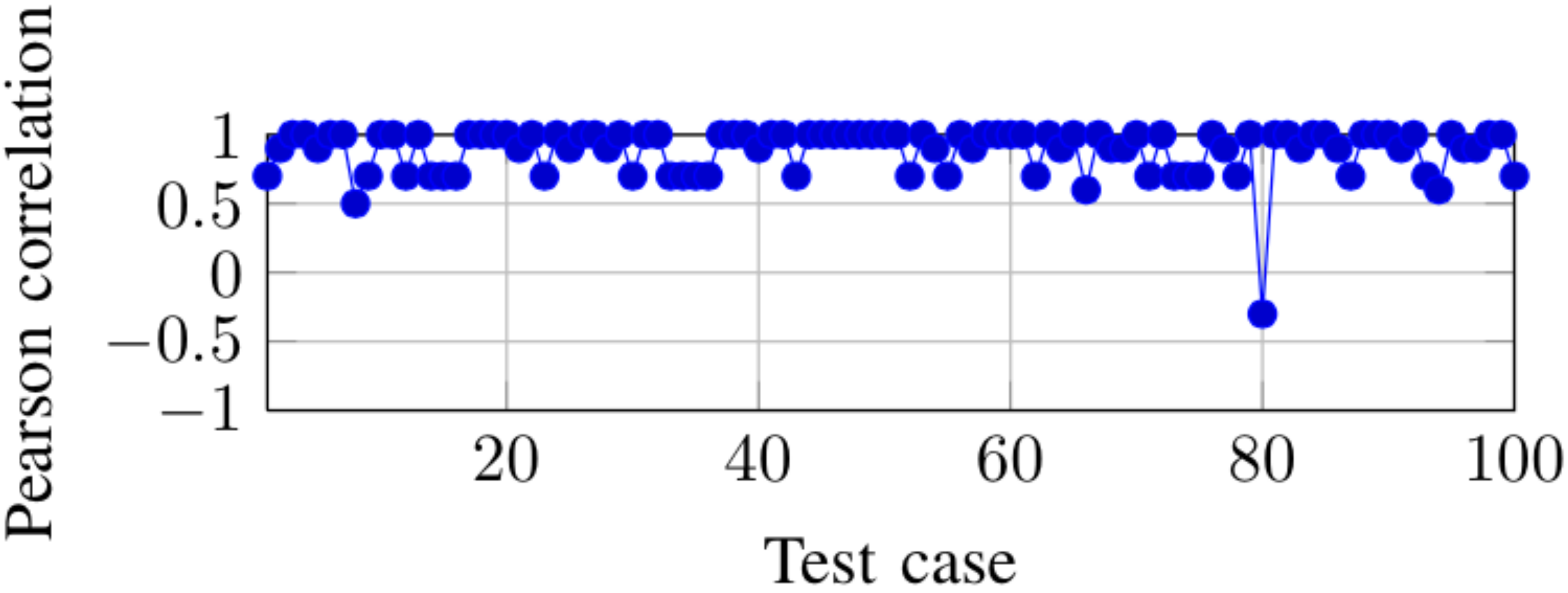


Randomly pick a CFG on each level and they become our G_1, G_2, G_3,…

Implementation

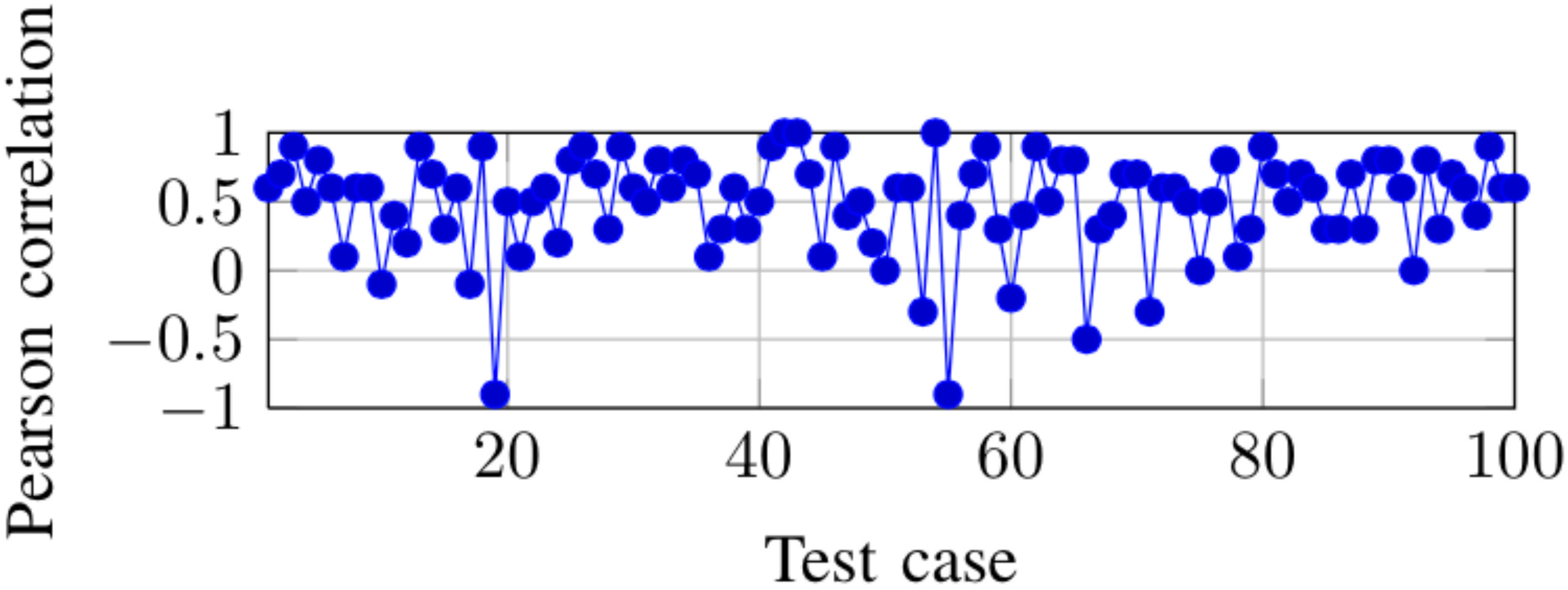
- Re-coded four CFG similarity algorithms in Python
- Implemented the evaluation framework
- Generated an EDG with five levels
- Picked 100 test cases (each test case comprises five CFGs)

Evaluation results



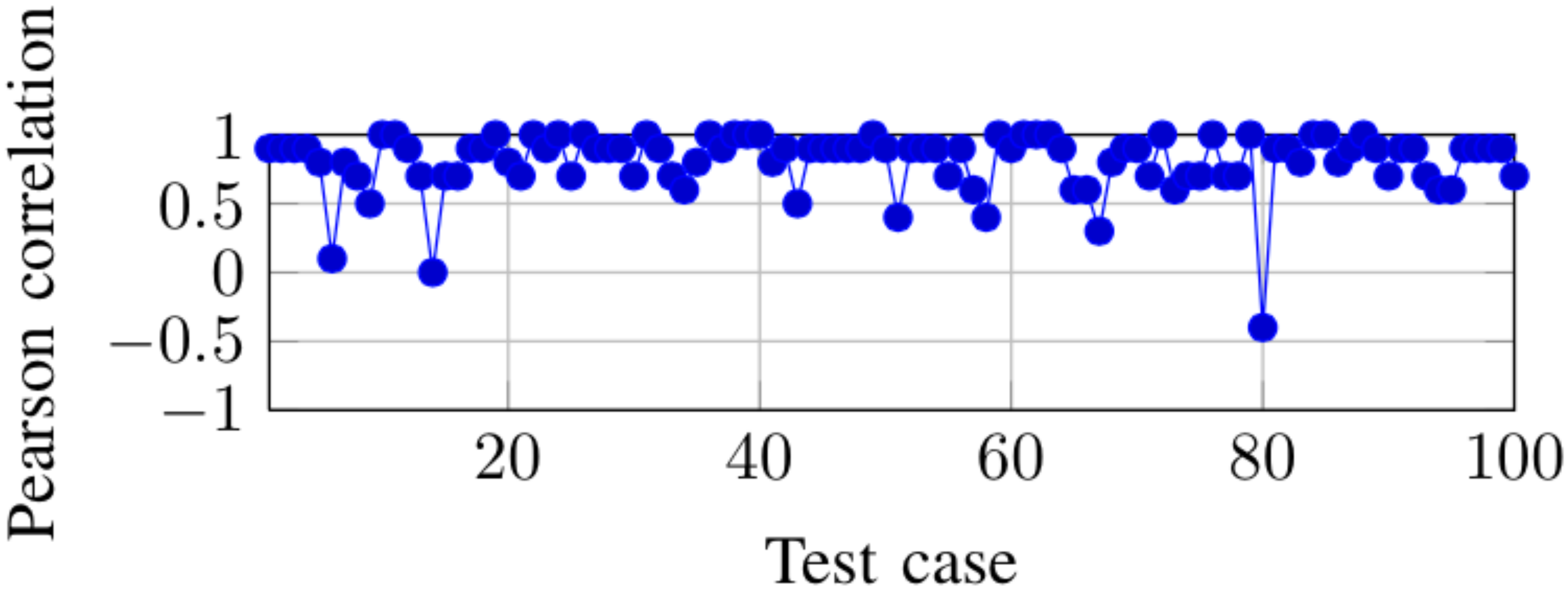
(a) $\mathcal{A}_{\text{Hu}}$

Evaluation results



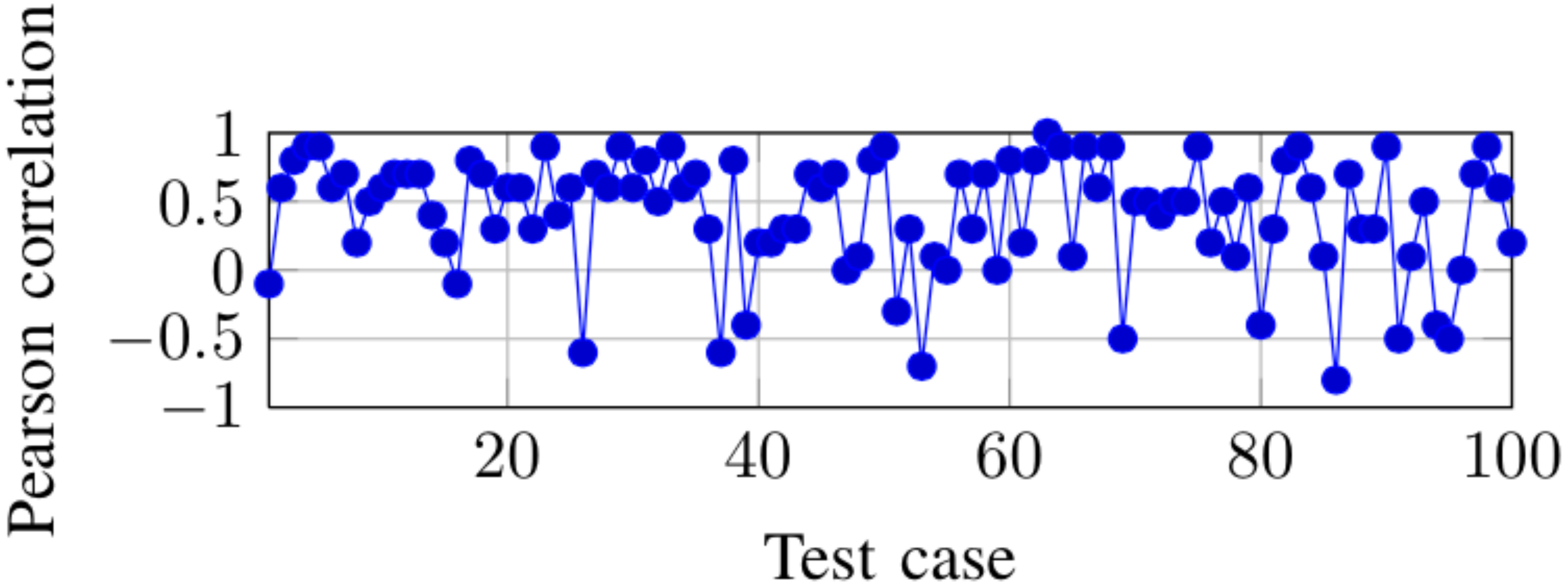
(b) $\mathcal{A}_{\text{Kruegel}}$

Evaluation results



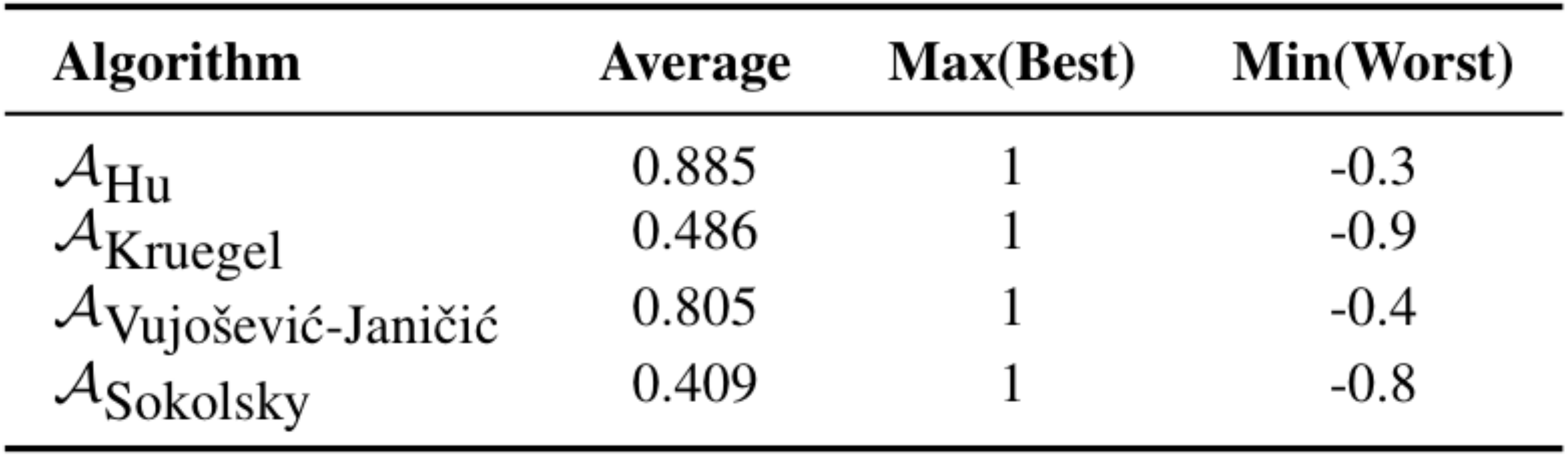
(c) $\mathcal{A}_{\text{Vujošević-Janičić}}$

Evaluation results



(d) $\mathcal{A}_{\text{Sokolsky}}$

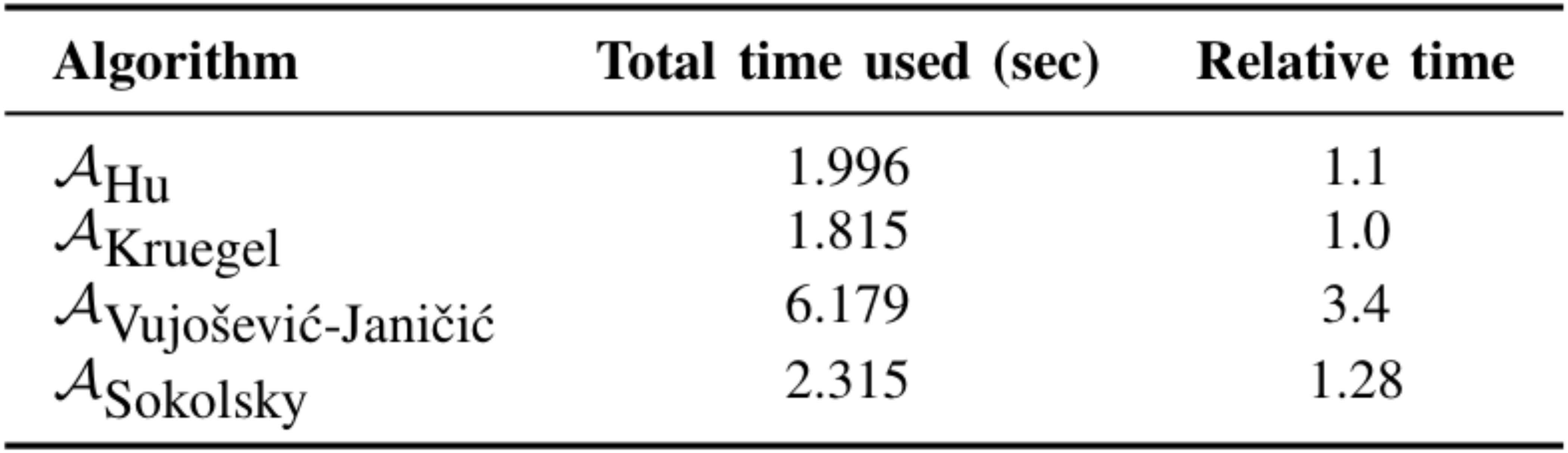
Evaluation results



Algorithm	Average	Max(Best)	Min(Worst)
$\mathcal{A}_{\text{Hu}}$	0.885	1	-0.3
$\mathcal{A}_{\text{Kruegel}}$	0.486	1	-0.9
$\mathcal{A}_{\text{Vujošević-Janičić}}$	0.805	1	-0.4
$\mathcal{A}_{\text{Sokolsky}}$	0.409	1	-0.8

“Goodness score” statistics of the four algorithms

Evaluation results



Algorithm	Total time used (sec)	Relative time
$\mathcal{A}_{\text{Hu}}$	1.996	1.1
$\mathcal{A}_{\text{Kruegel}}$	1.815	1.0
$\mathcal{A}_{\text{Vujošević-Janičić}}$	6.179	3.4
$\mathcal{A}_{\text{Sokolsky}}$	2.315	1.28

Time used by the four algorithms to finish 100 test cases

Related work

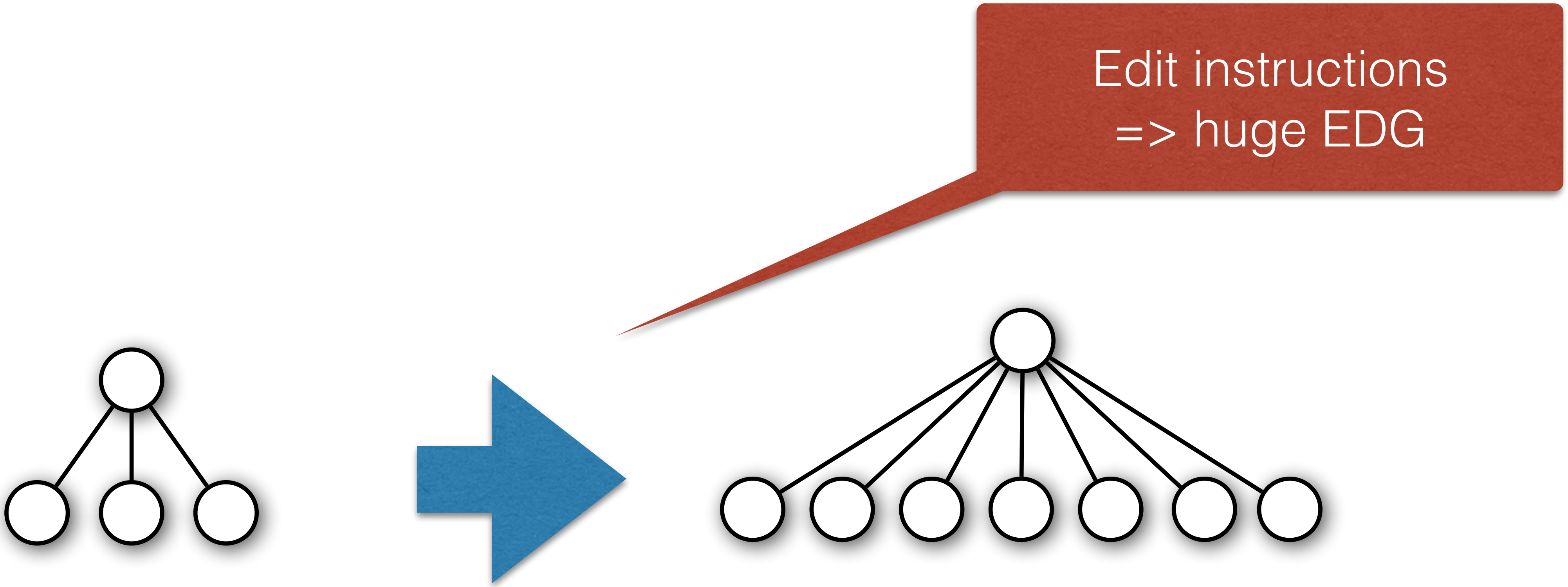
- An evaluation framework for **text** plagiarism detection
    - Generate artificial plagiarism cases
    - Shuffling, removing, inserting, or replacing words or short phrases at random

Related work

- An evaluation framework for code clone detection tools
    - Inject mutated code fragments into the code base

Future work

- Generate CFGs with instructions in the nodes



Edit instructions
=> huge EDG

Try our framework

- http://cfgsim.cs.arizona.edu/
- Evaluate existing algorithms
- Compare your own algorithm with the others
- Fine tune your algorithm

Summary

- A methodology to evaluate CFG similarity algorithms
- Publicly available evaluation framework
- Serves as a benchmark for CFG similarity algorithms users / researchers

Thank you!