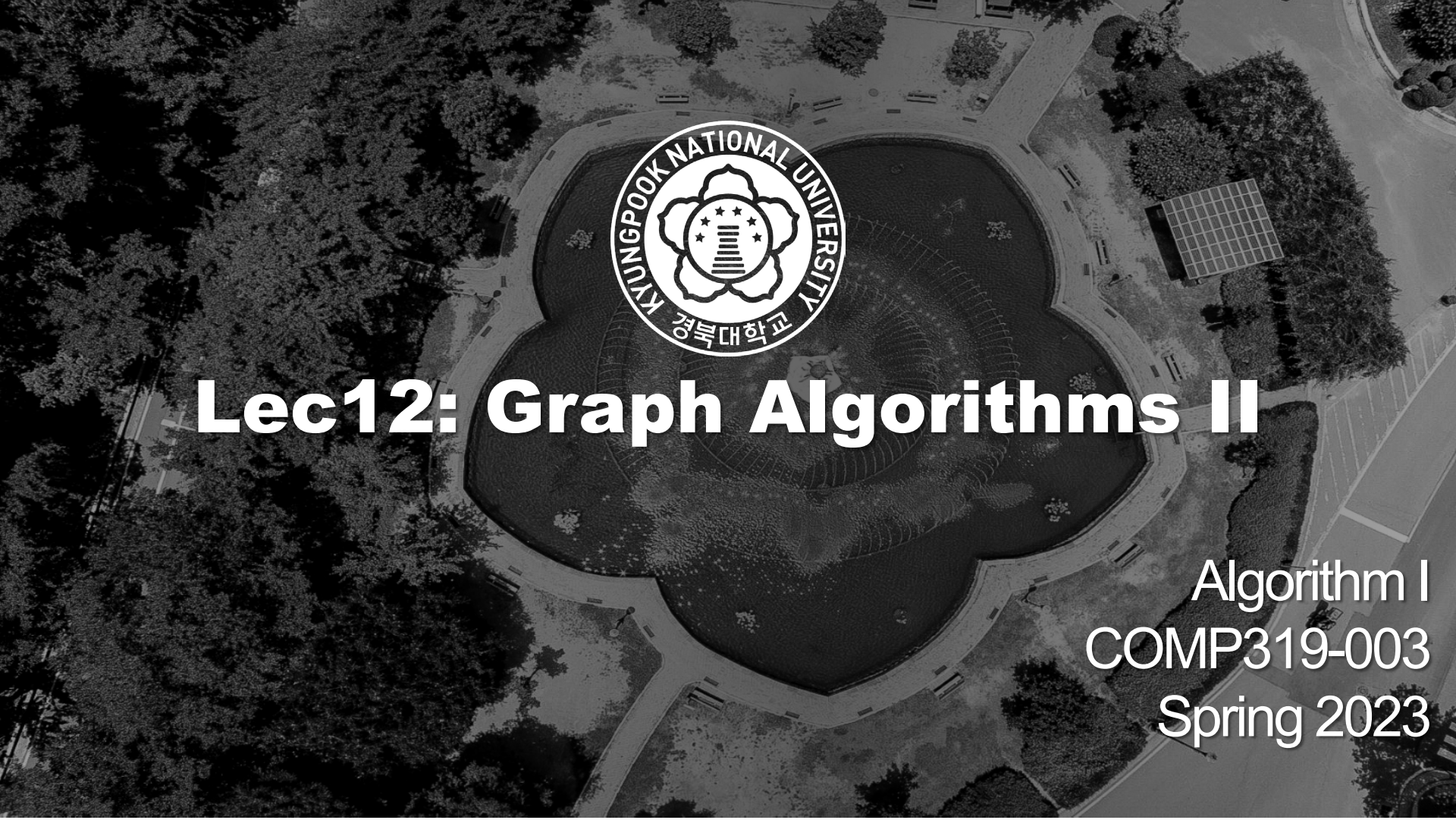


Lec12: Graph Algorithms II

Algorithm I
COMP319-003
Spring 2023

Instructor: Jiyeon Lee
School of Computer Science and Engineering
Kyungpook National University **(KNU)**

Last time

- Graph representation
- depth-first search (DFS)
- Plus, applications!
  - Topological sorting
  - In-order traversal of BSTs

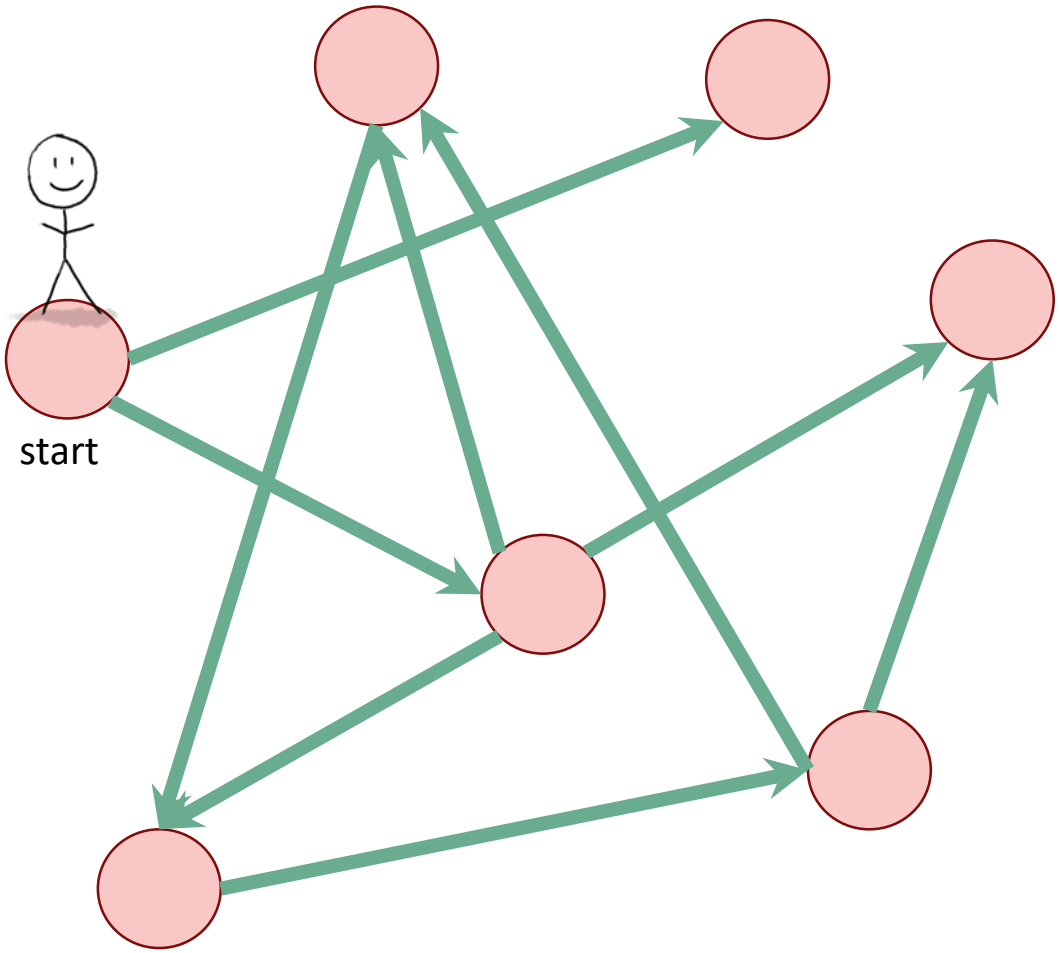
- The key was paying attention to the structure of the tree that the search algorithm implicitly builds.

Outline

1. Representations
2. DFS (Depth First Search)
   - Topological Ordering
   - **Strongly Connected Components**

- *Reading: CLRS 22.1 – 22.4*

Recall: DFS



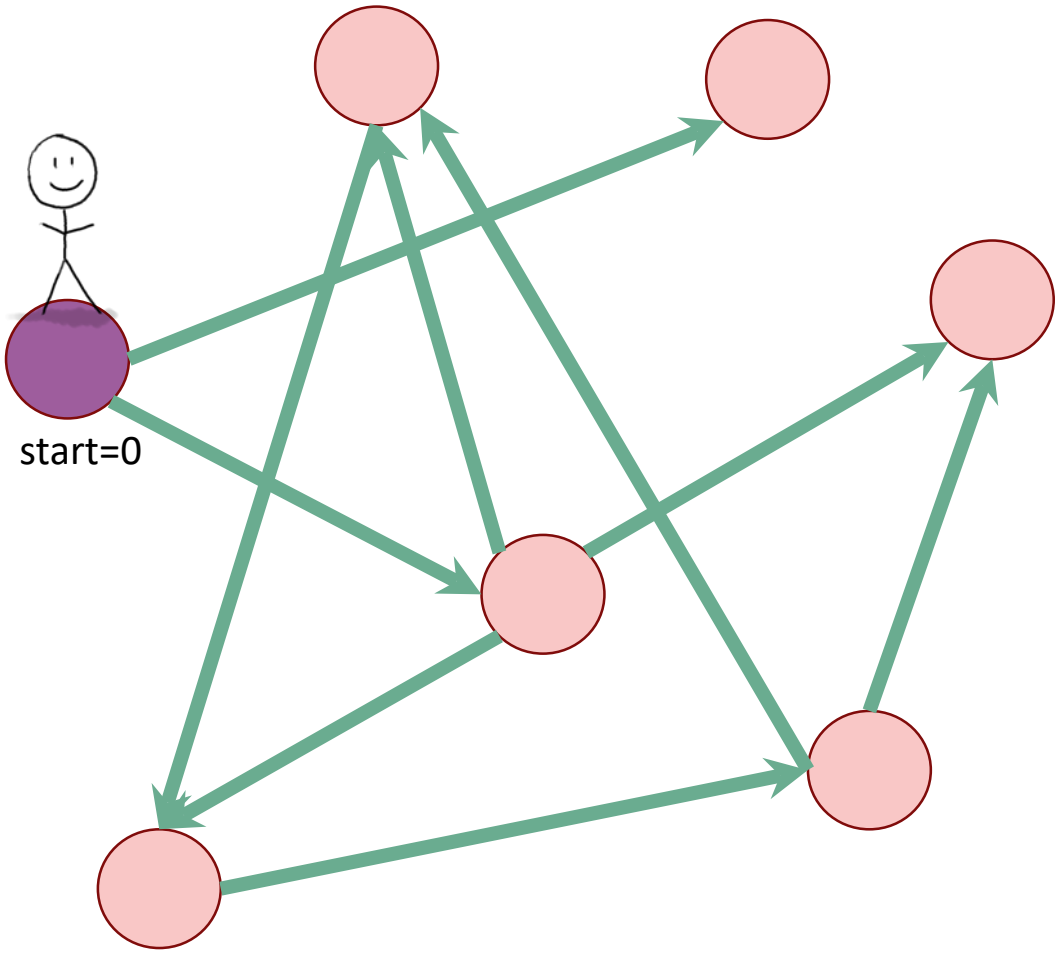
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

This is the same picture we had in the last lecture, except I've directed all the edges.

Recall: DFS



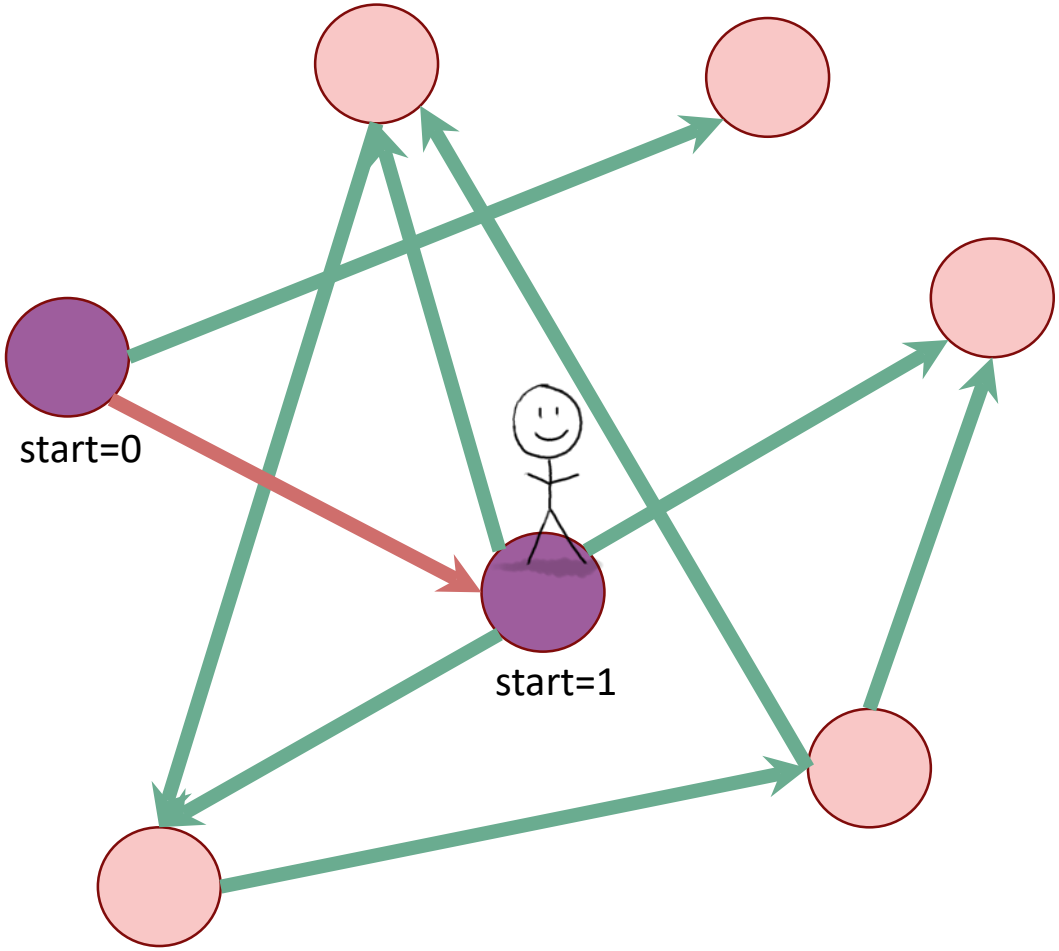
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



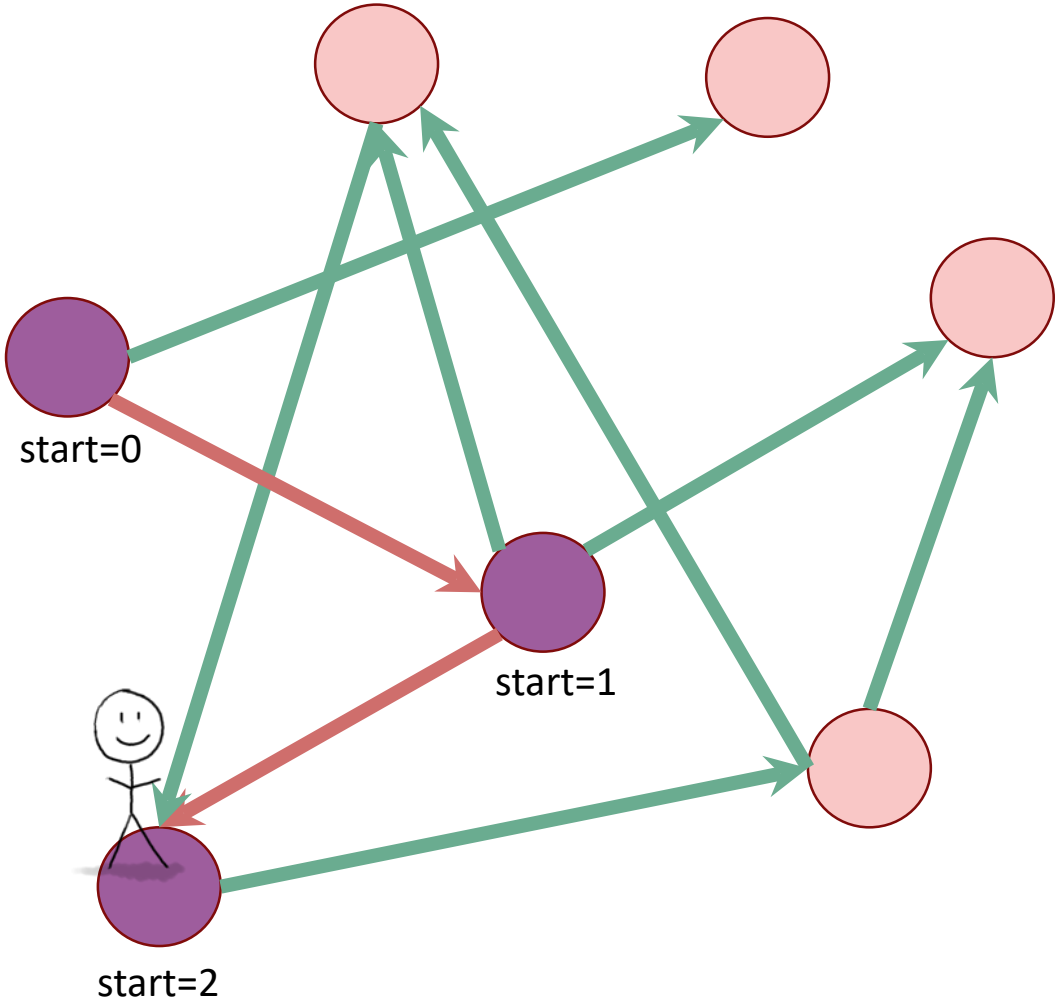
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



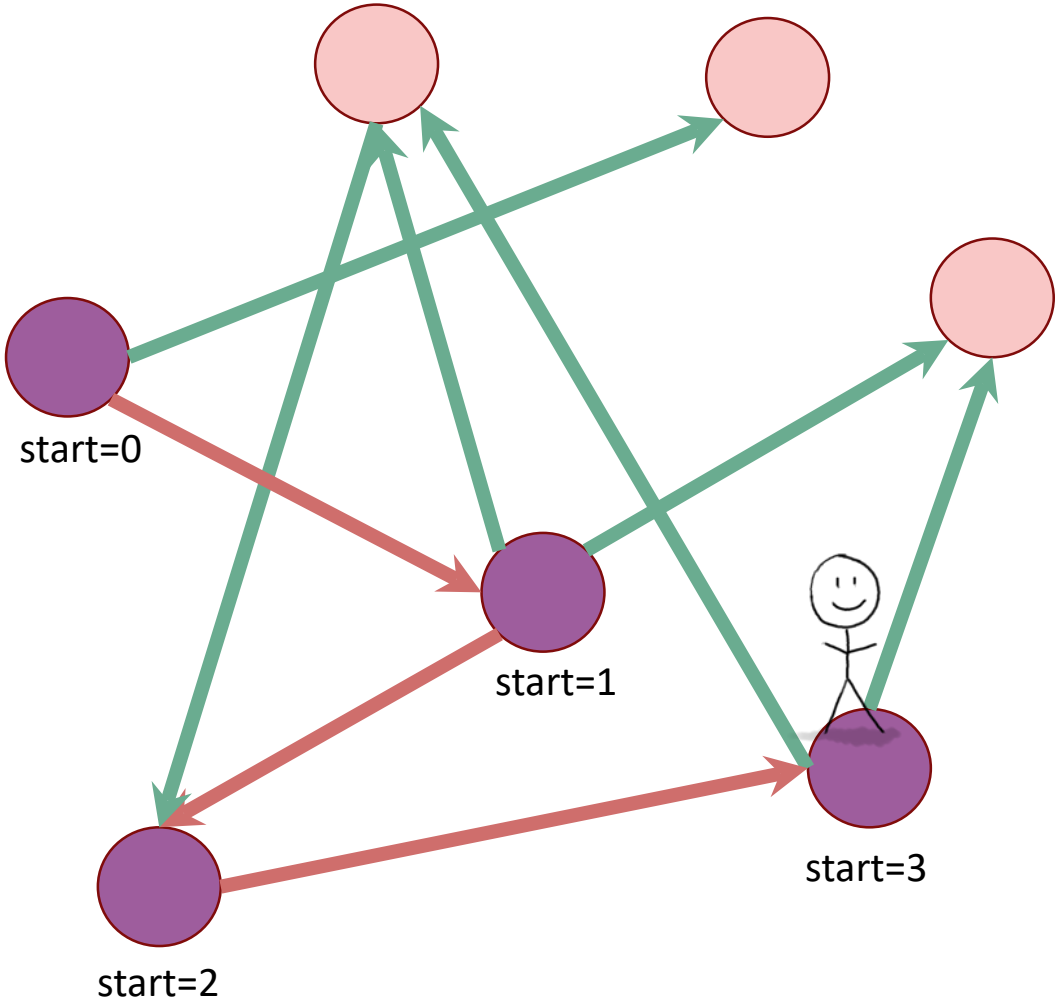
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



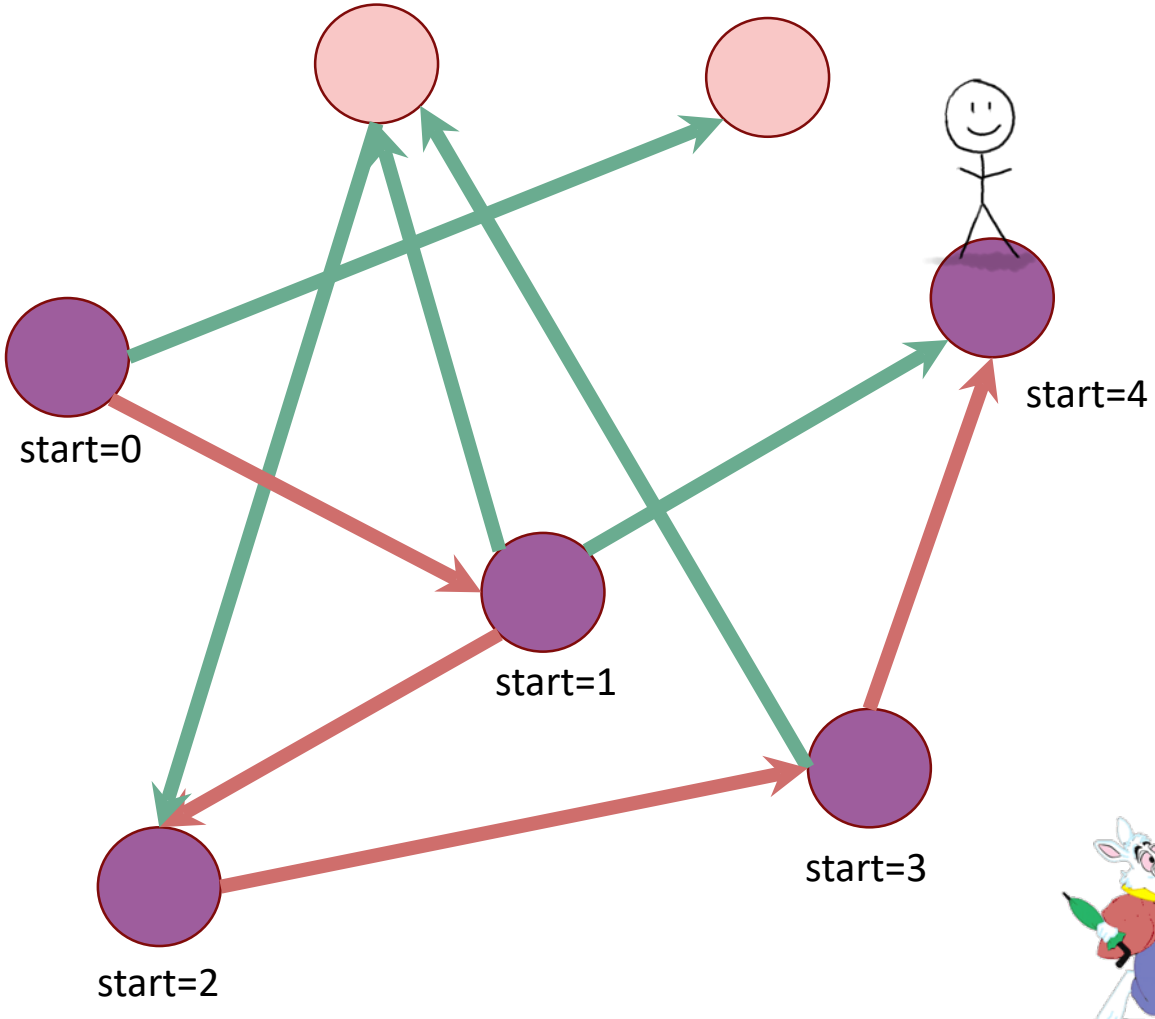
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



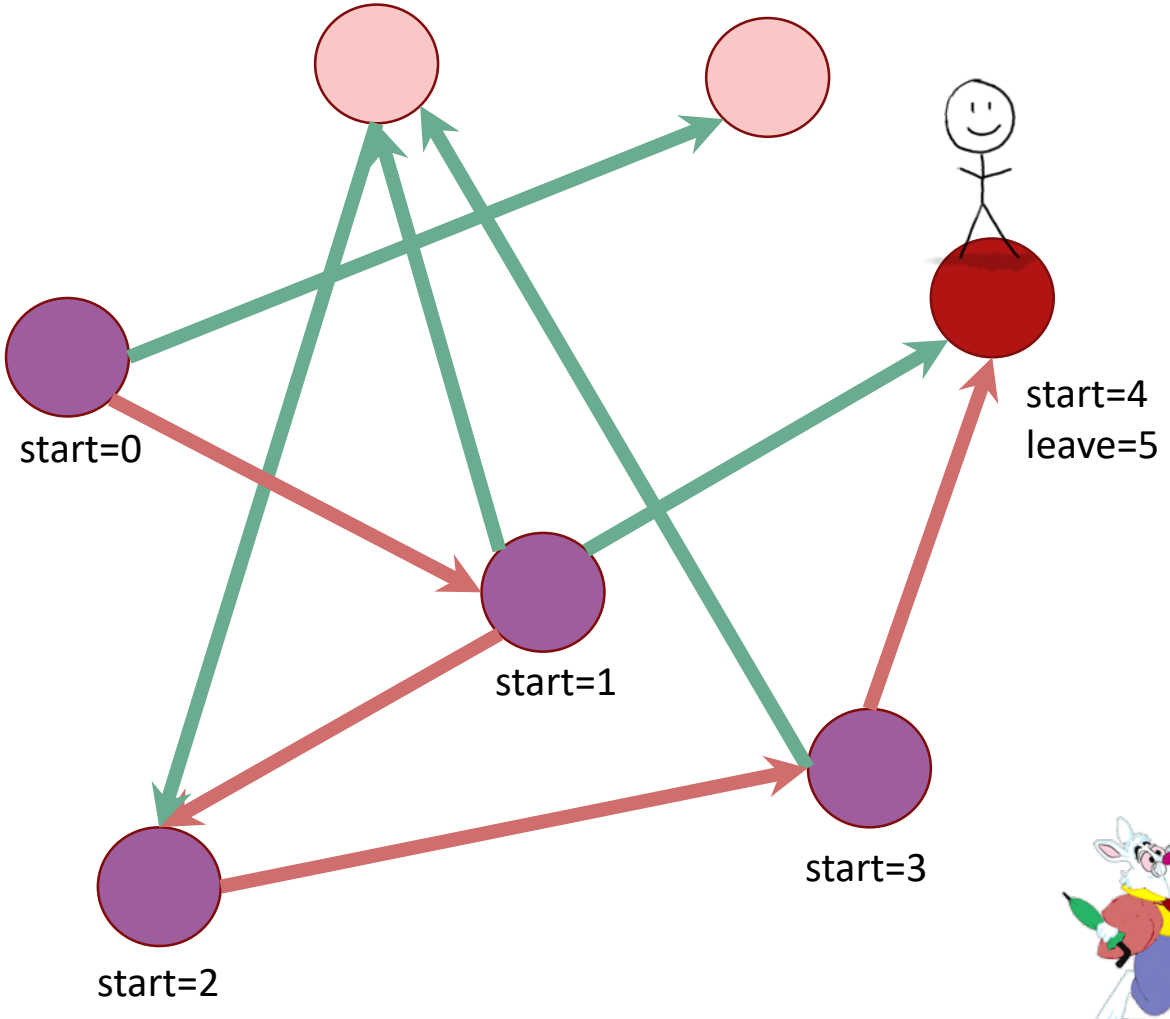
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



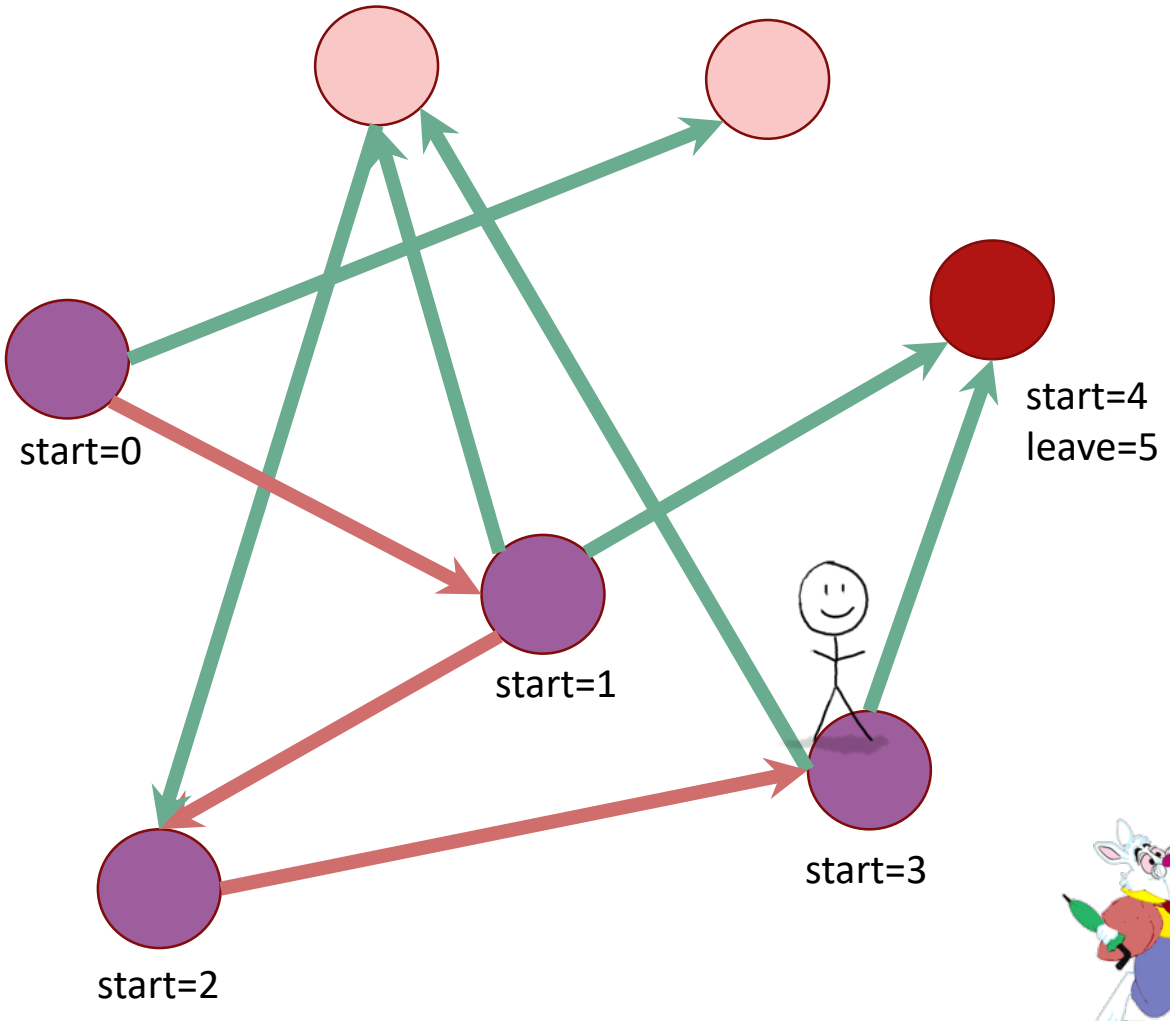
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



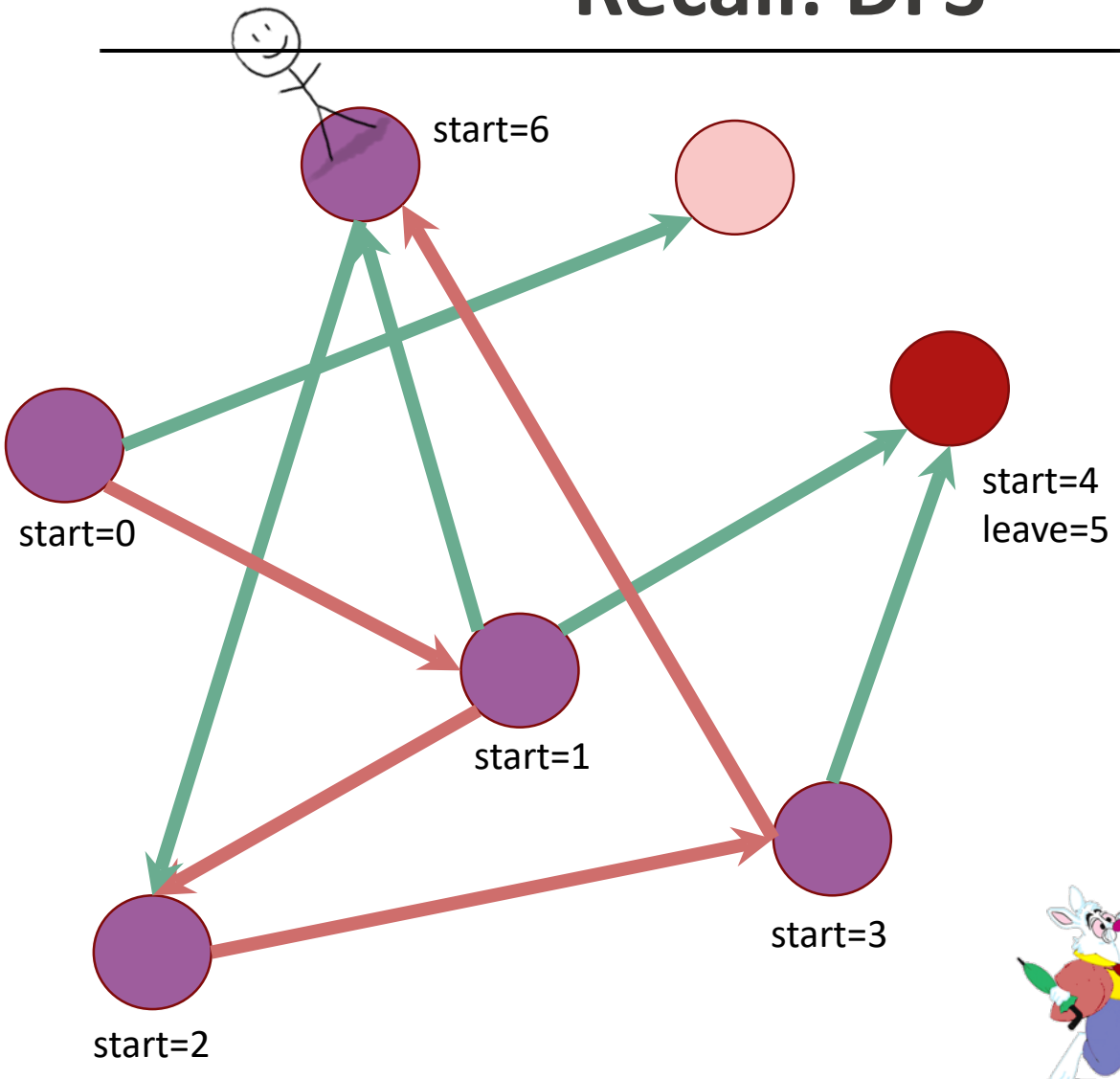
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



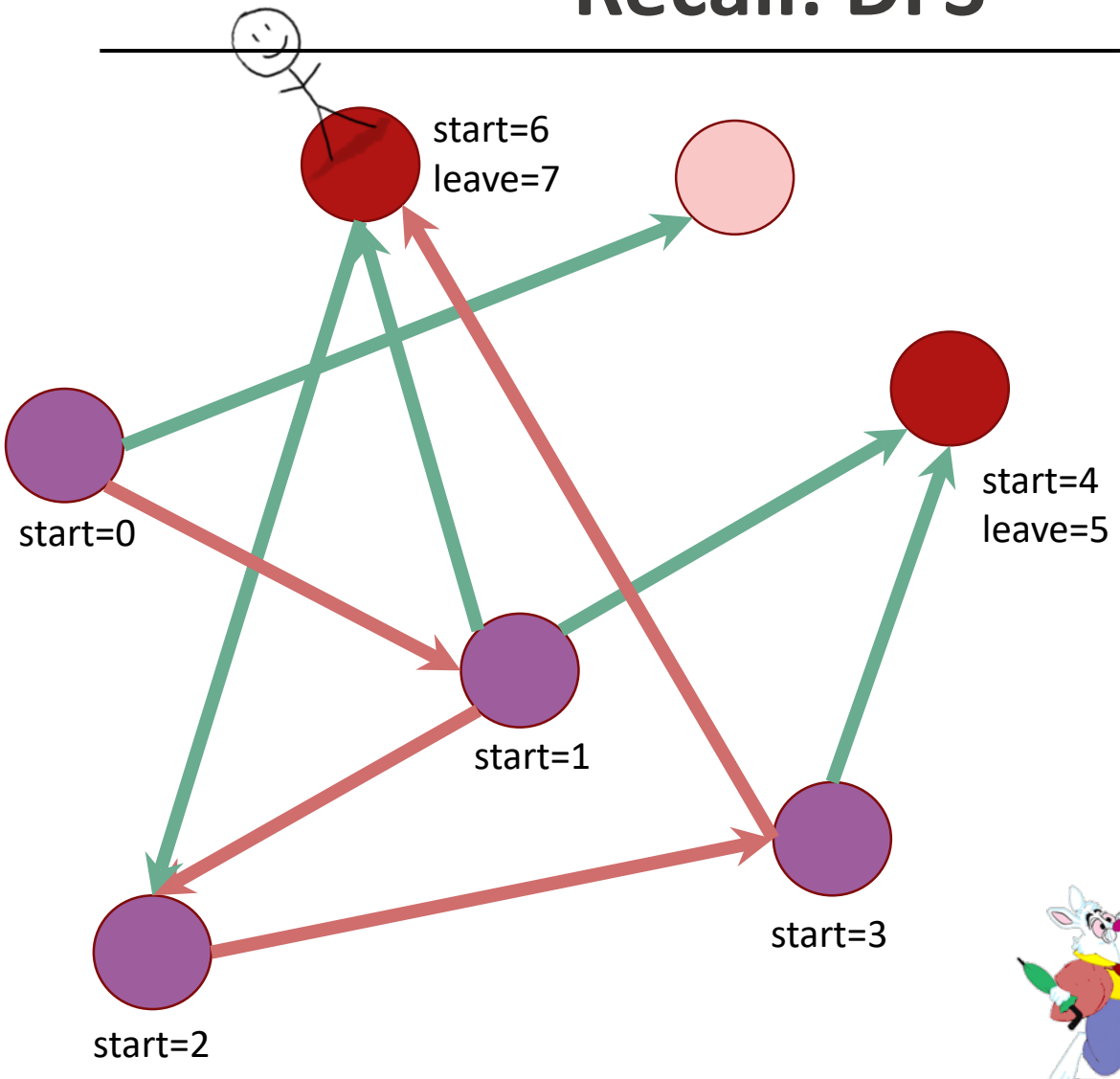
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



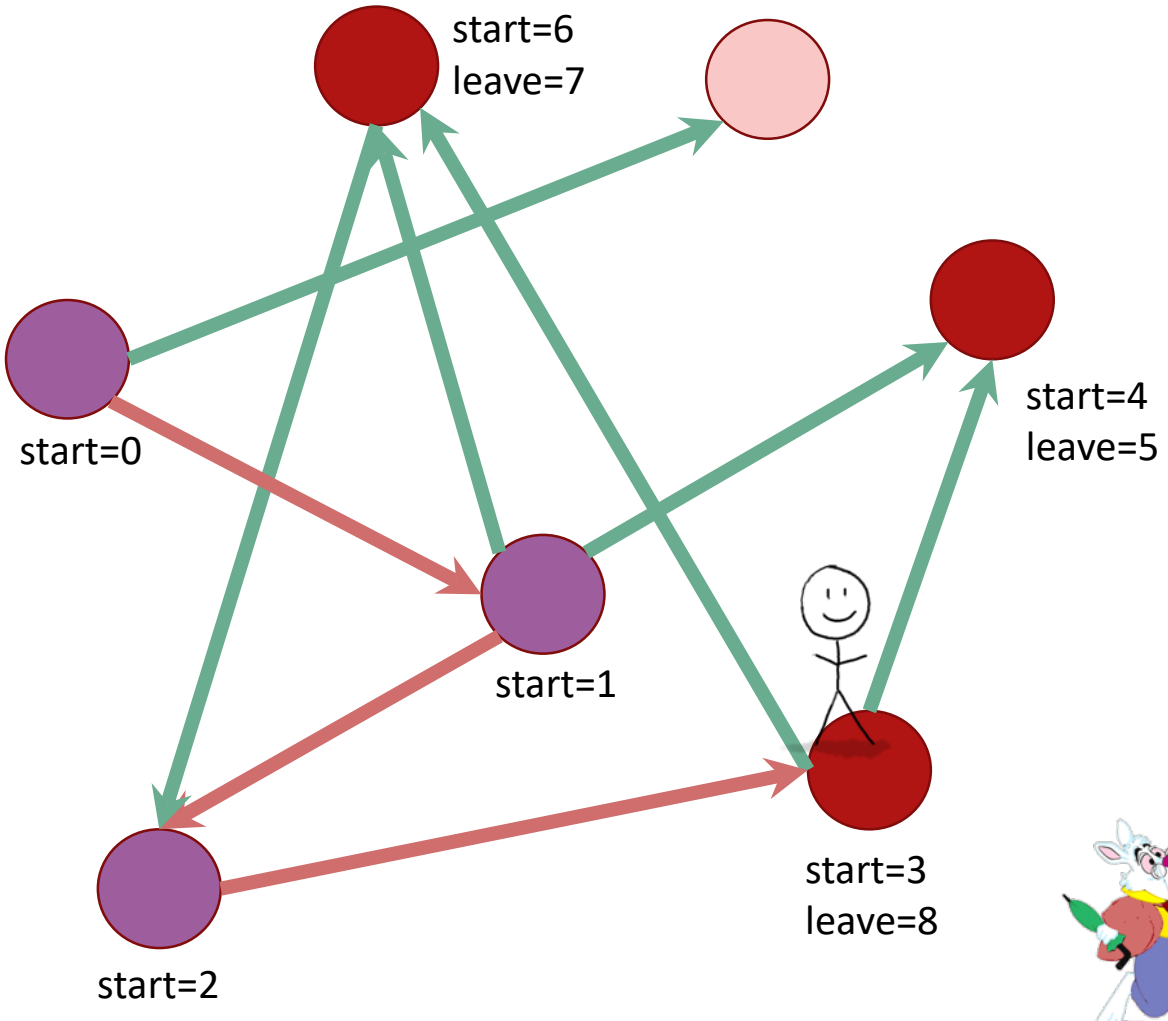
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



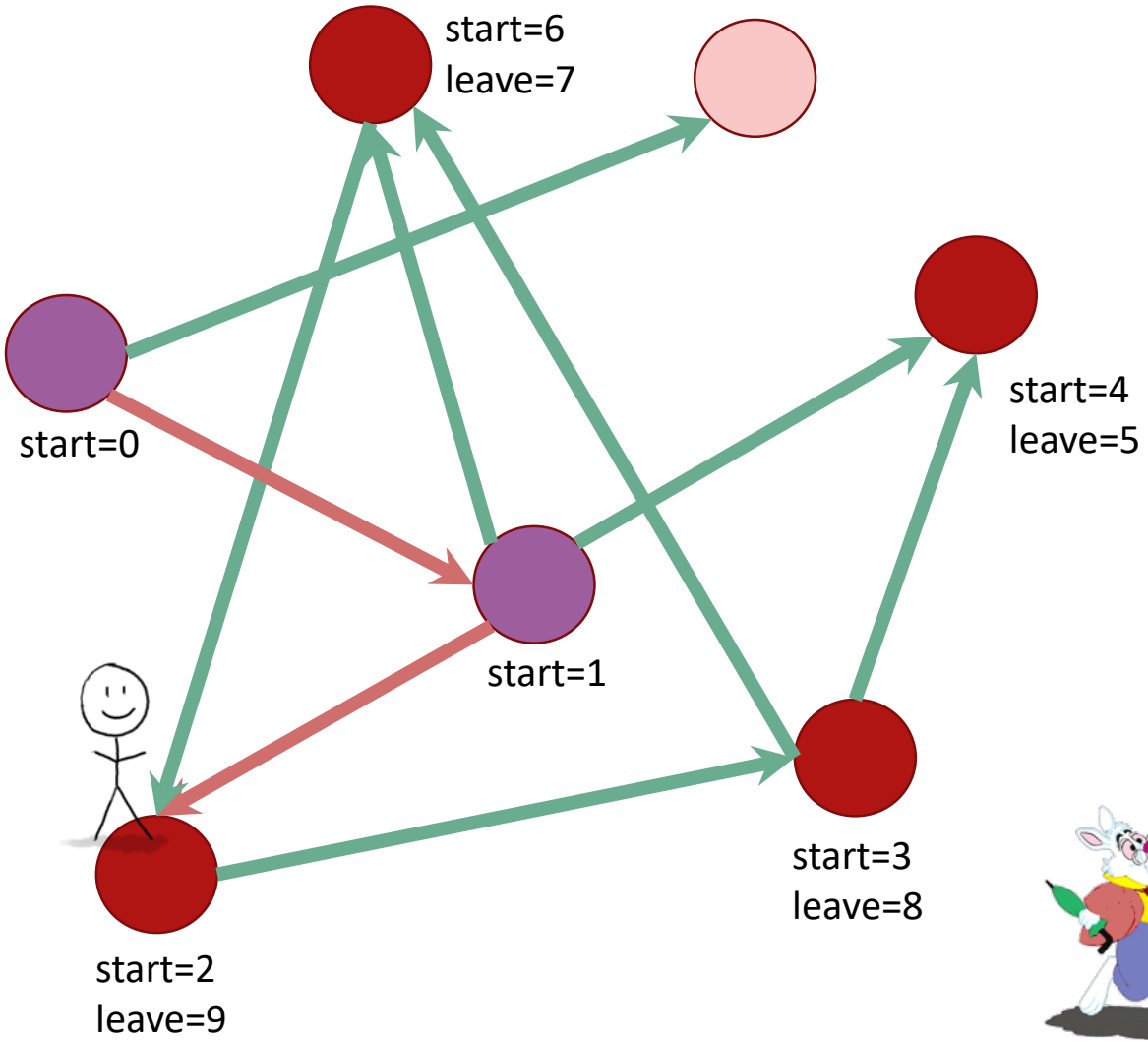
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



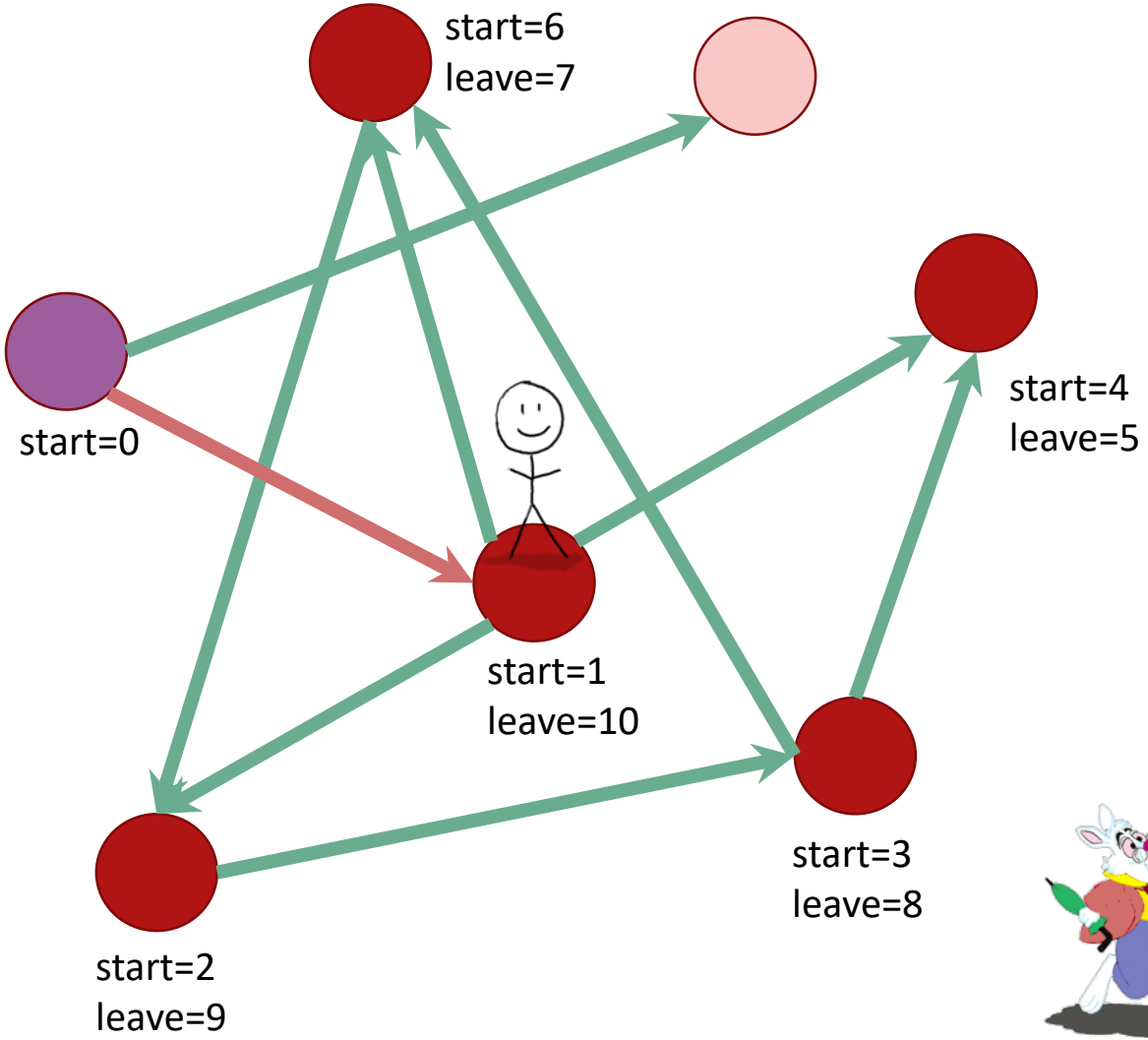
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



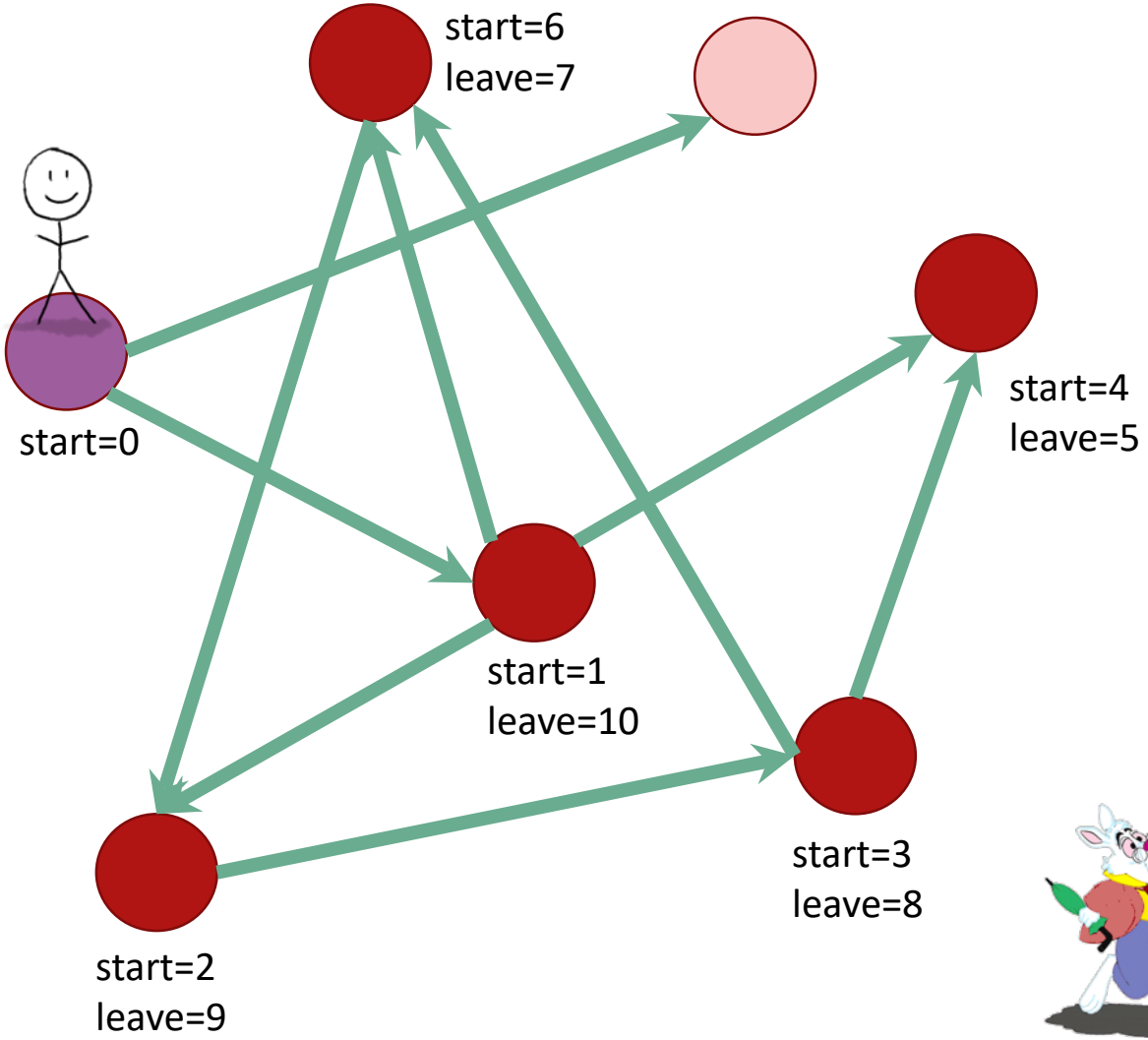
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



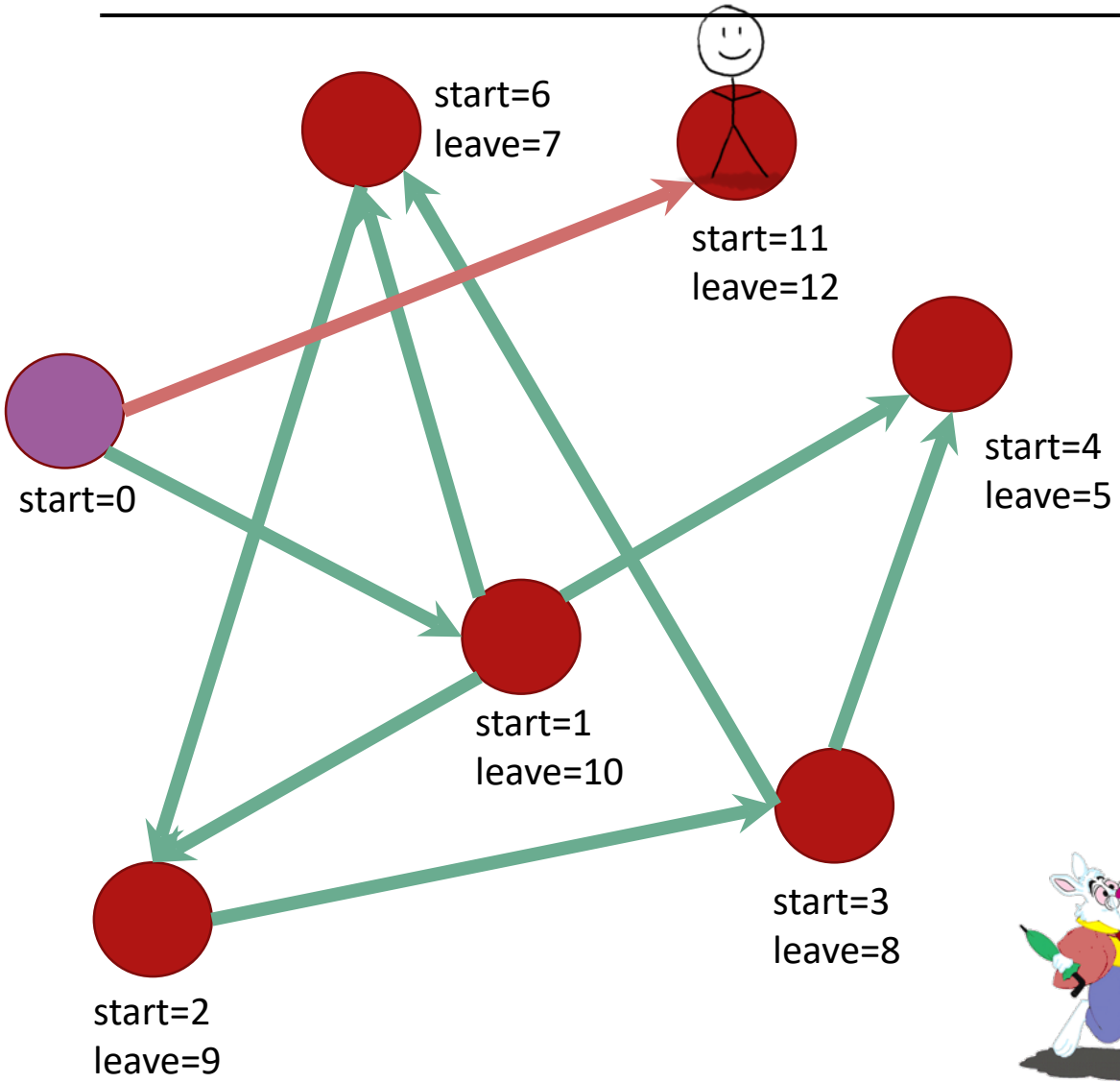
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS



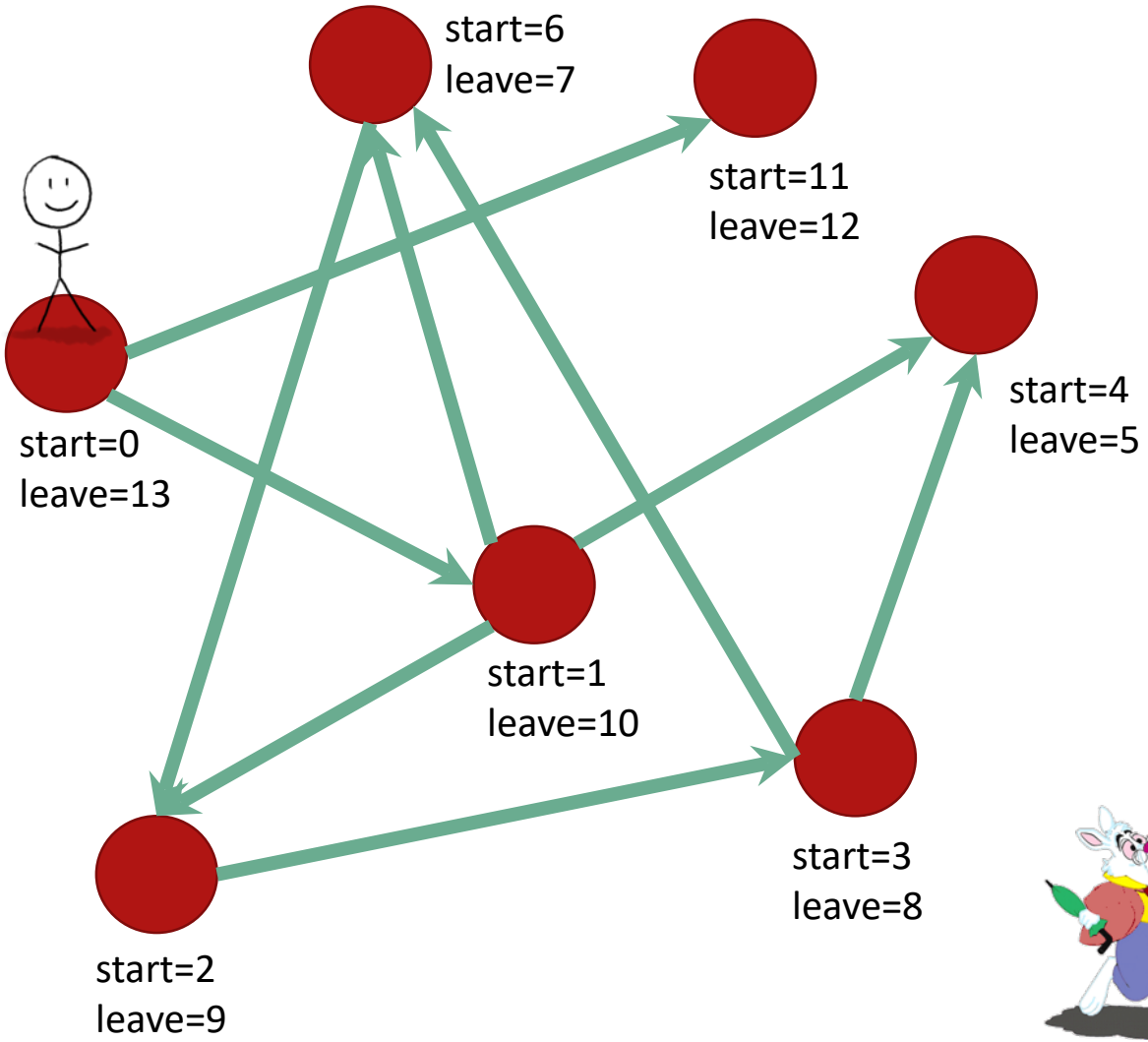
Not been there yet

Been there, haven't explored all the paths out.

Been there, have explored all the paths out.

Recall we also keep track of **start** and **finish** times for every vertex.

Recall: DFS

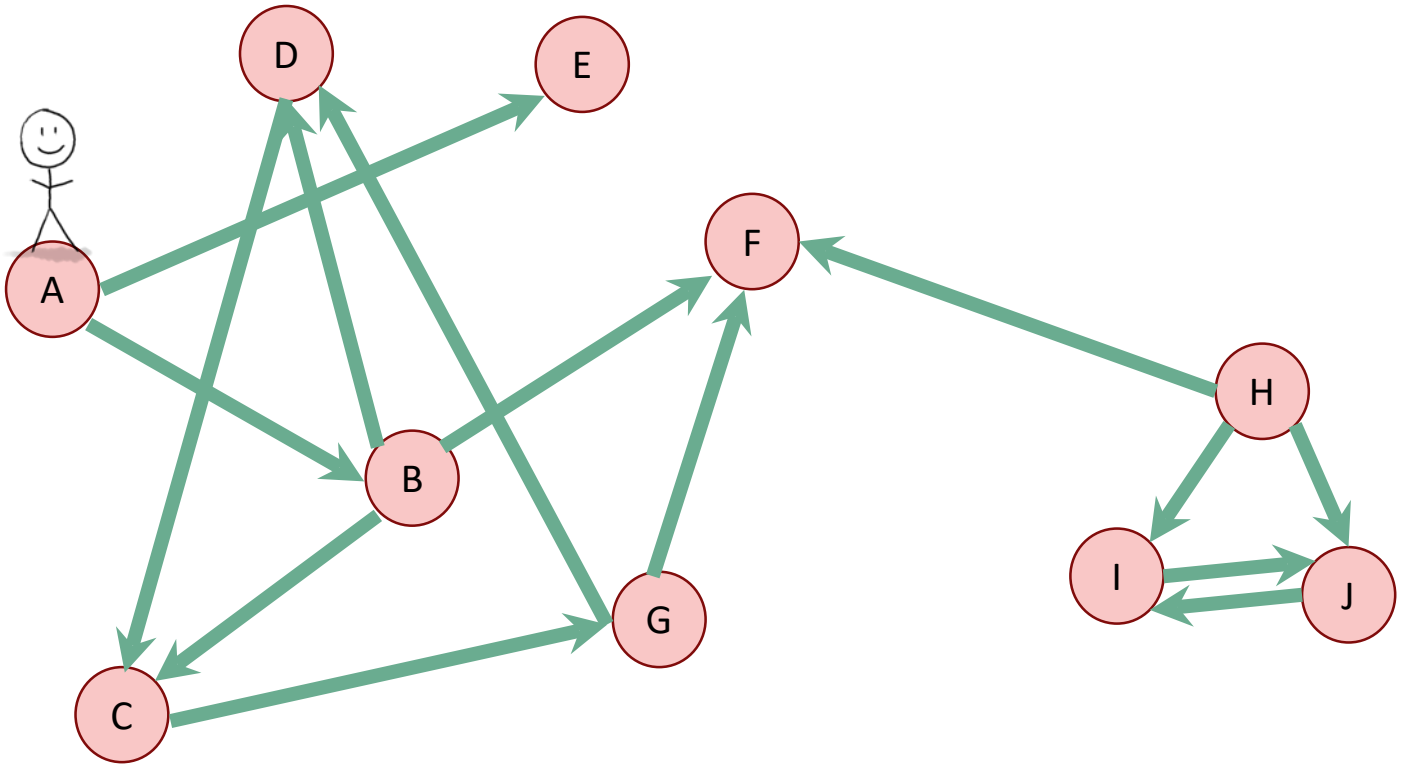


explored!

- Not been there yet
- Been there, haven't explored all the paths out.
- Been there, have explored all the paths out.

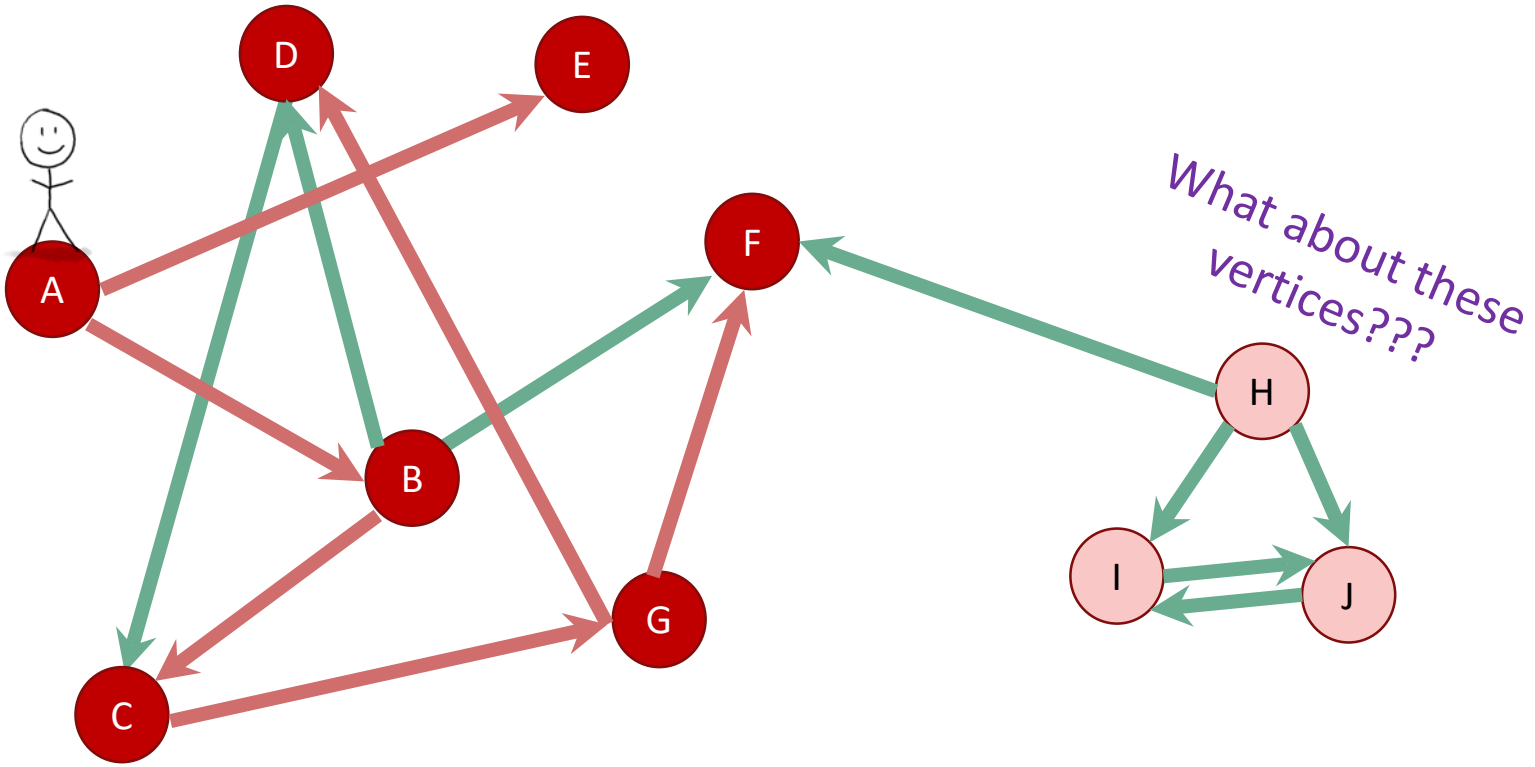
Recall we also keep track of **start** and **finish** times for every vertex.

When you can’t reach everything



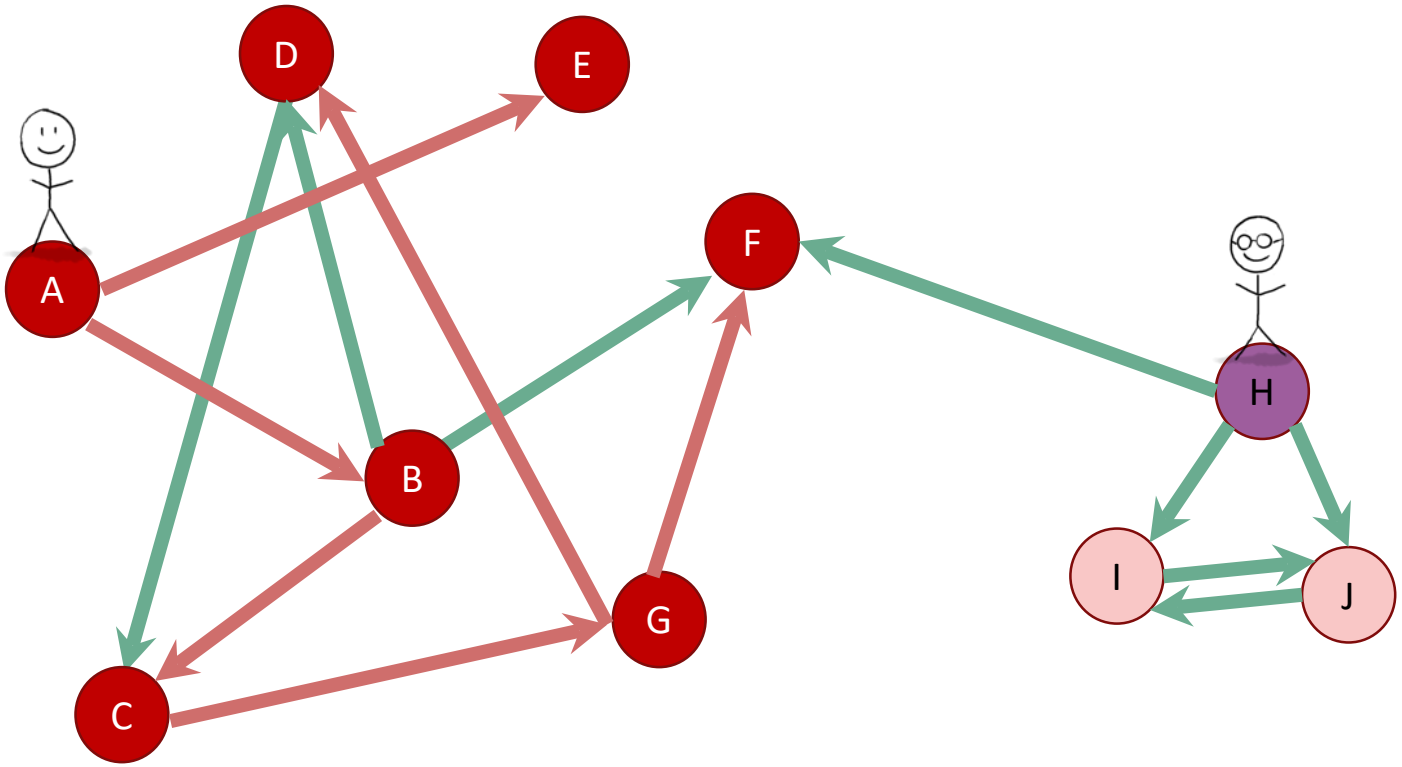
When you can't reach everything

- Run DFS repeatedly to get a depth-first forest



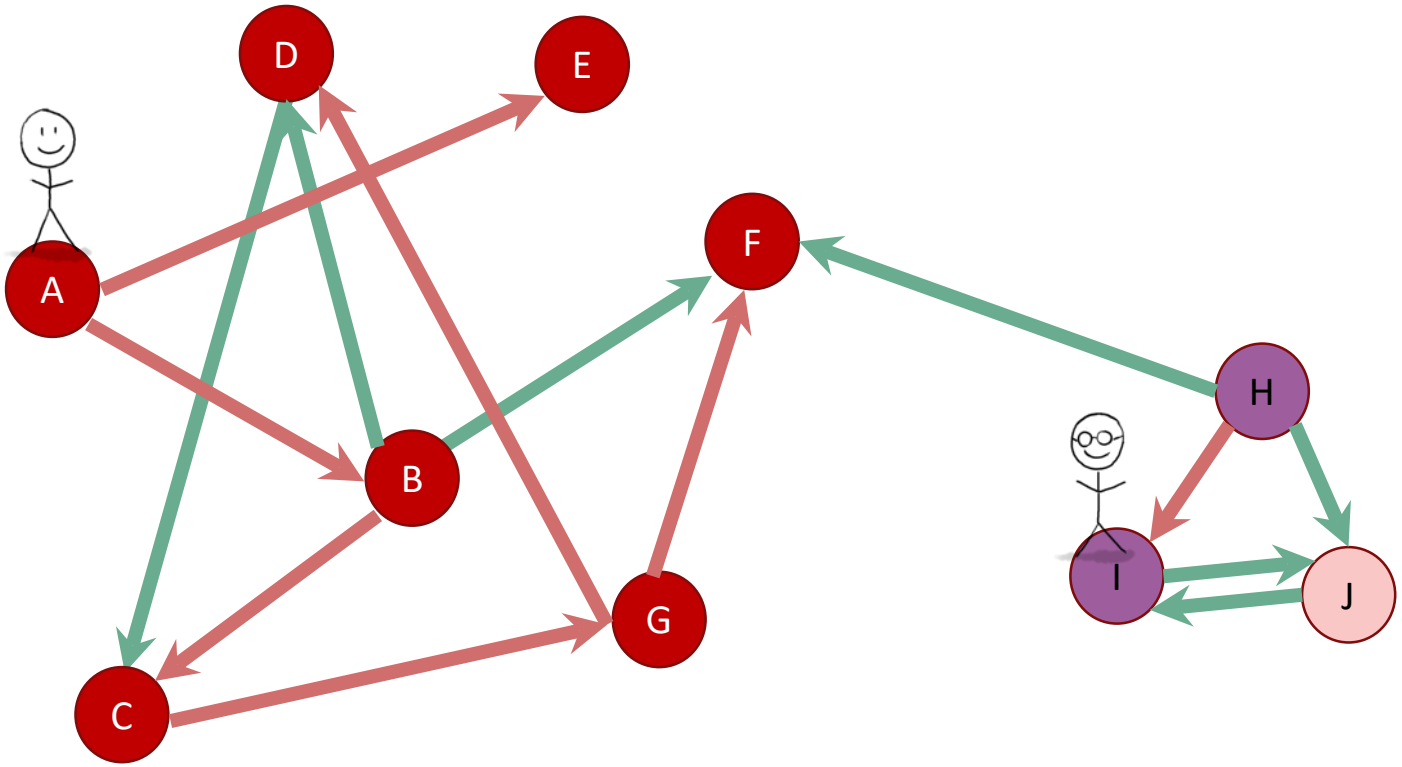
When you can't reach everything

- Run DFS repeatedly to get a depth-first forest



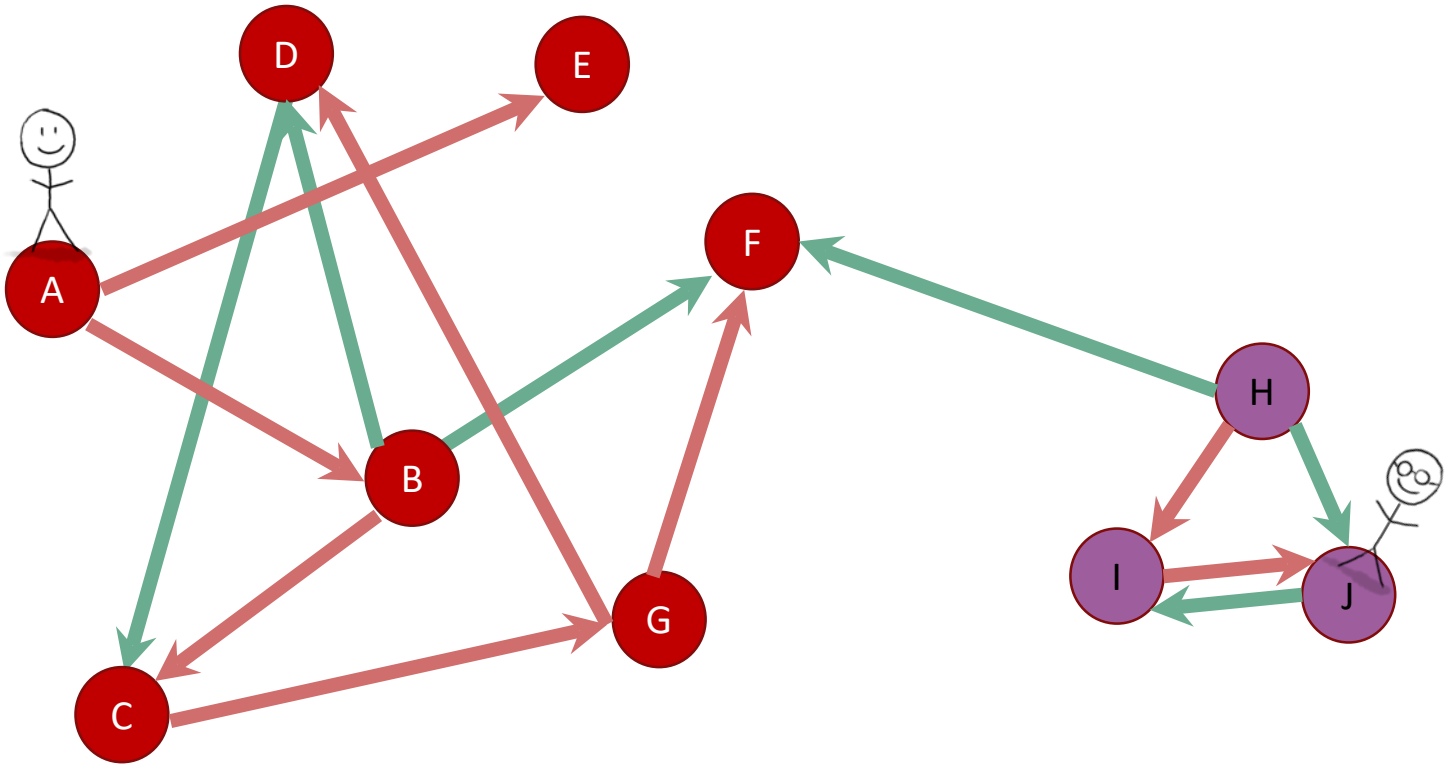
When you can't reach everything

- Run DFS repeatedly to get a depth-first forest



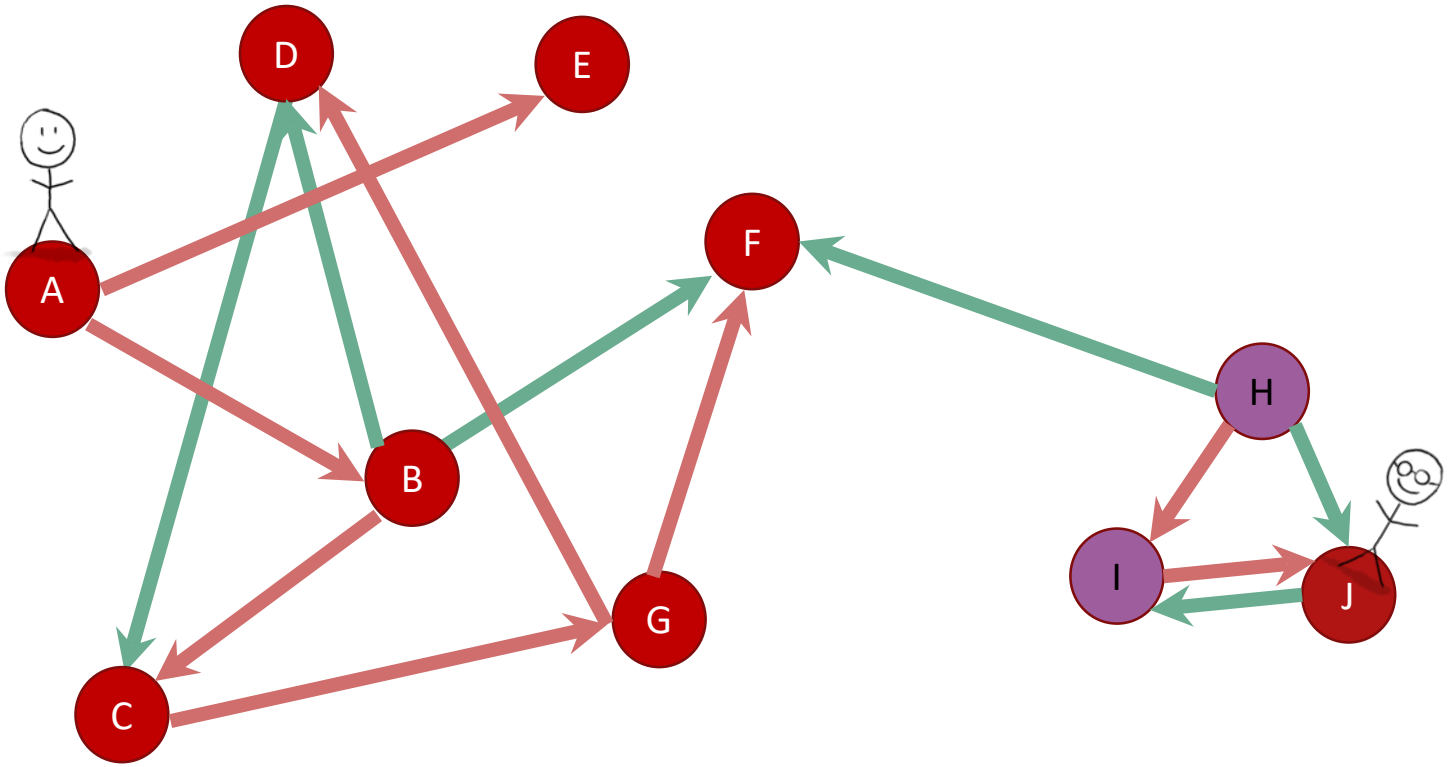
When you can't reach everything

- Run DFS repeatedly to get a depth-first forest



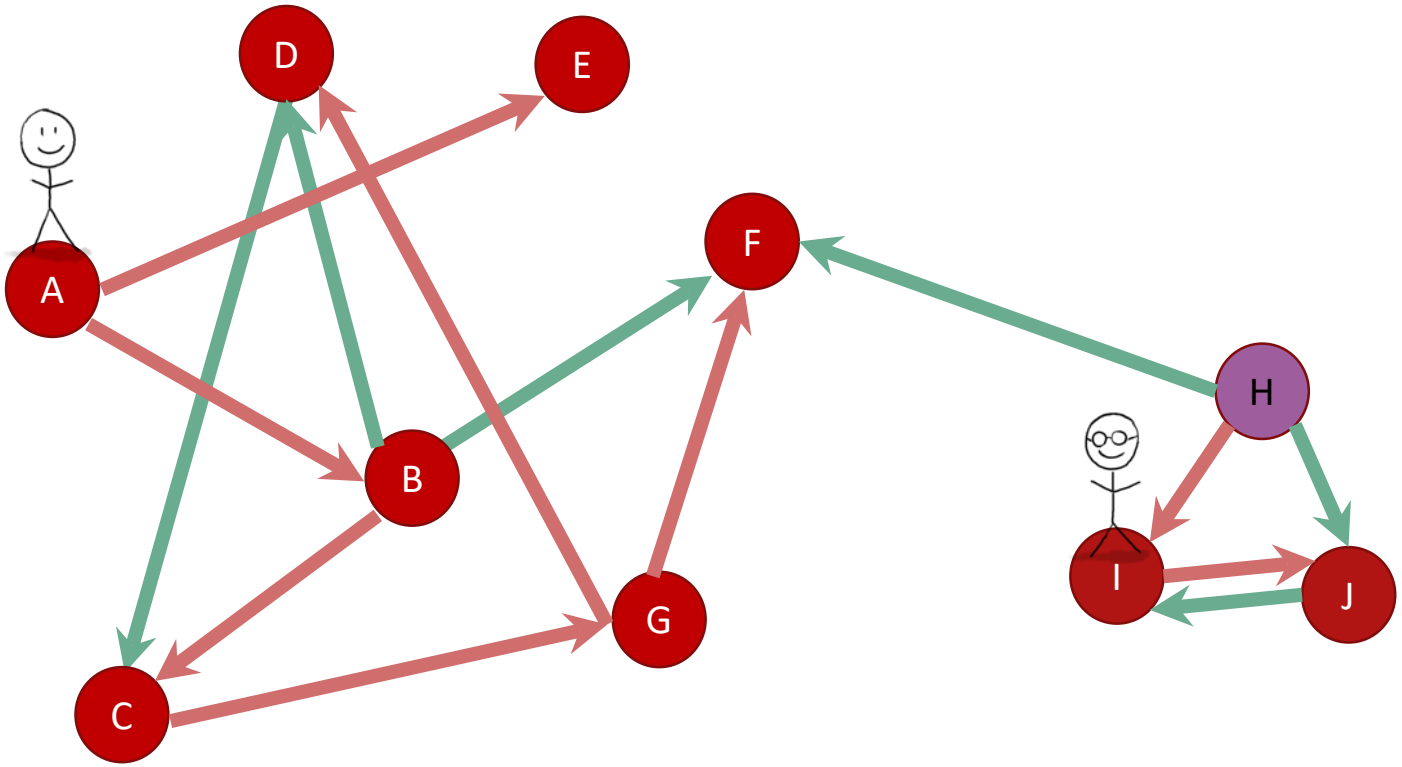
When you can't reach everything

- Run DFS repeatedly to get a depth-first forest



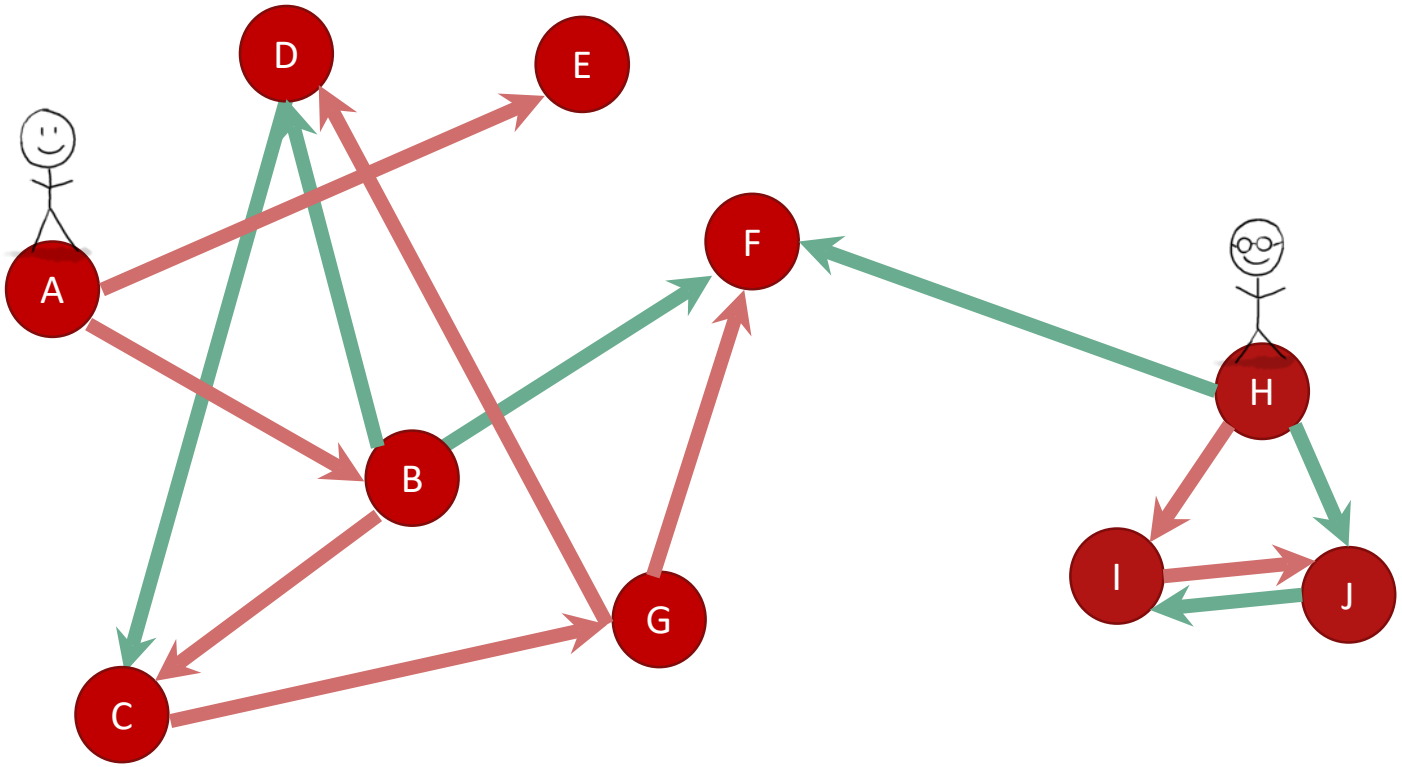
When you can't reach everything

- Run DFS repeatedly to get a depth-first forest



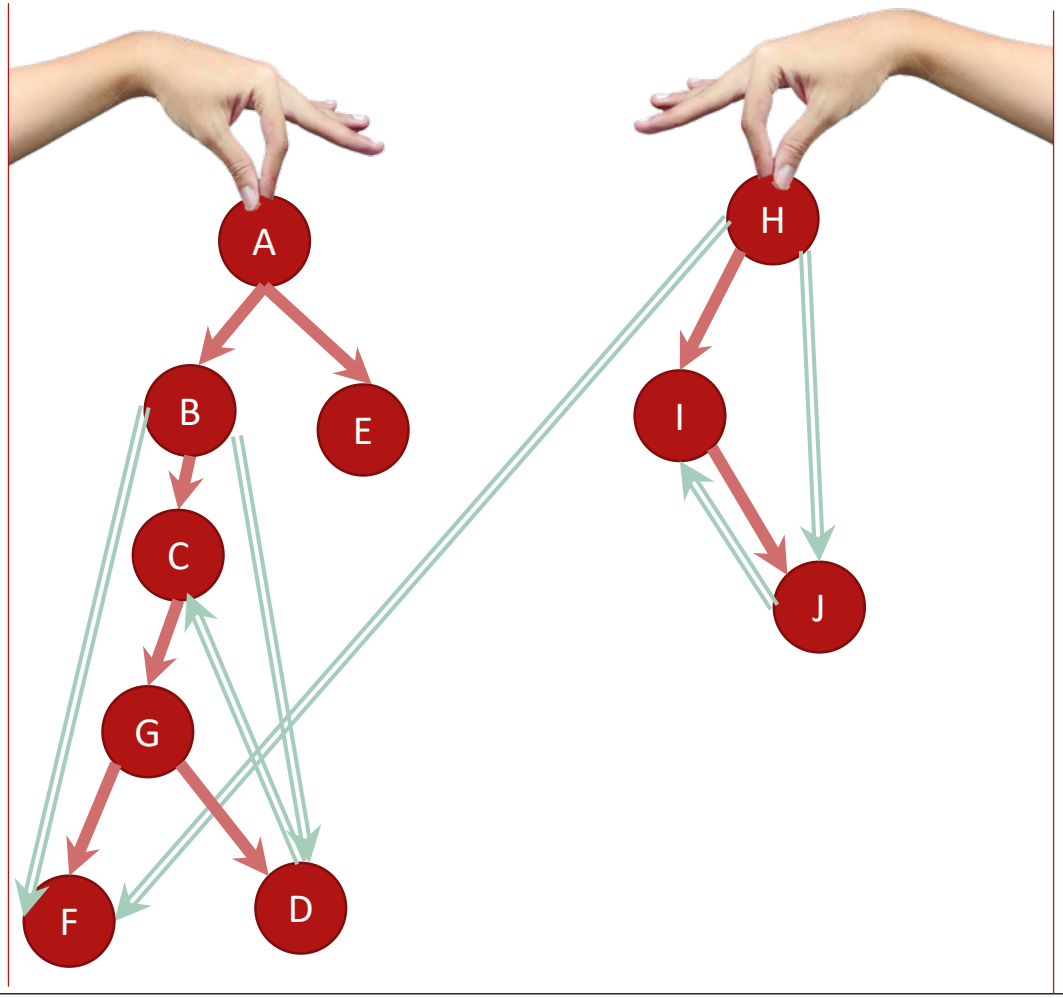
When you can't reach everything

- Run DFS repeatedly to get a depth-first forest



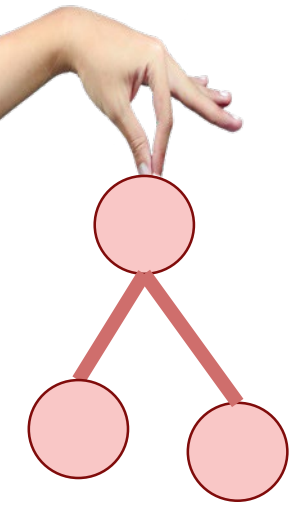
When you can’t reach everything

- The DFS forest is made up of DFS trees



Recall

DFS tree



If v and w are in different trees, it's always this last one.

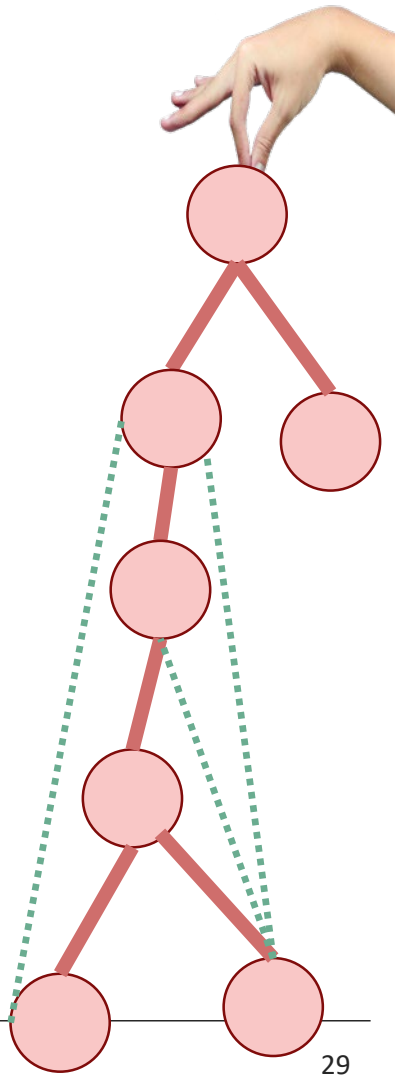
If v is a descendent of w in this tree:

If w is a descendent of v in this tree:

If neither are descendants of each other:

(or the other way around)

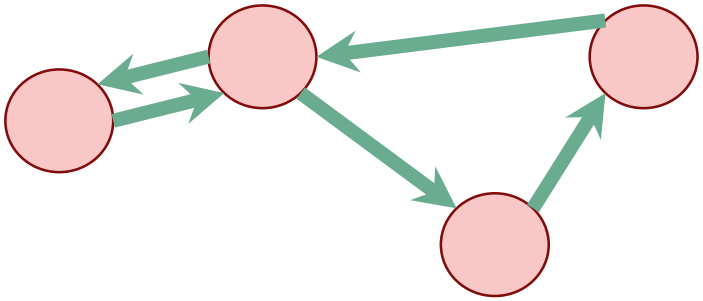
DFS tree



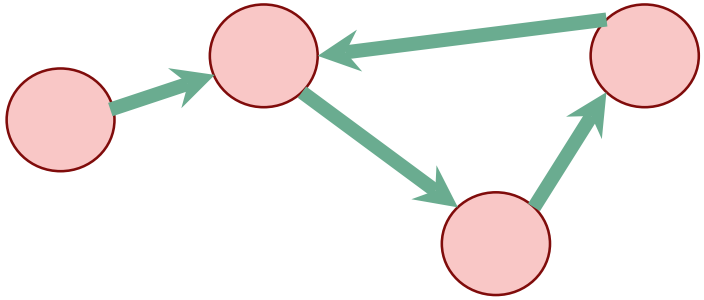
Strongly Connected Components

Strongly Connected Components (SCCs)

- Definition:
  - A directed graph G = (V,E) is **strongly connected** if:
  - for all v, w in V:
    - there is a path **from v to w** and
    - there is a path **from w to v**.



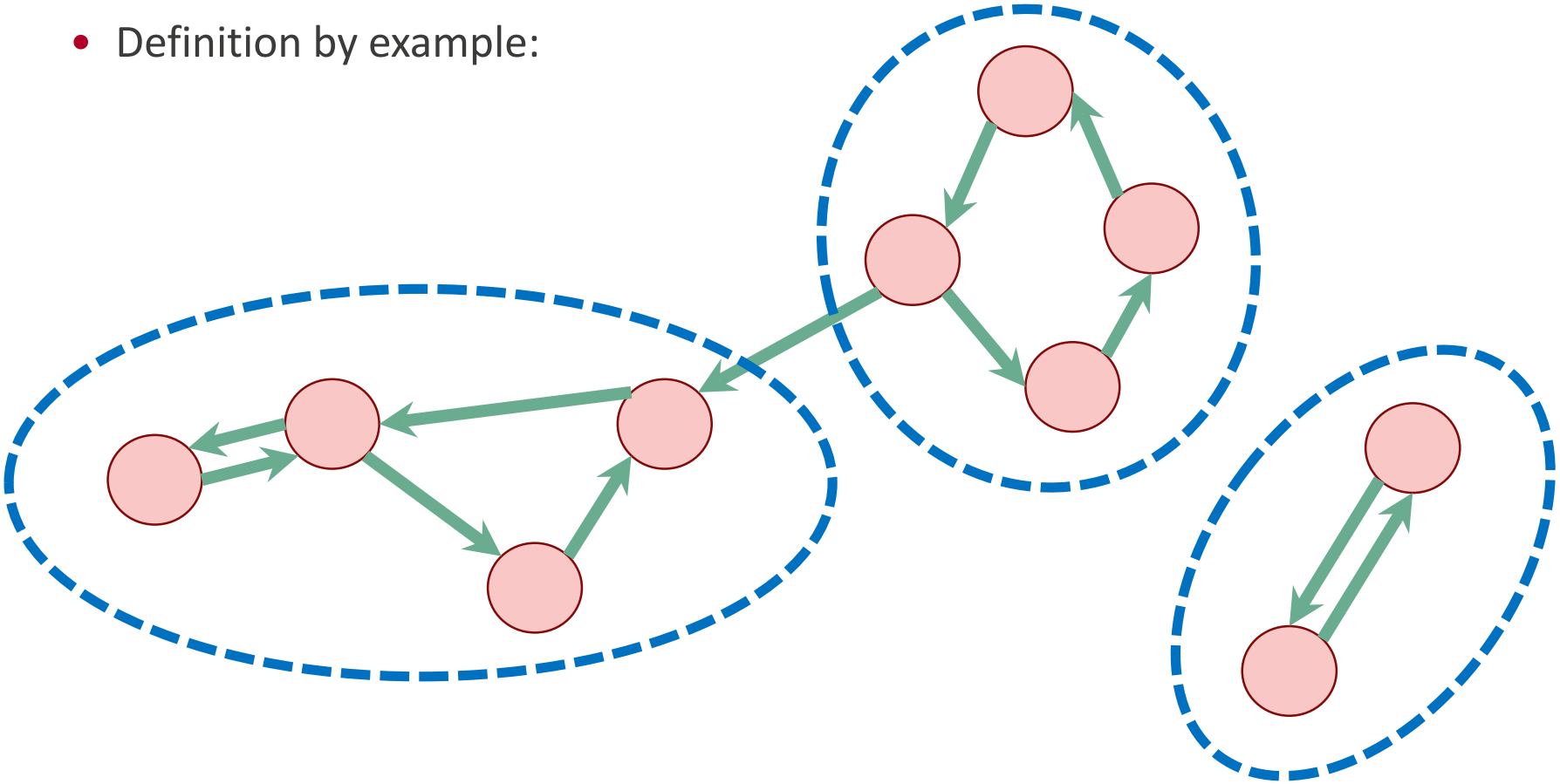
Strongly connected



Not strongly connected

Strongly Connected Components (SCCs)

- The SCC is an equivalence class in a **“mutually reachable”** equivalence relation.
- Definition by example:



Why do we care about SCCs?

- Strongly connected components tell you about **many things.**
- Sometimes we want to find the SCCs as a first step.
  - E.g., algorithms for solving 2-SAT.

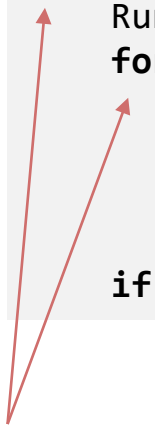
$$(x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$$

  - E.g., economist who has to first break up his labor market data into SCCs in order to make sense of it.

How to find SCCs?

- **Try 1:**
  - Consider all possible decompositions and check.
- **Try 2:**
  - For each pair (u, v), run DFS to find if there are paths u to v and v to u.
  - Aggregate that information to figure out the SCCs.

```
SCCs = [ ]
for each u:
    Run DFS from u
    for each vertex v that u can reach:
        if v is in an SCC we've already found:
            Run DFS from v to see if you can reach u
            if so, add u to v's SCC
            Break
    if we didn't break, create a new SCC which just contains
```



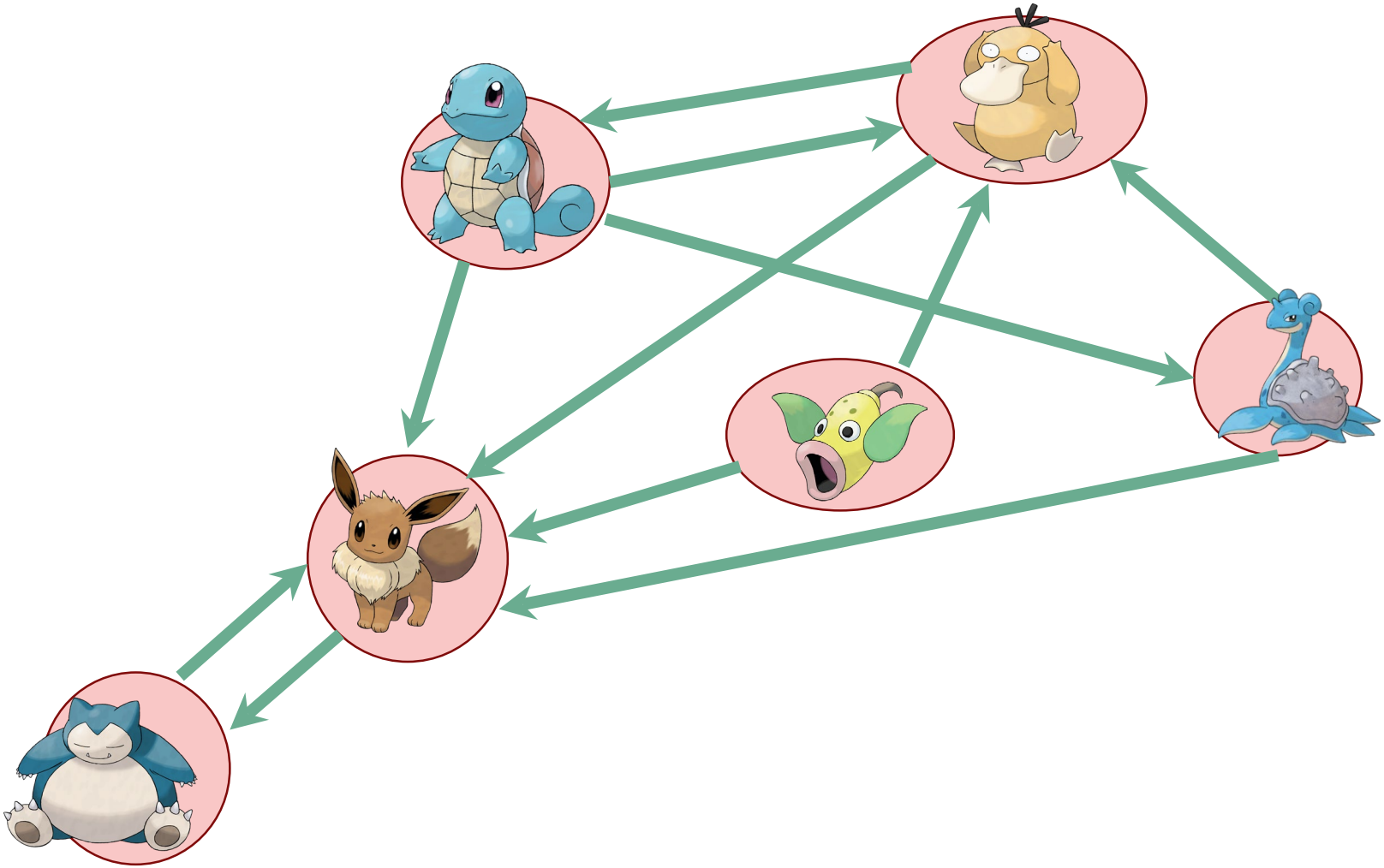
  - --> Definitely **not** any better than $O(n^2)$

Try 3 (Kosaraju's algorithm)

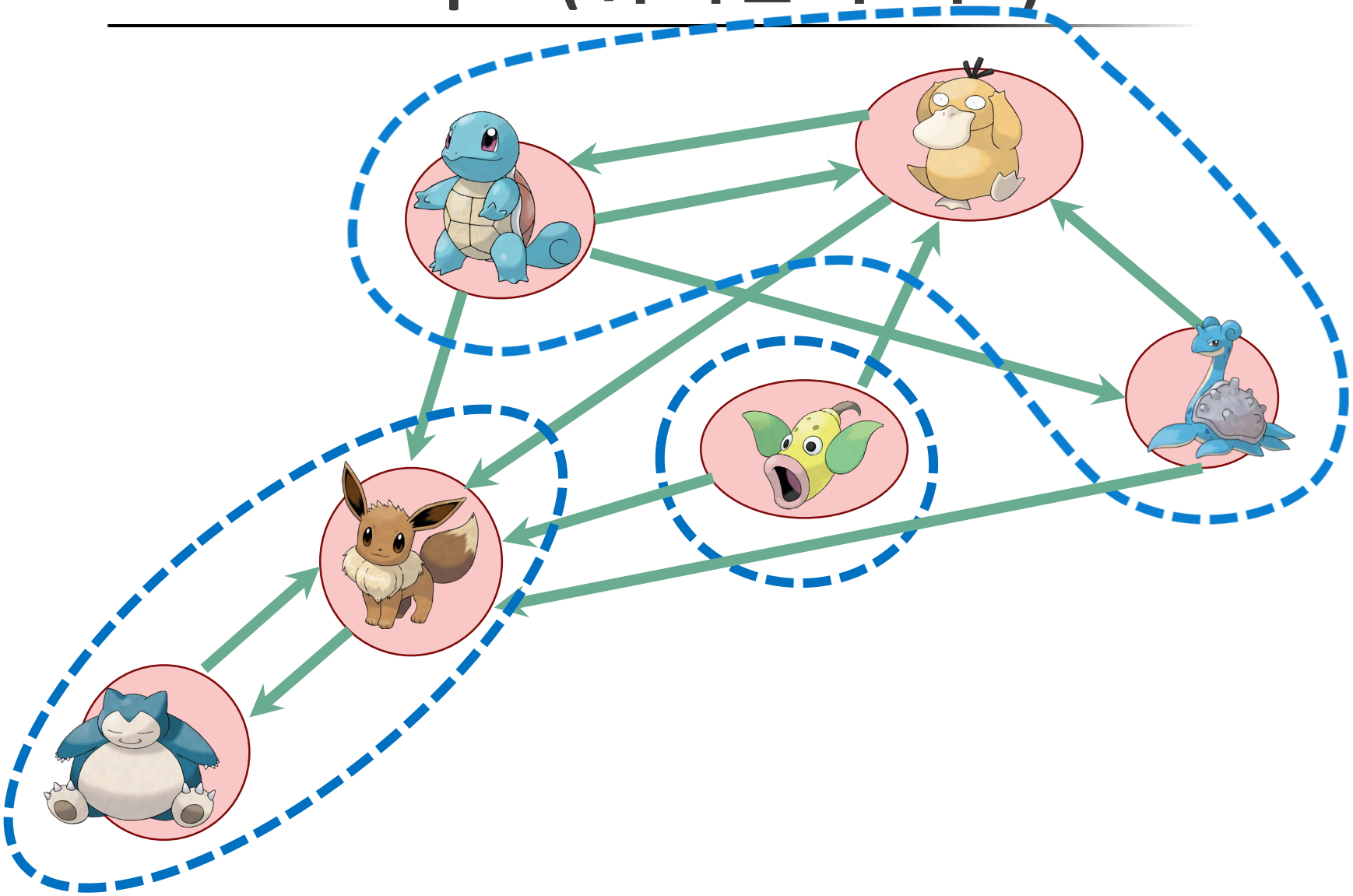
- Running time: O(n + m)

- How does it work?
  1. Do DFS to create a DFS forest.
     - Choose starting vertices in any order.
     - Keep track of finishing times.
  2. Reverse all the edges in the graph.
  3. Do DFS again to create another DFS forest.
     - This time, start the vertices in the reverse order of the finishing times that they had from the first DFS run.
  4. The SCCs are the different trees in the second DFS forest.

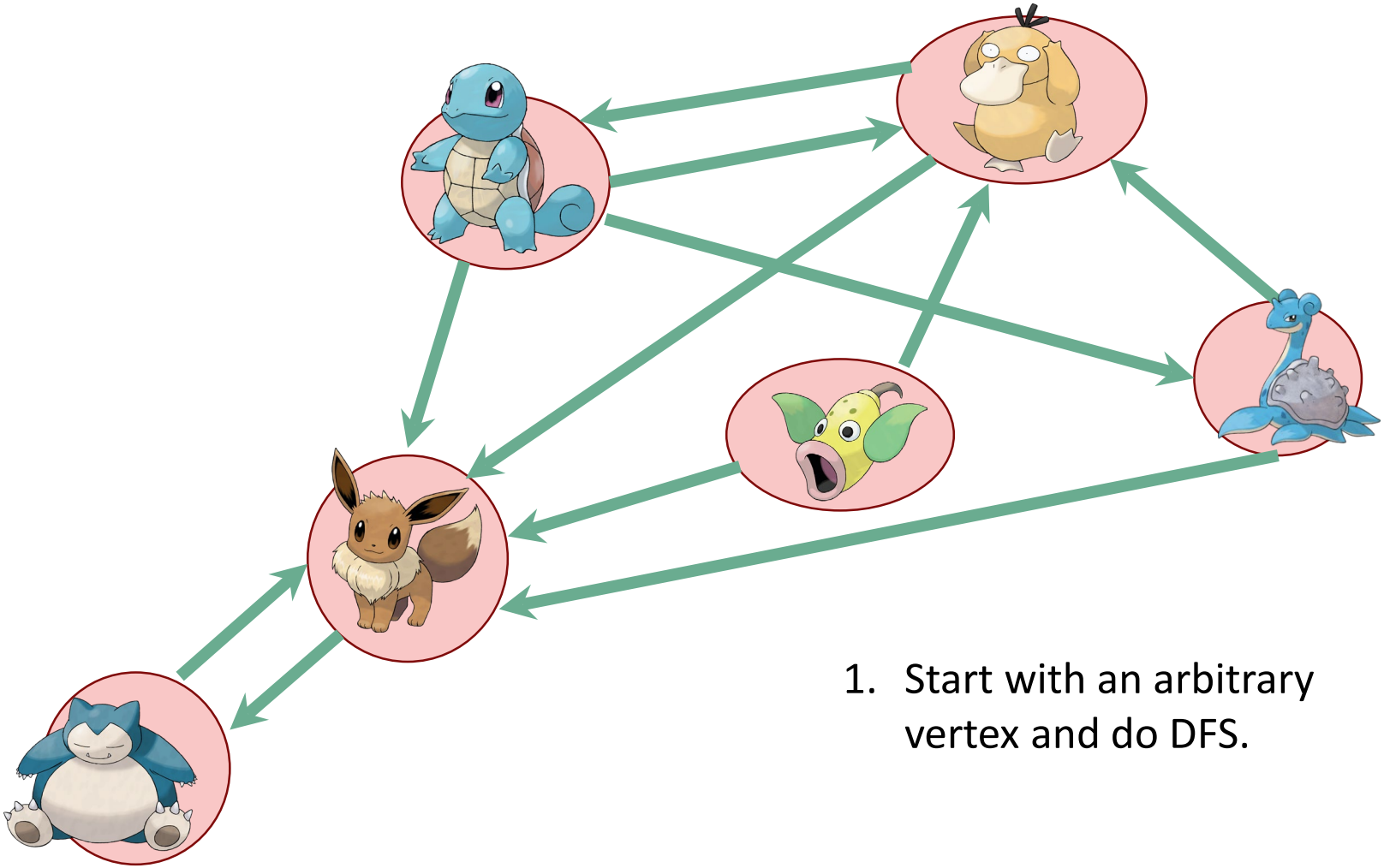
Example (귀여움 주의!!)



Example (귀여움 주의!!)

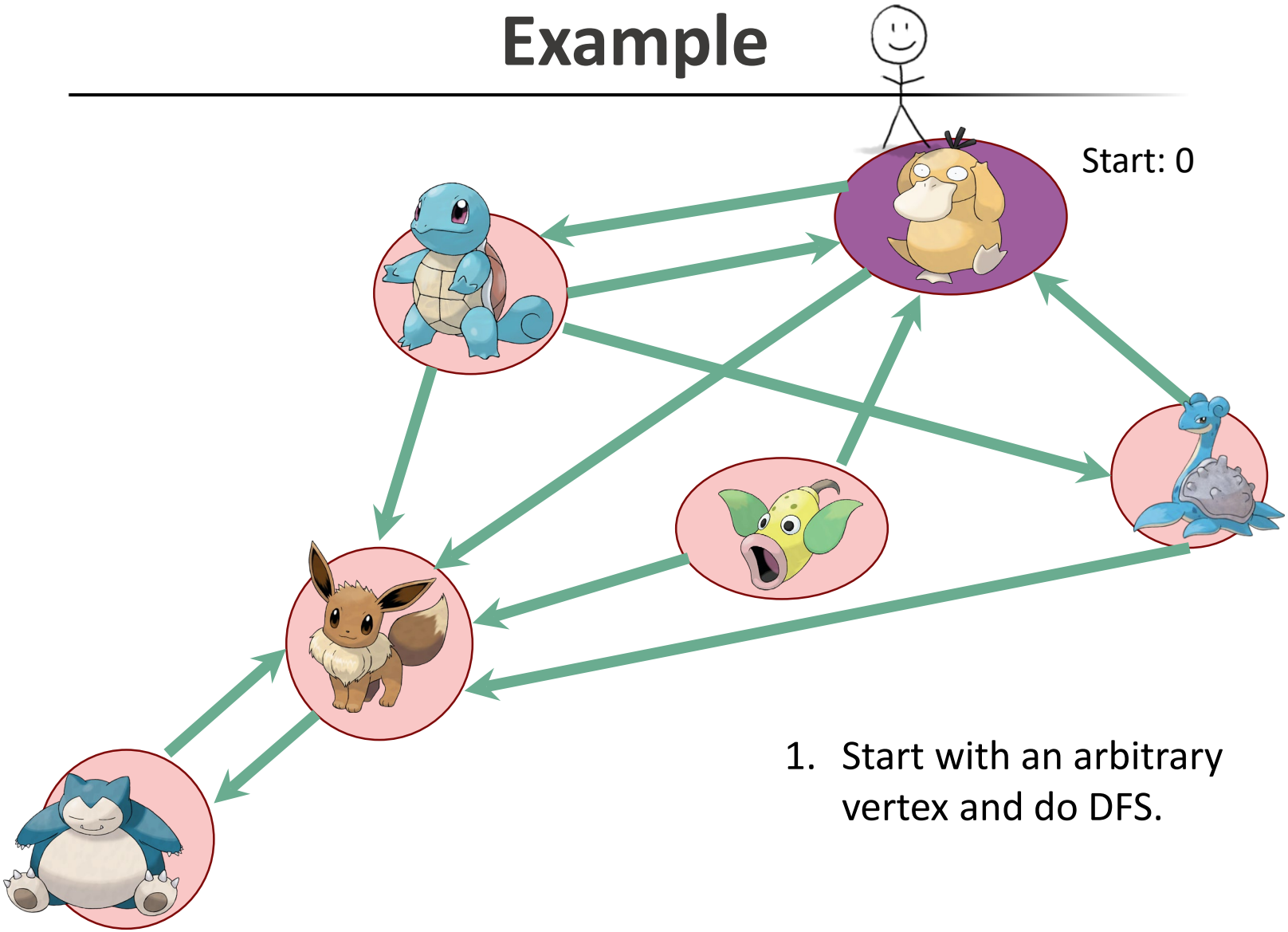


Example



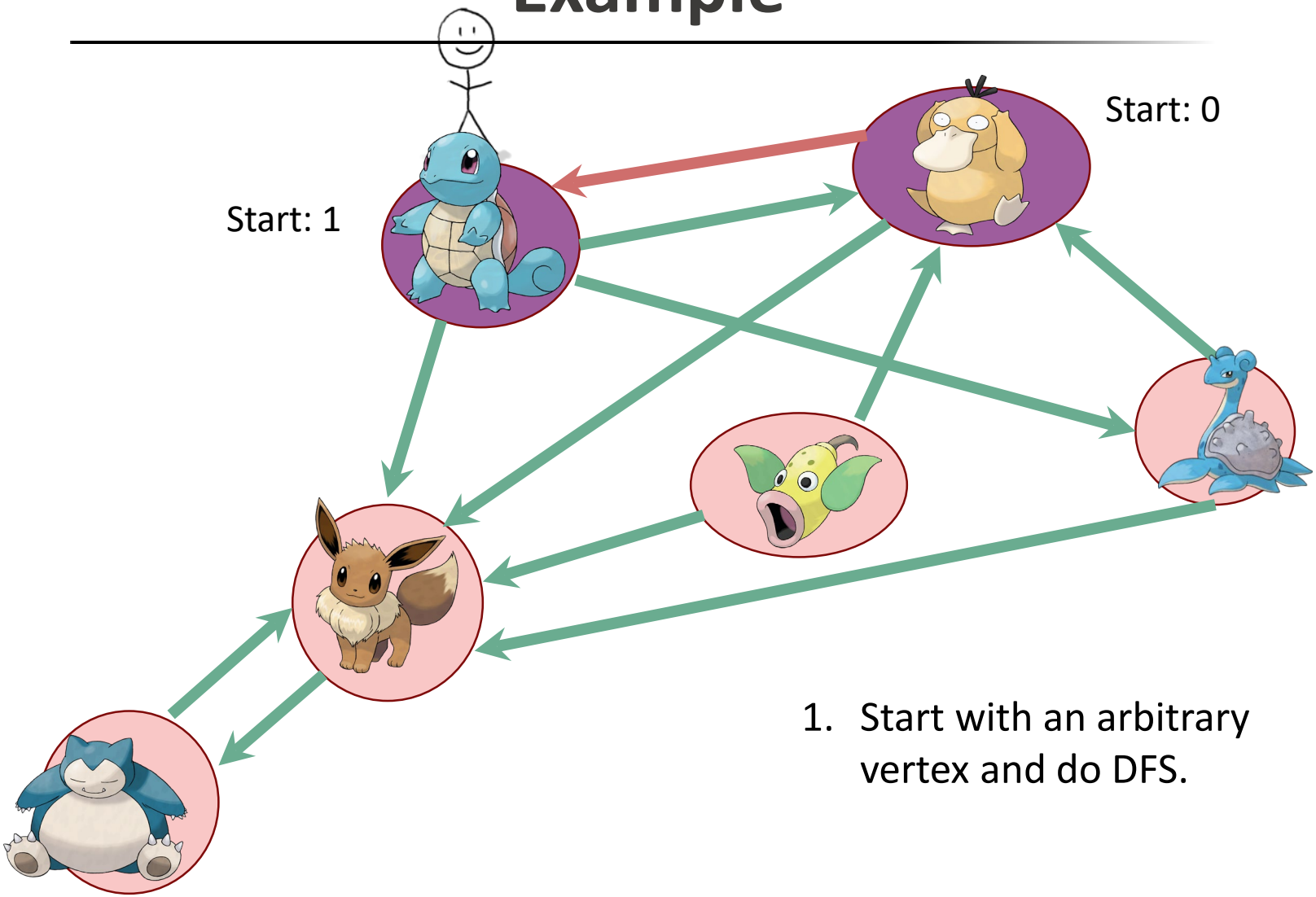
1. Start with an arbitrary vertex and do DFS.

Example



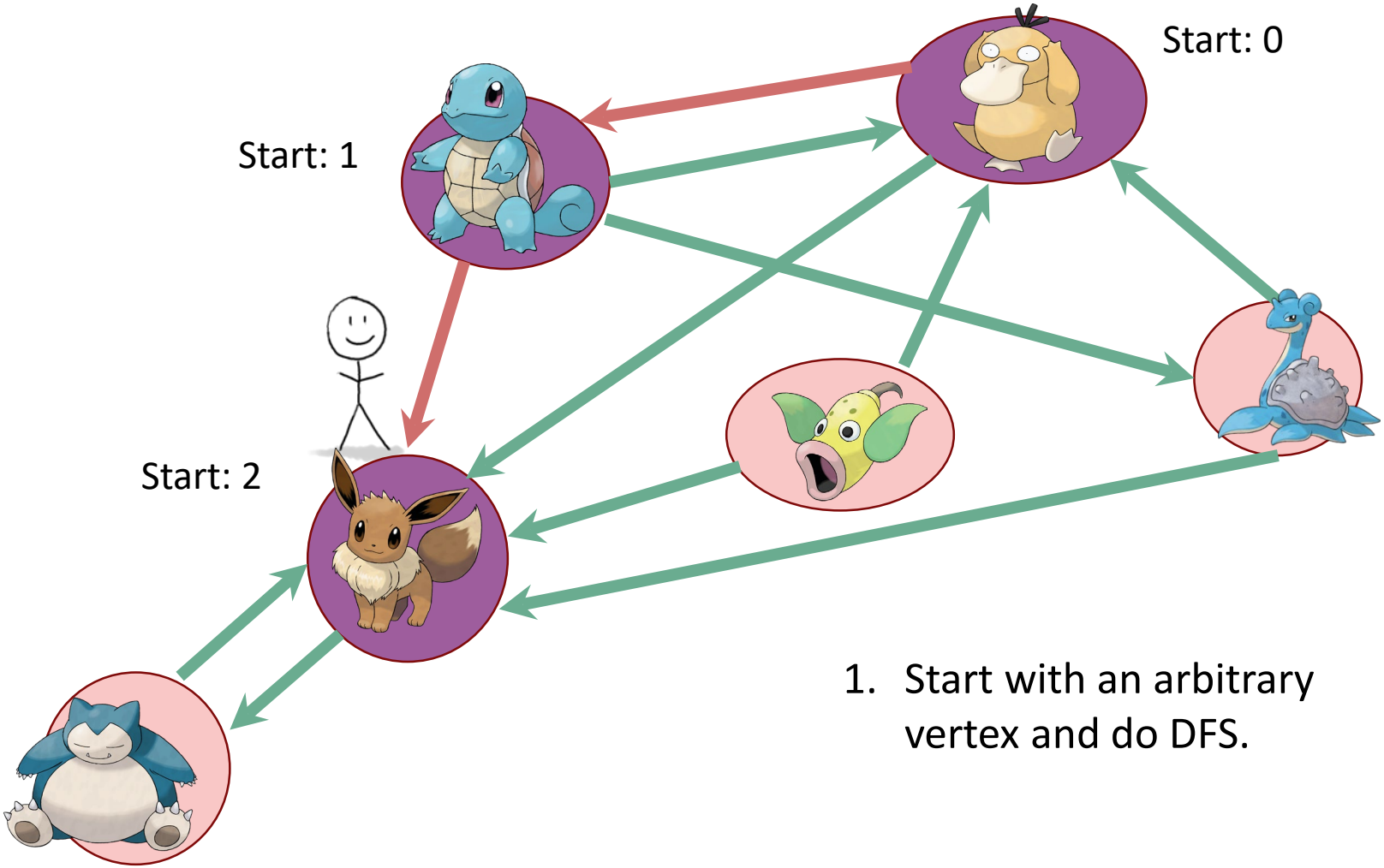
1. Start with an arbitrary vertex and do DFS.

Example



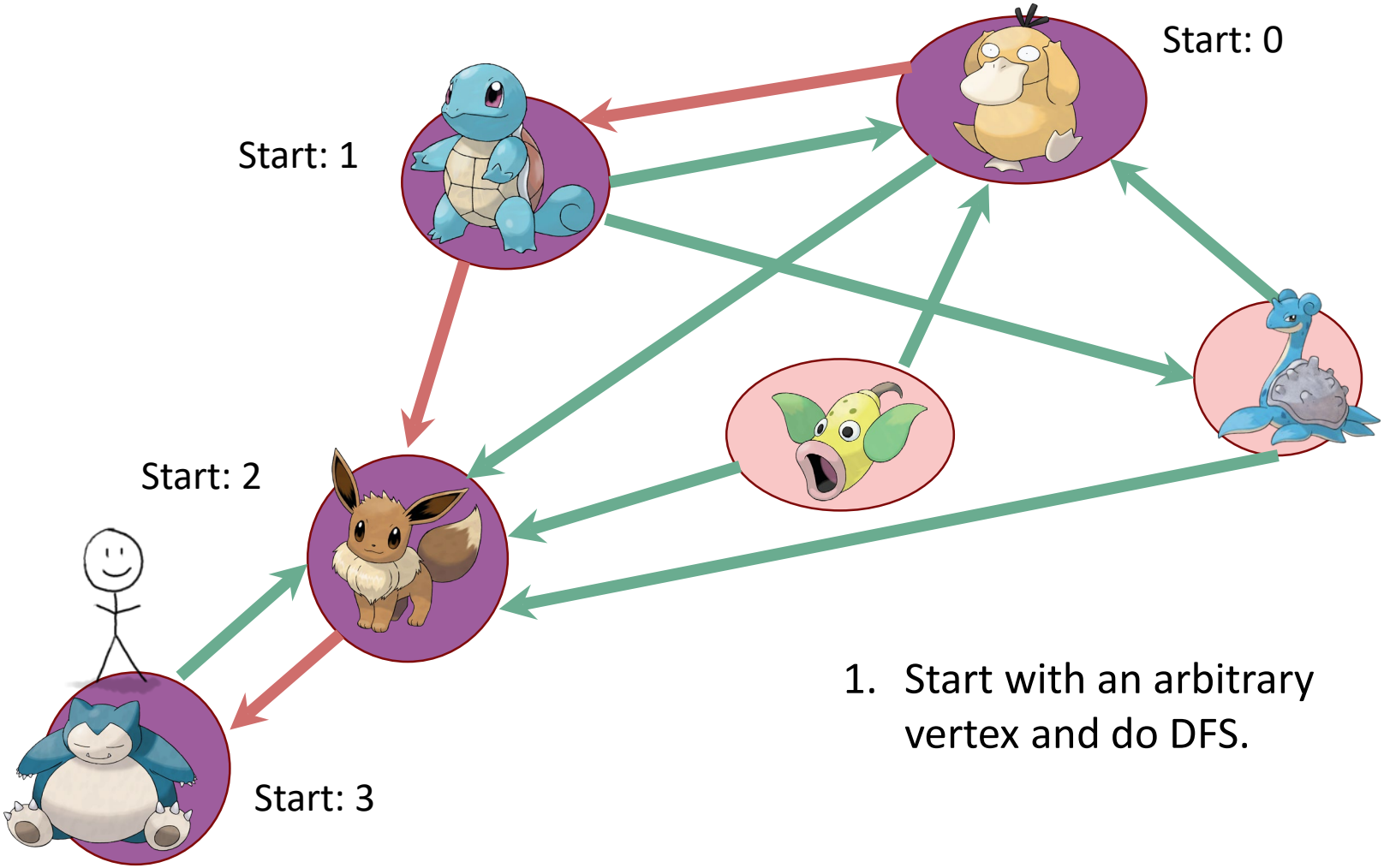
1. Start with an arbitrary vertex and do DFS.

Example



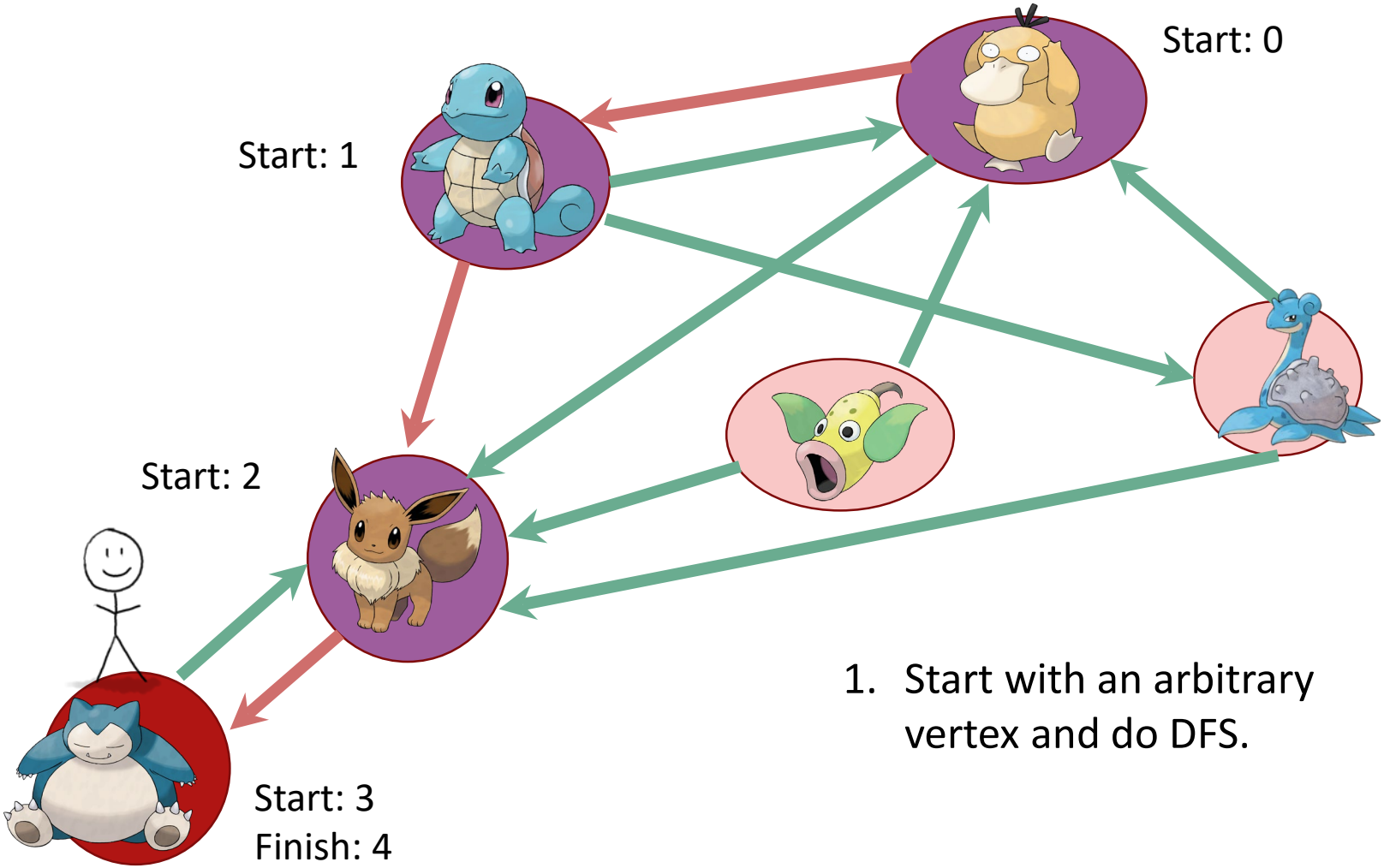
1. Start with an arbitrary vertex and do DFS.

Example



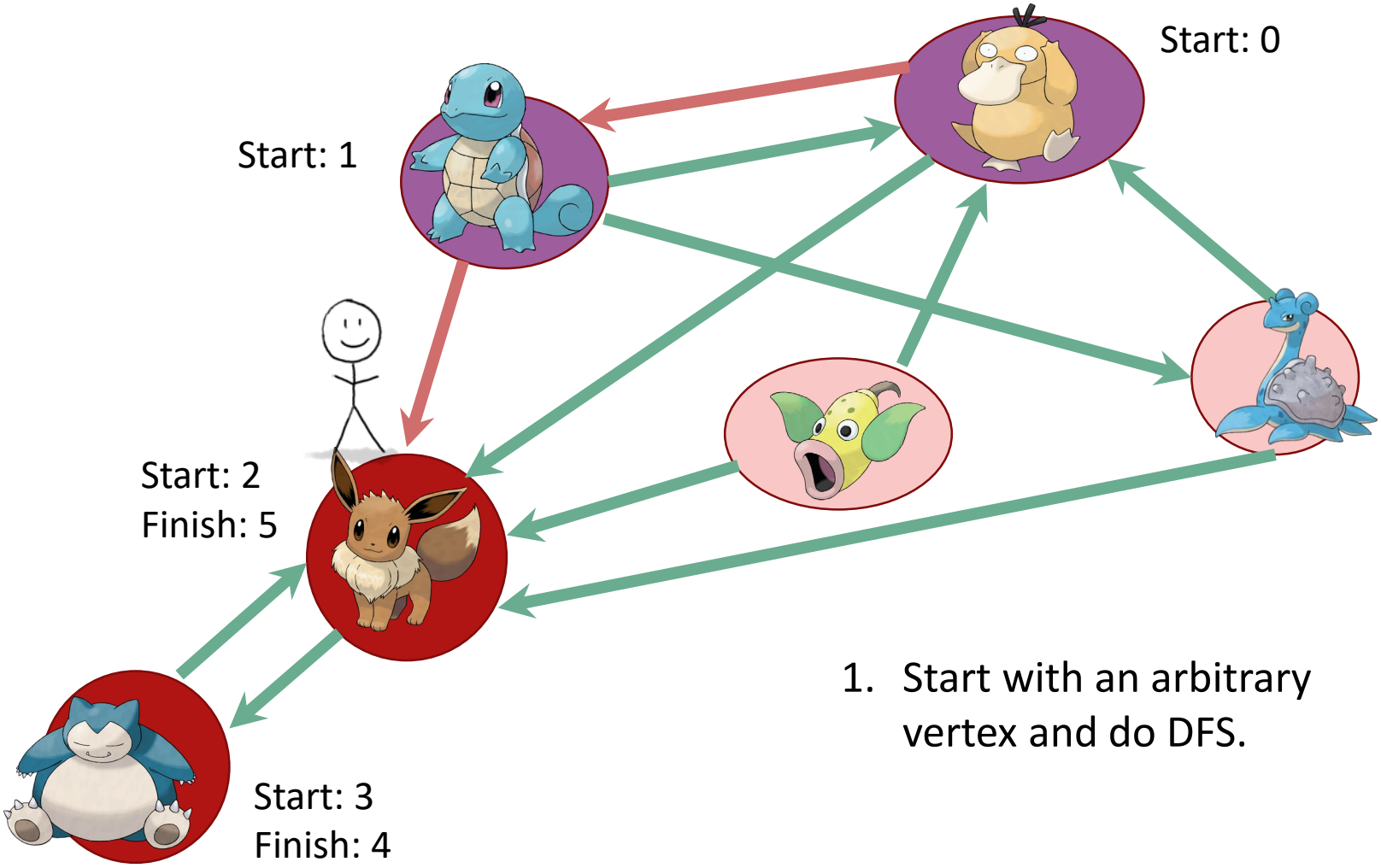
1. Start with an arbitrary vertex and do DFS.

Example



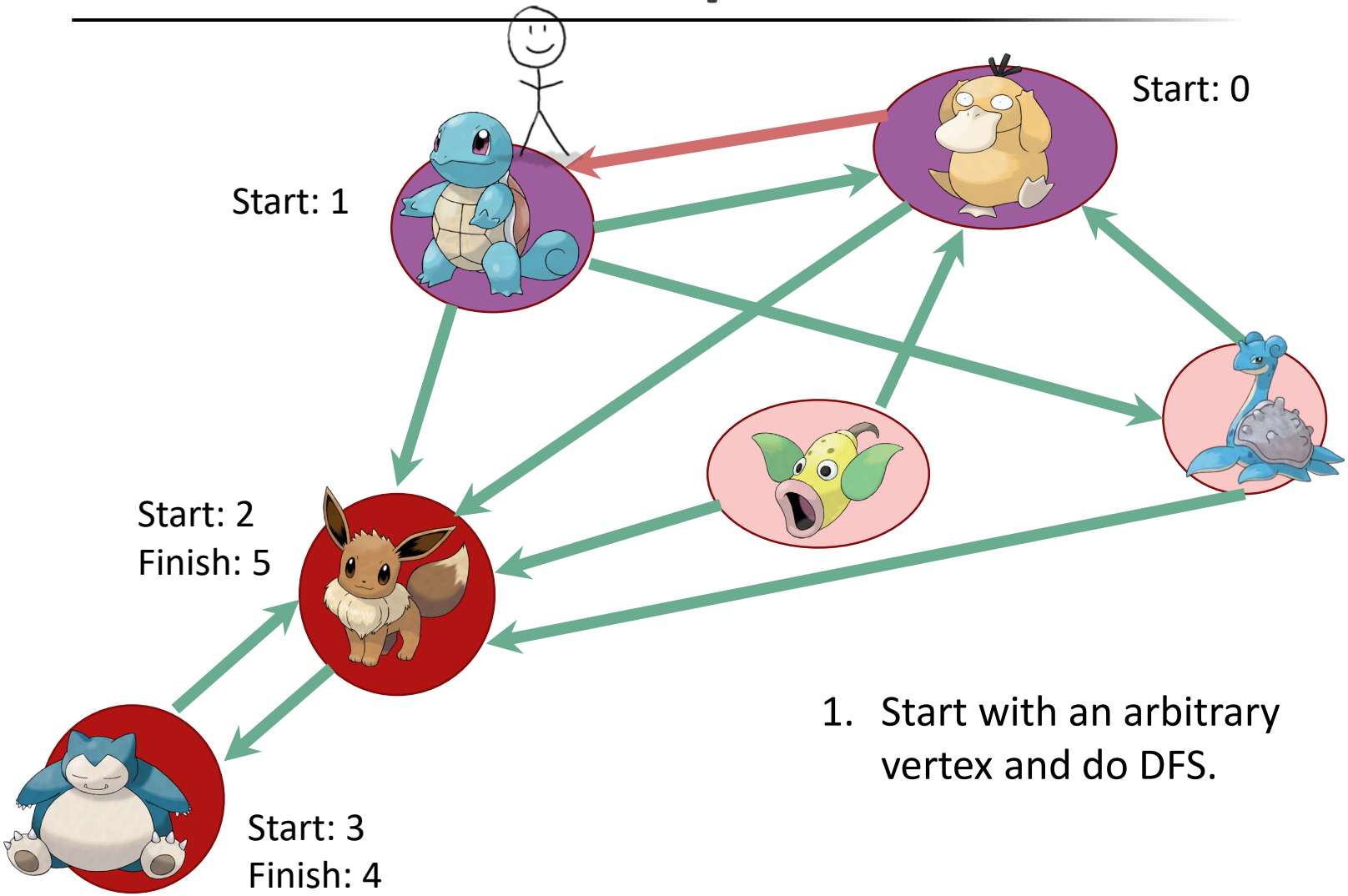
1. Start with an arbitrary vertex and do DFS.

Example



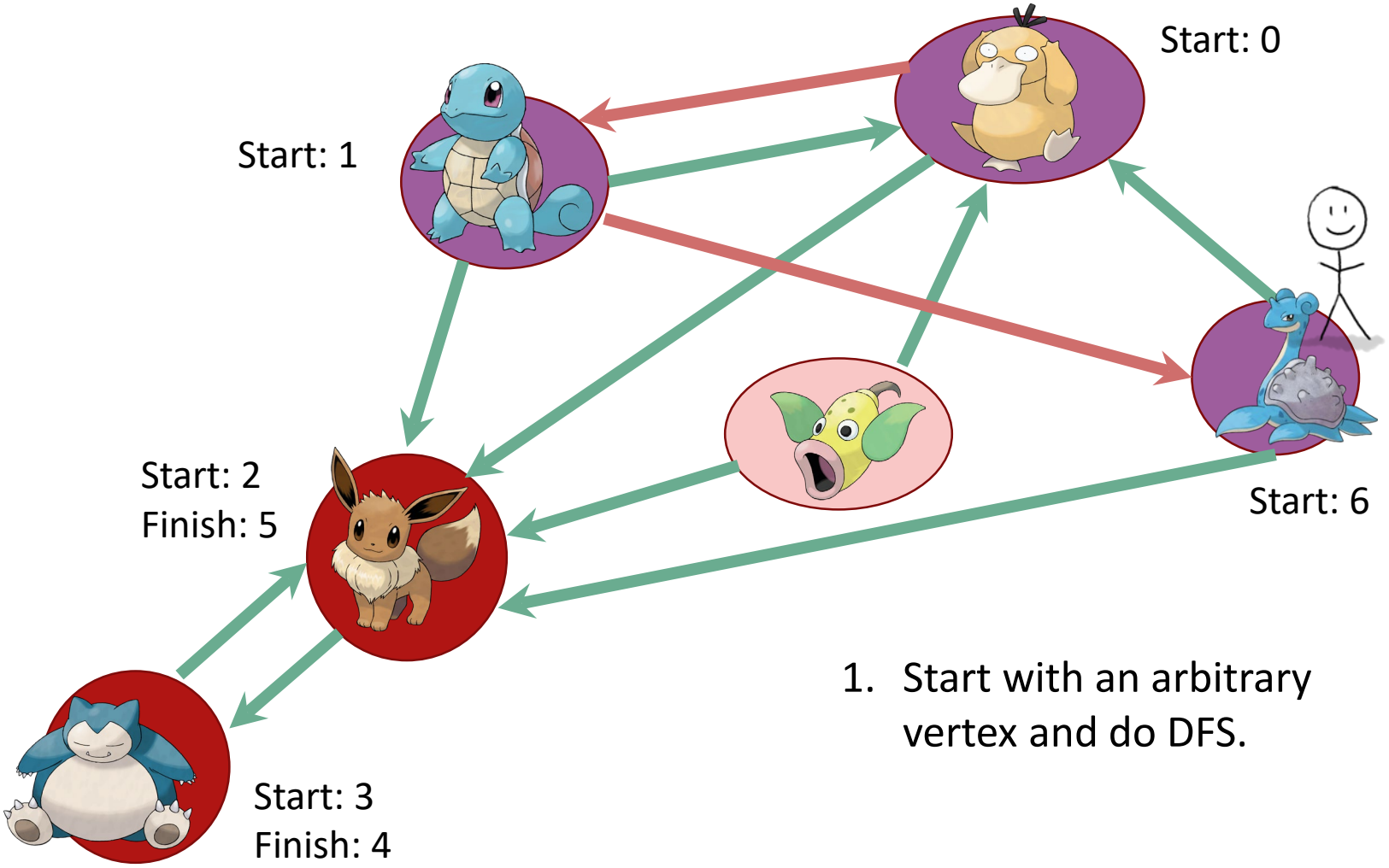
1. Start with an arbitrary vertex and do DFS.

Example



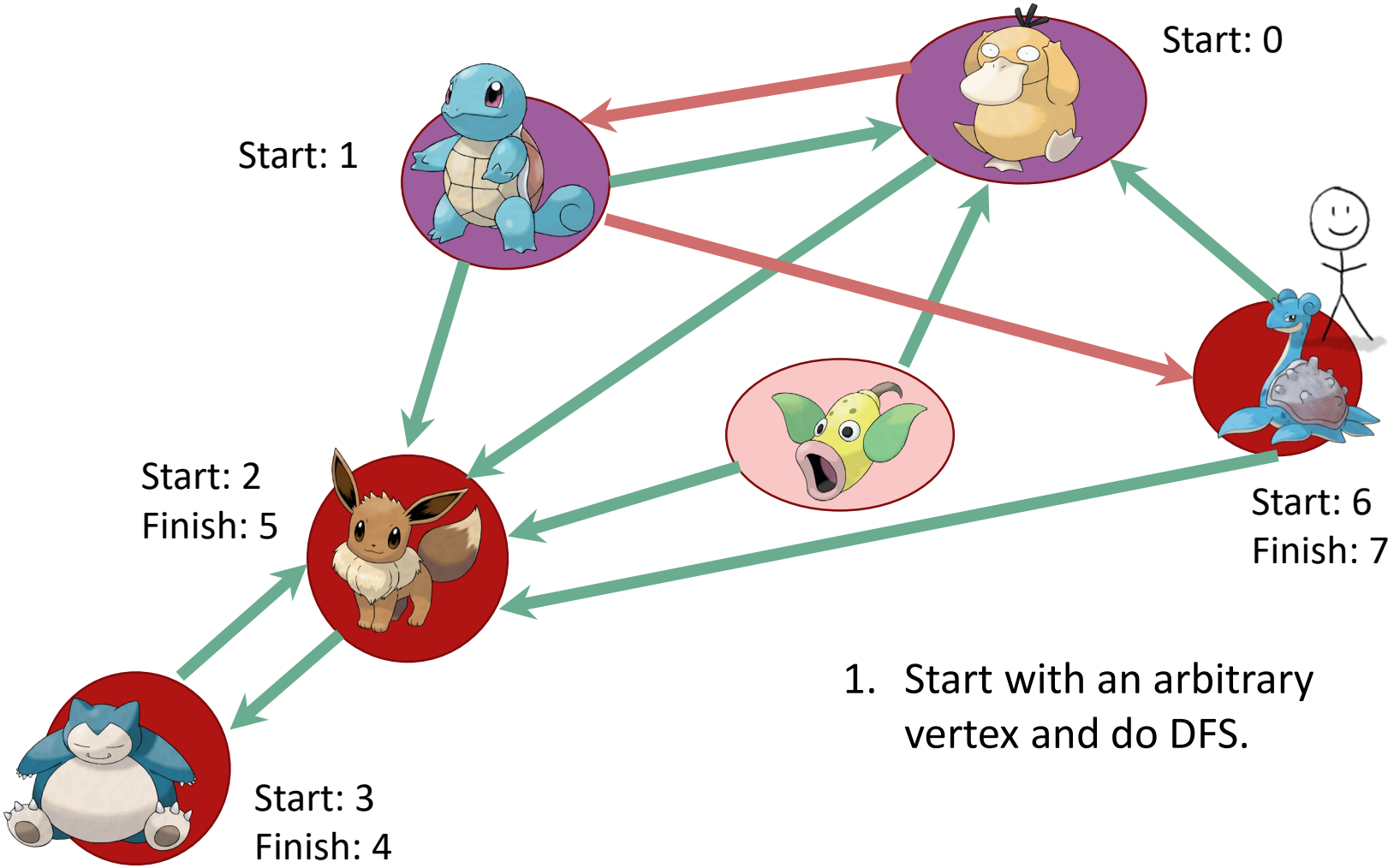
1. Start with an arbitrary vertex and do DFS.

Example



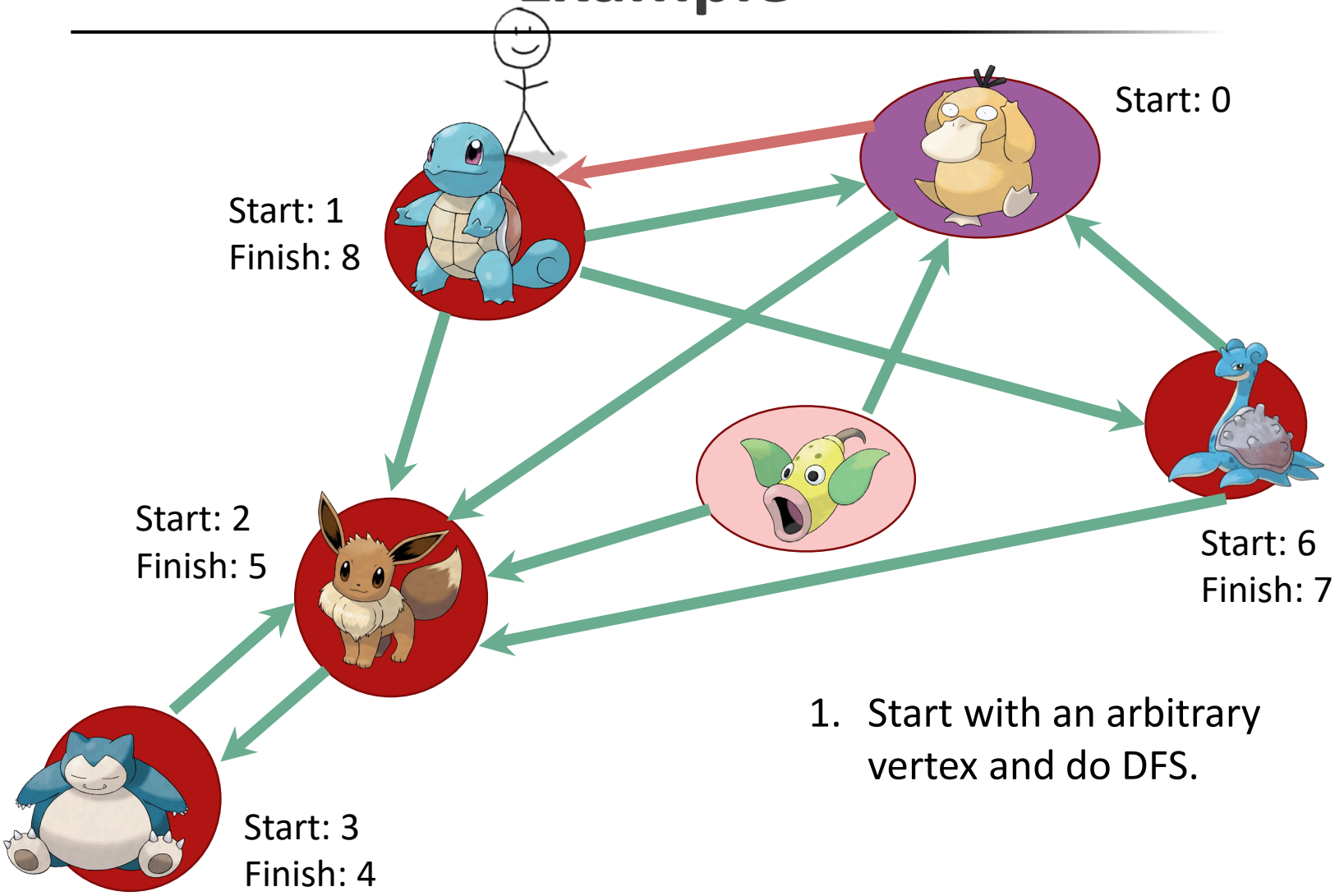
1. Start with an arbitrary vertex and do DFS.

Example



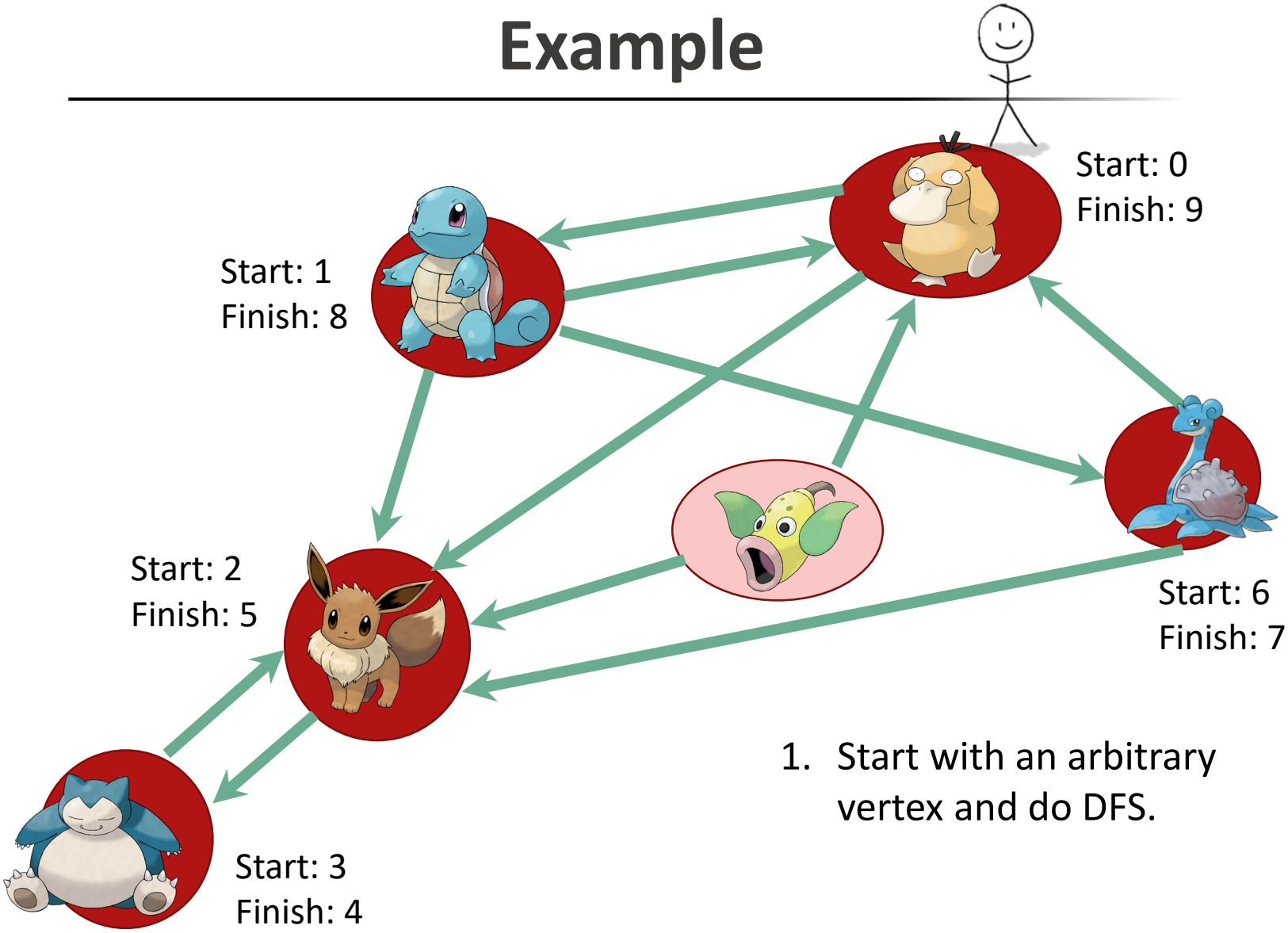
1. Start with an arbitrary vertex and do DFS.

Example



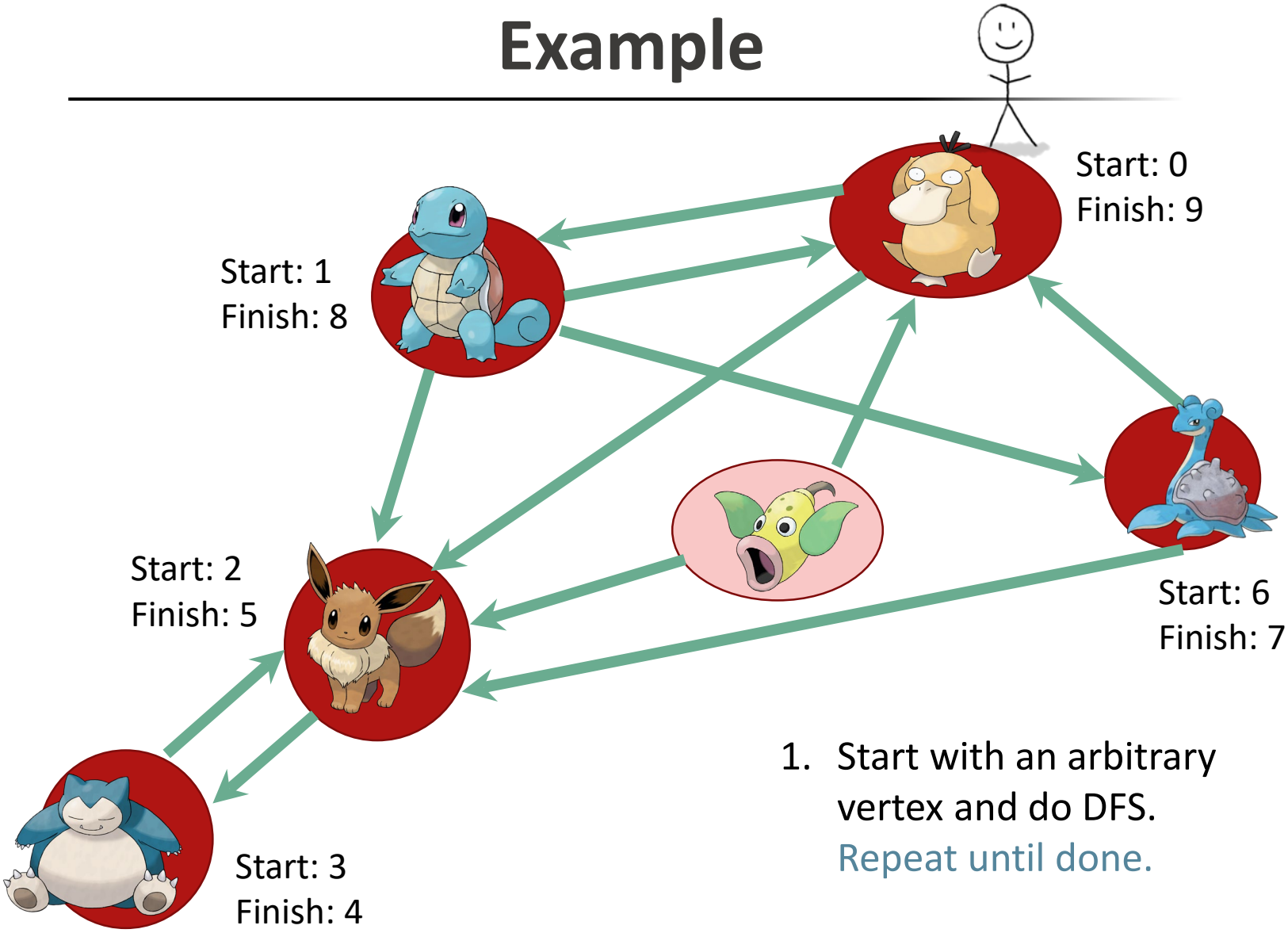
1. Start with an arbitrary vertex and do DFS.

Example



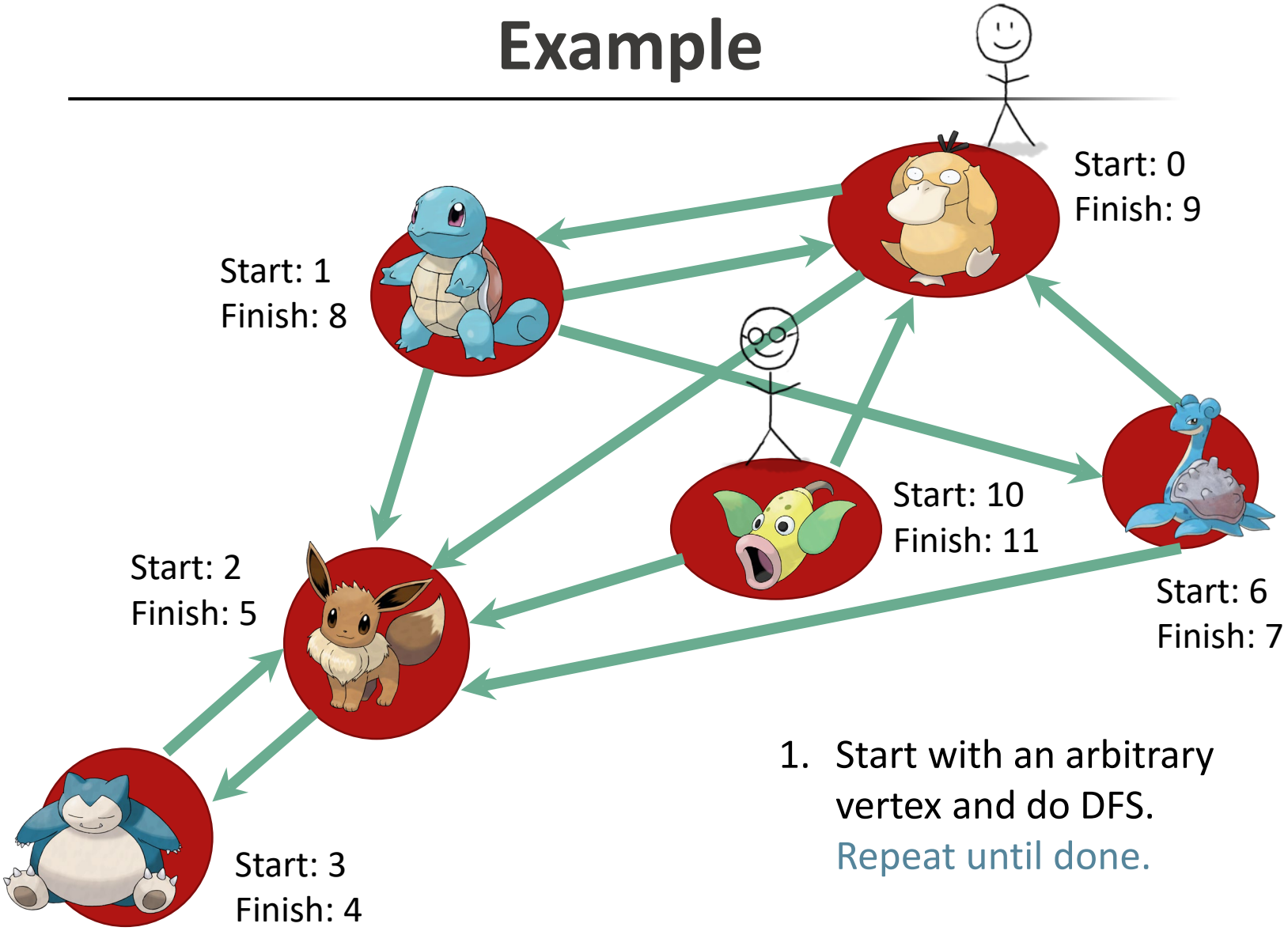
1. Start with an arbitrary vertex and do DFS.

Example



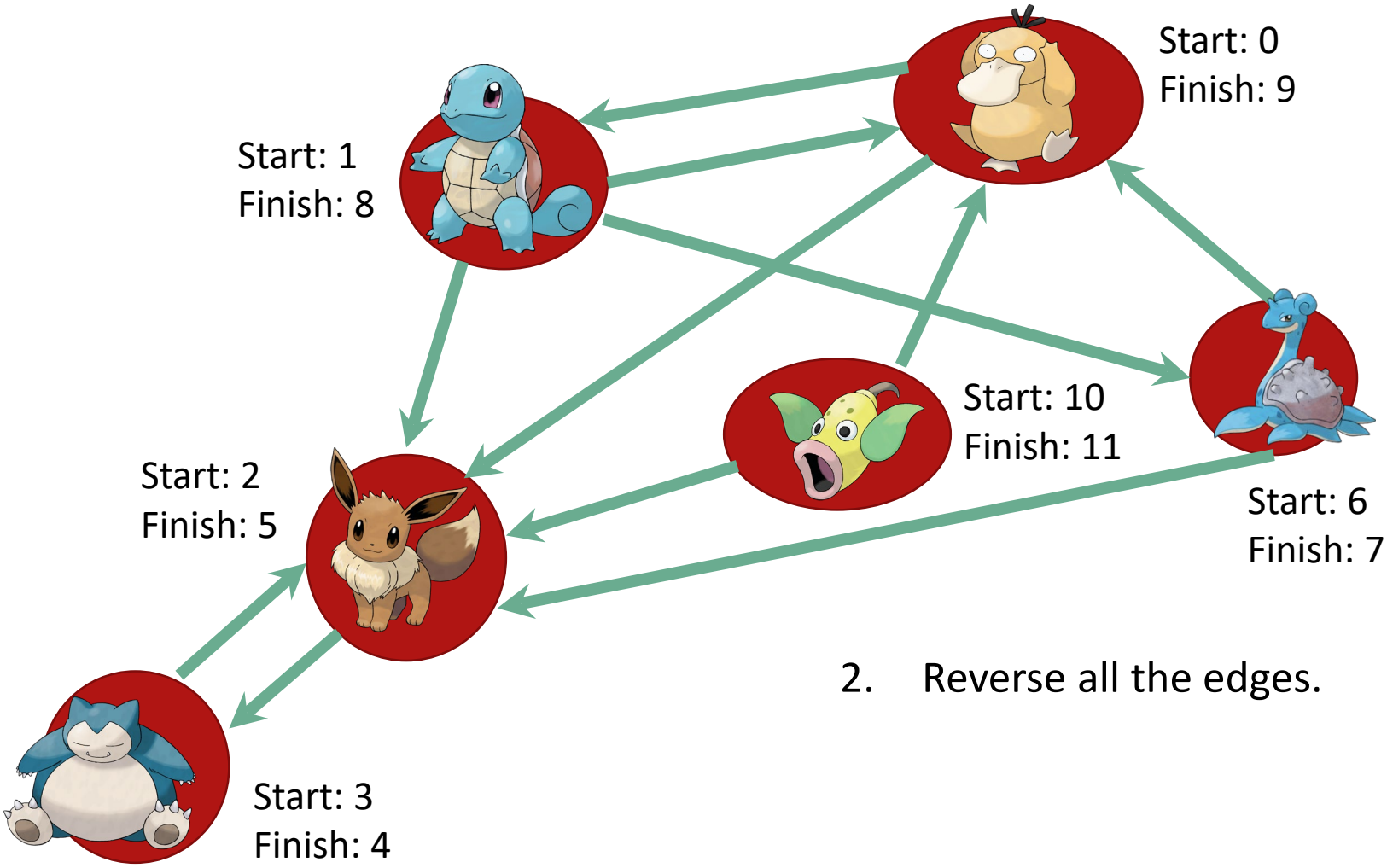
1. Start with an arbitrary vertex and do DFS.
   Repeat until done.

Example



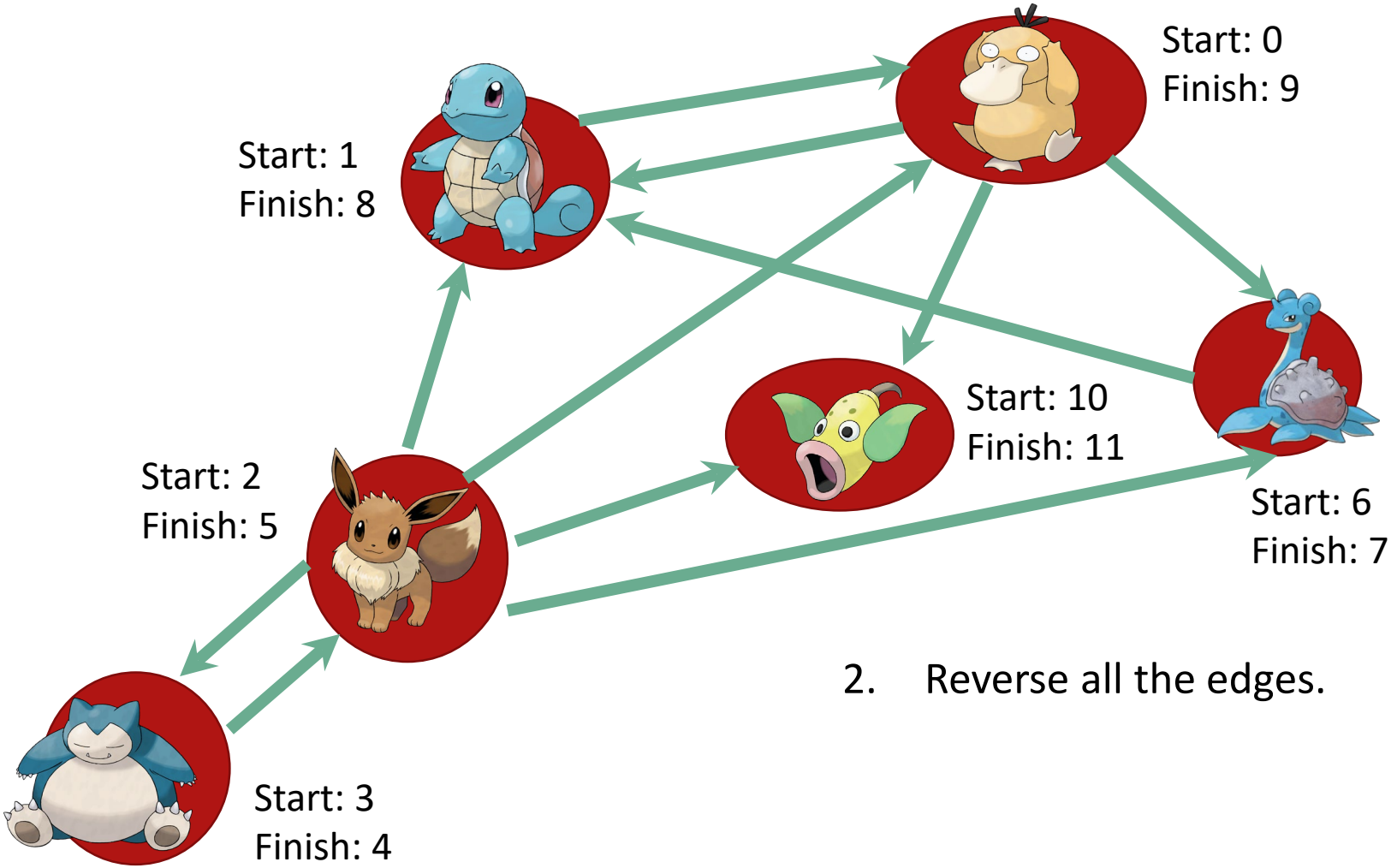
1. Start with an arbitrary vertex and do DFS.
   Repeat until done.

Example



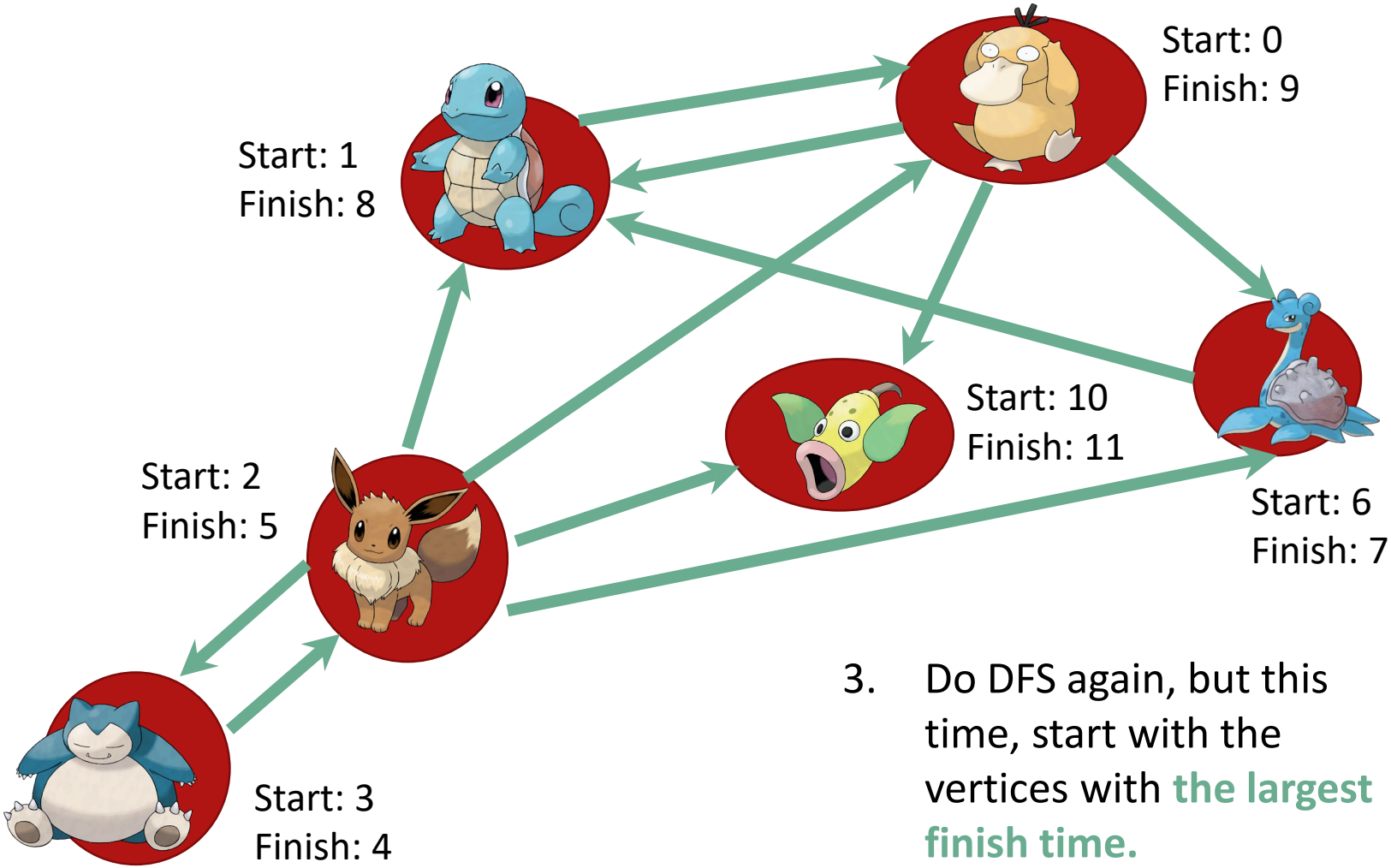
2. Reverse all the edges.

Example



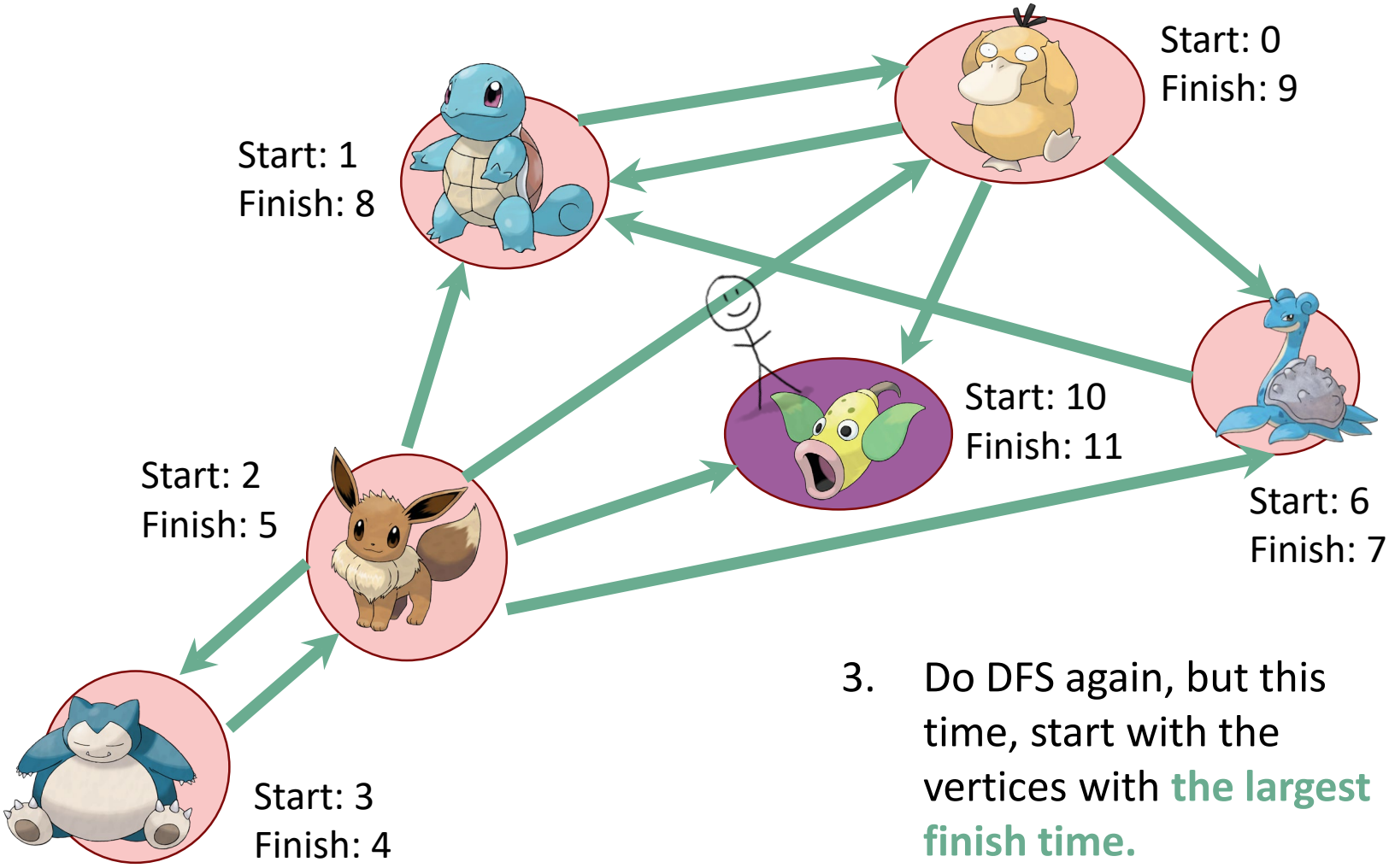
2. Reverse all the edges.

Example



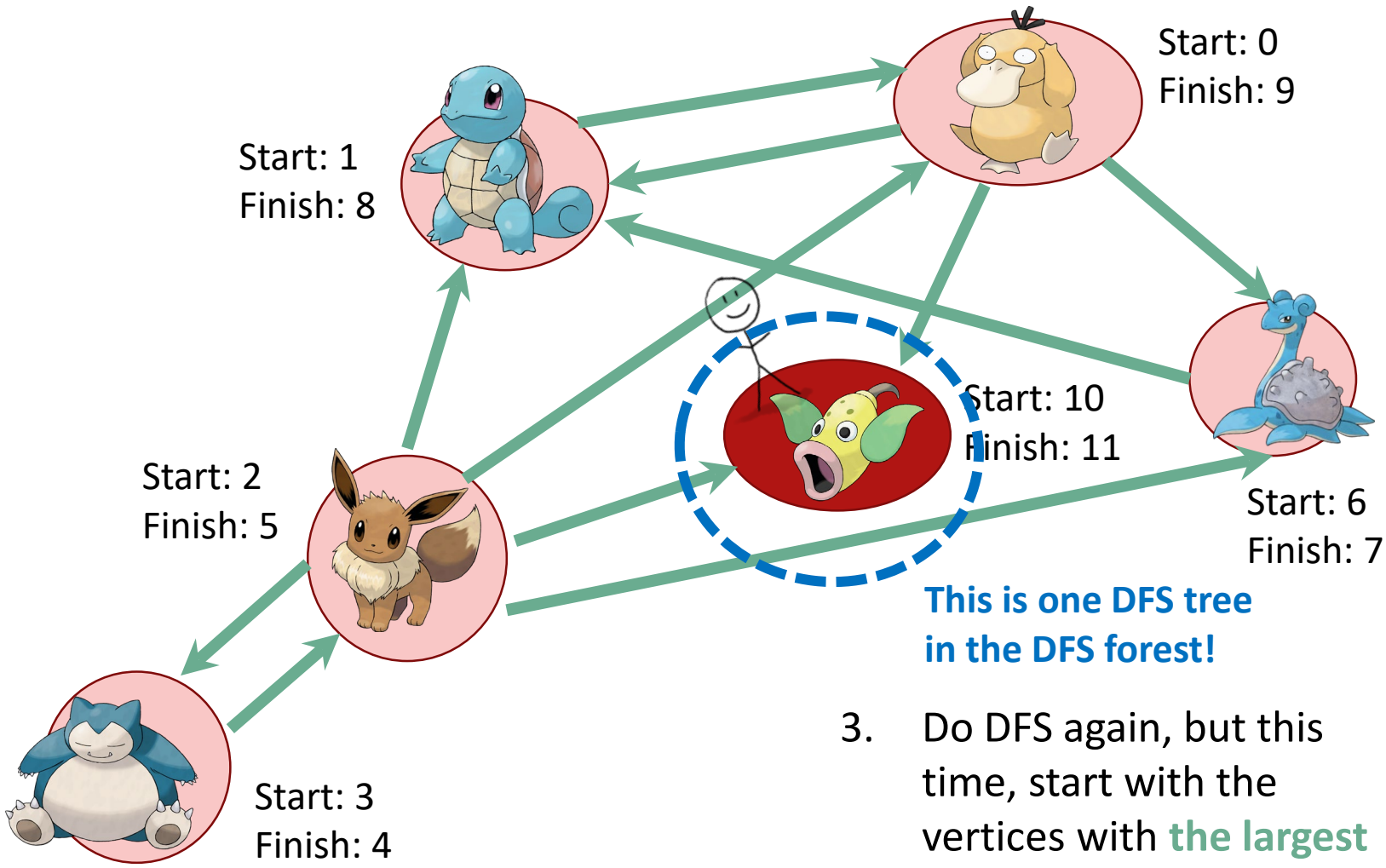
3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example



3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

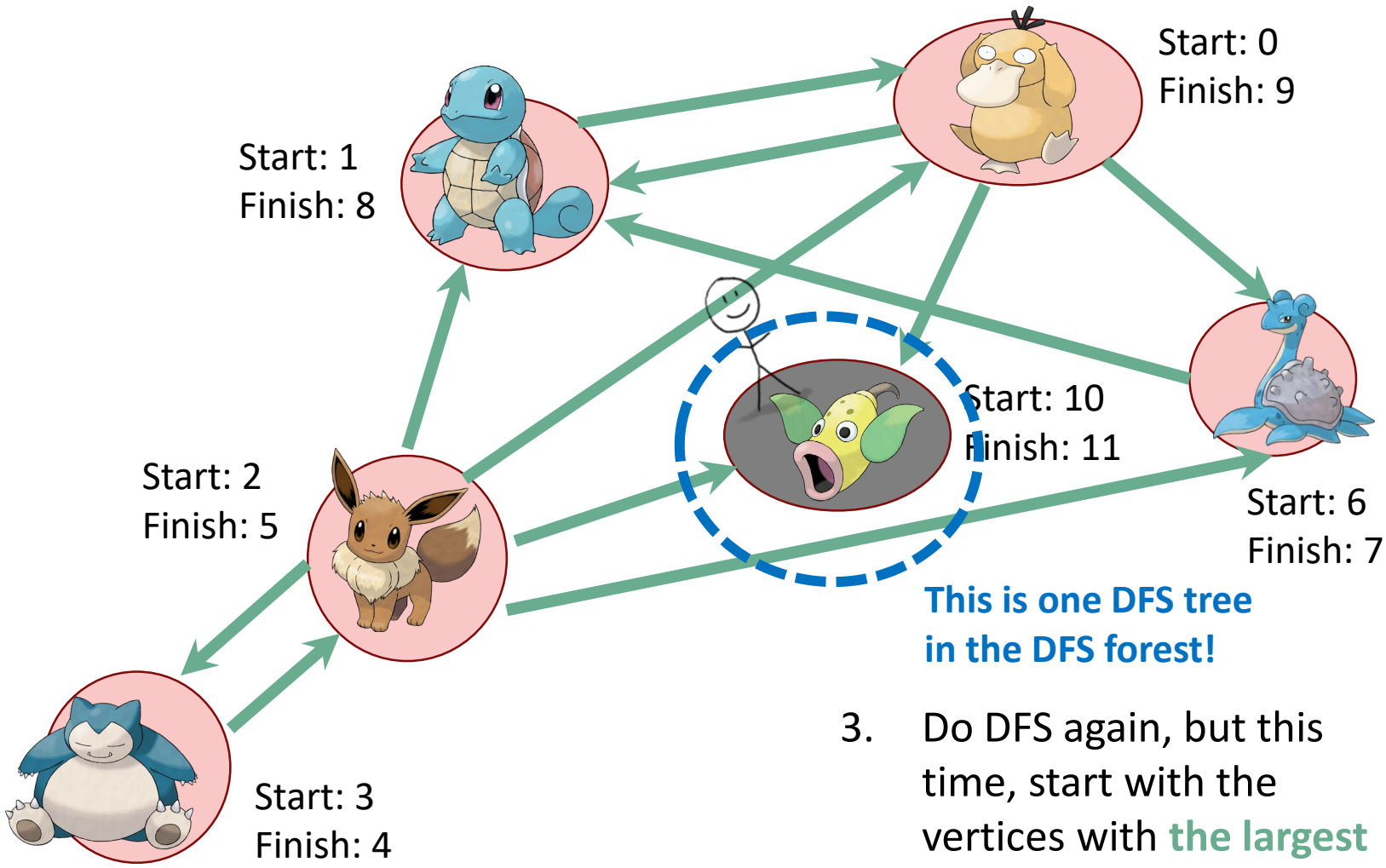
Example



This is one DFS tree in the DFS forest!

3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

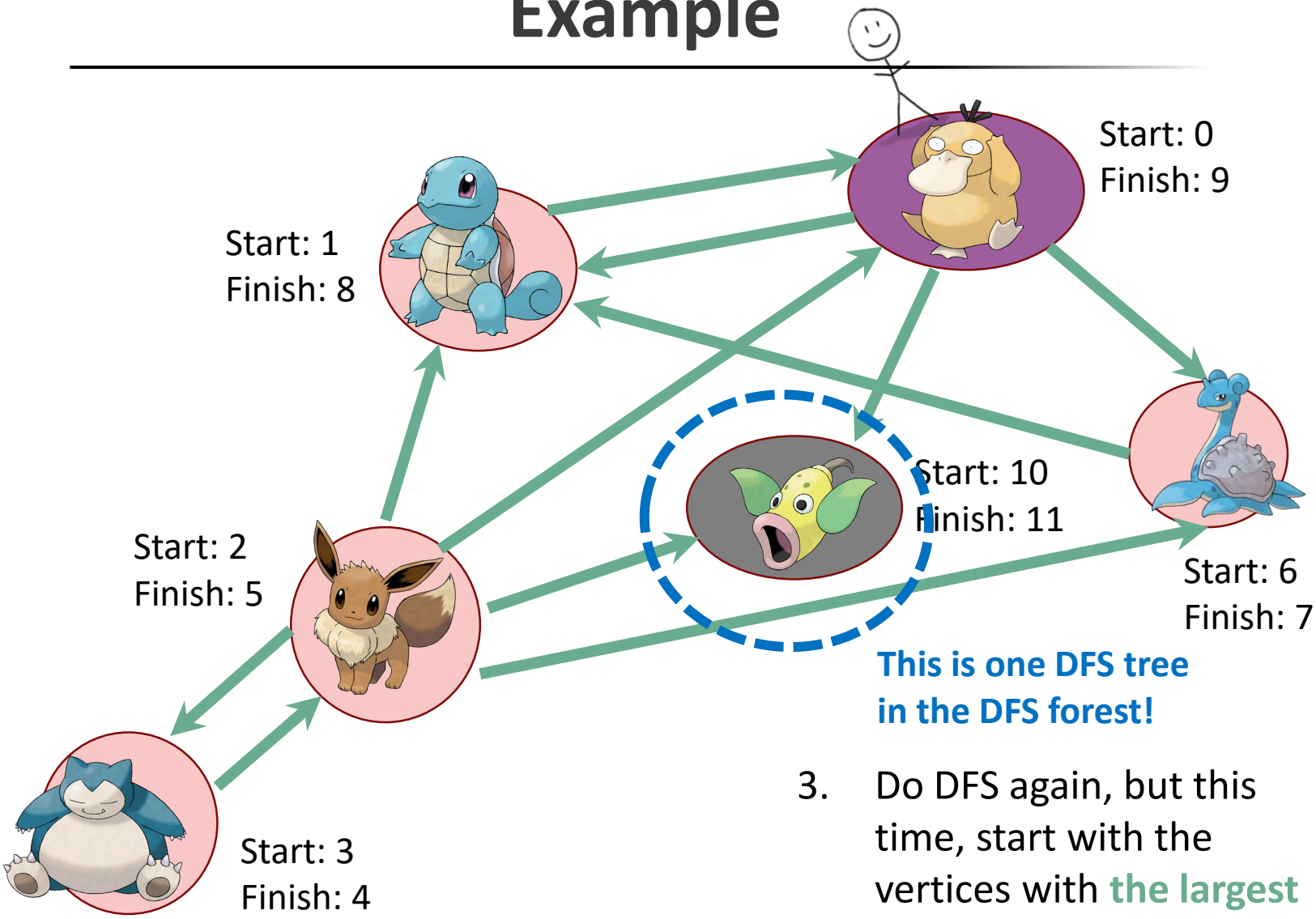
Example



This is one DFS tree in the DFS forest!

3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

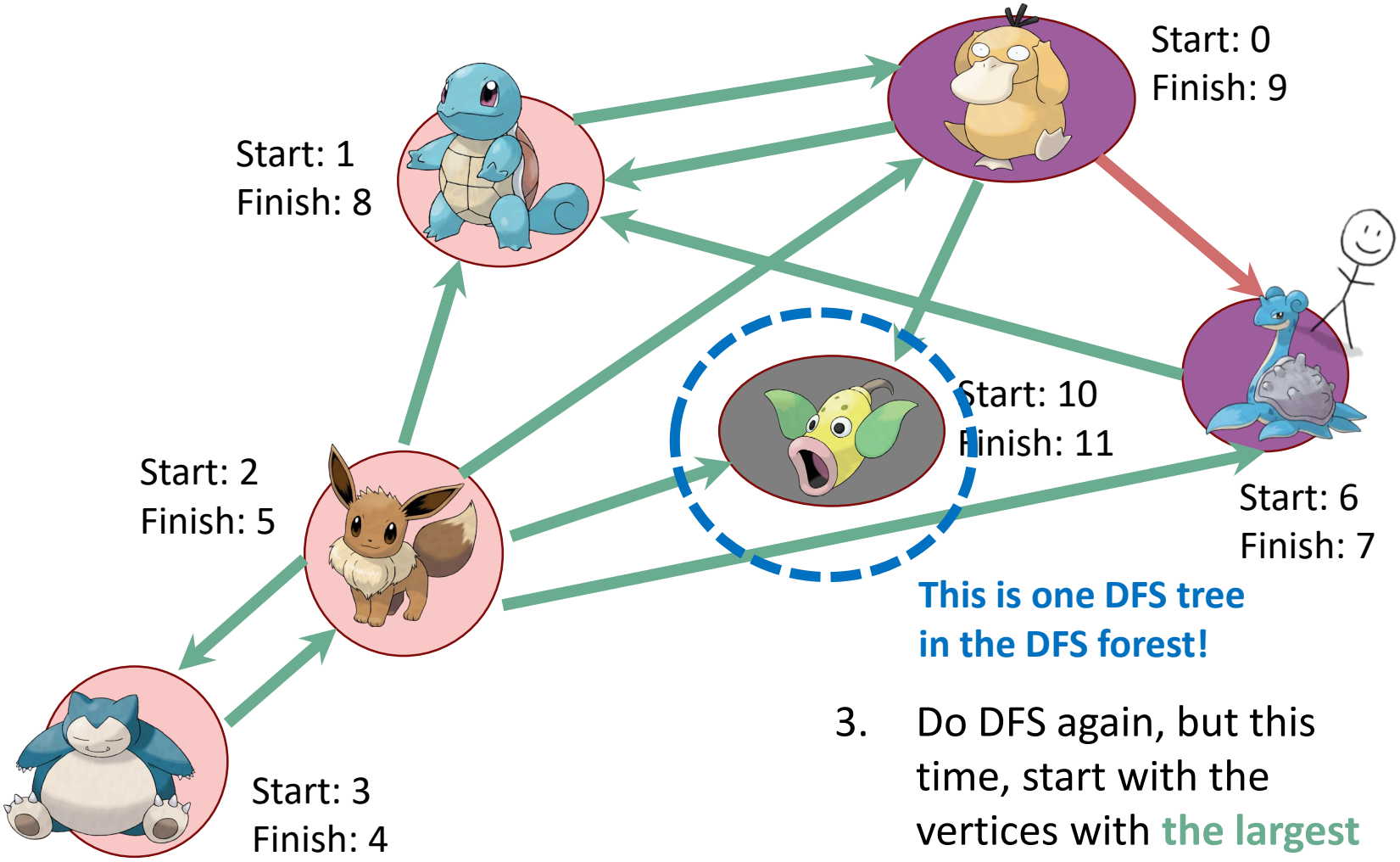


This is one DFS tree in the DFS forest!

3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

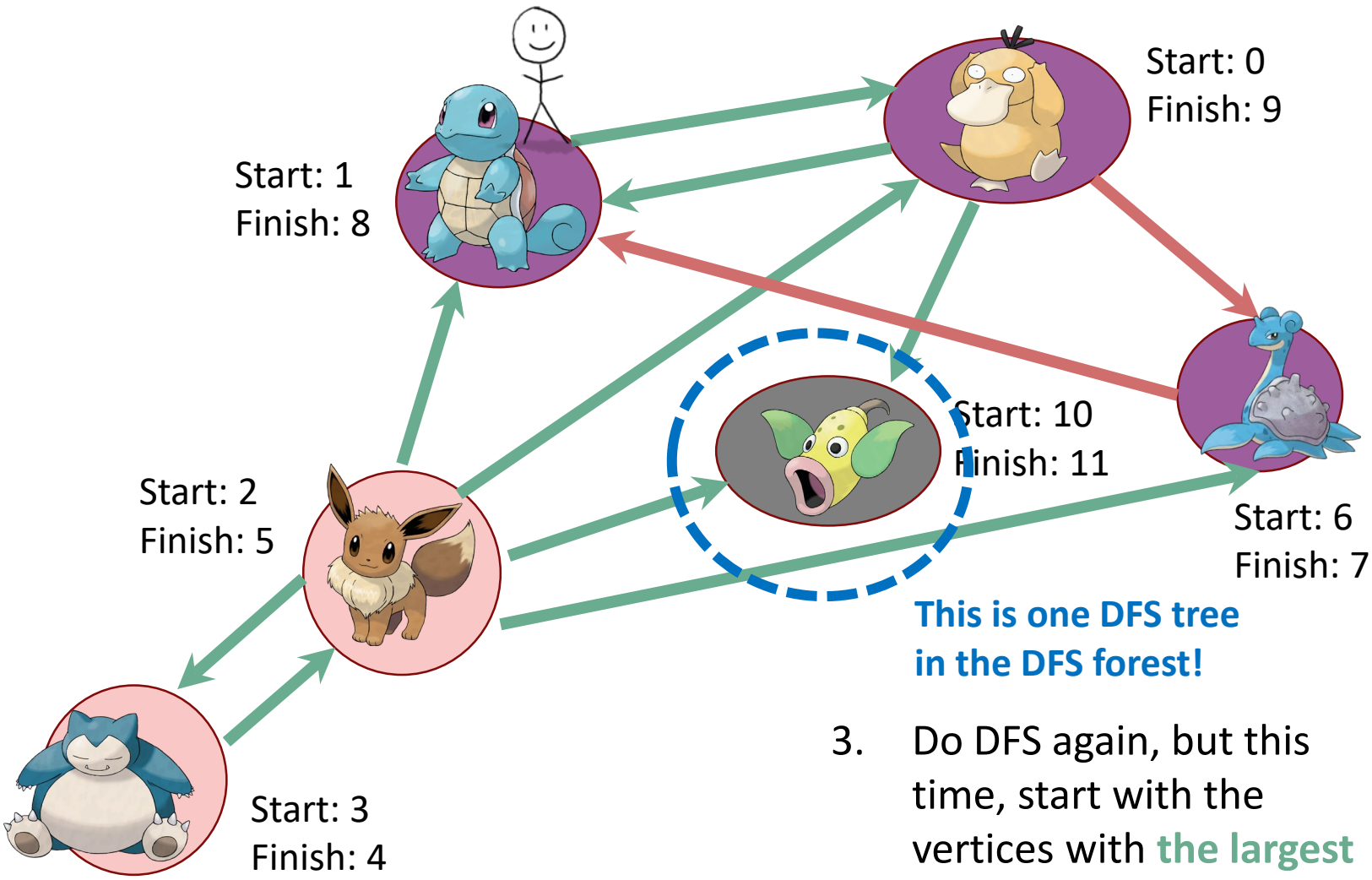


This is one DFS tree in the DFS forest!

3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

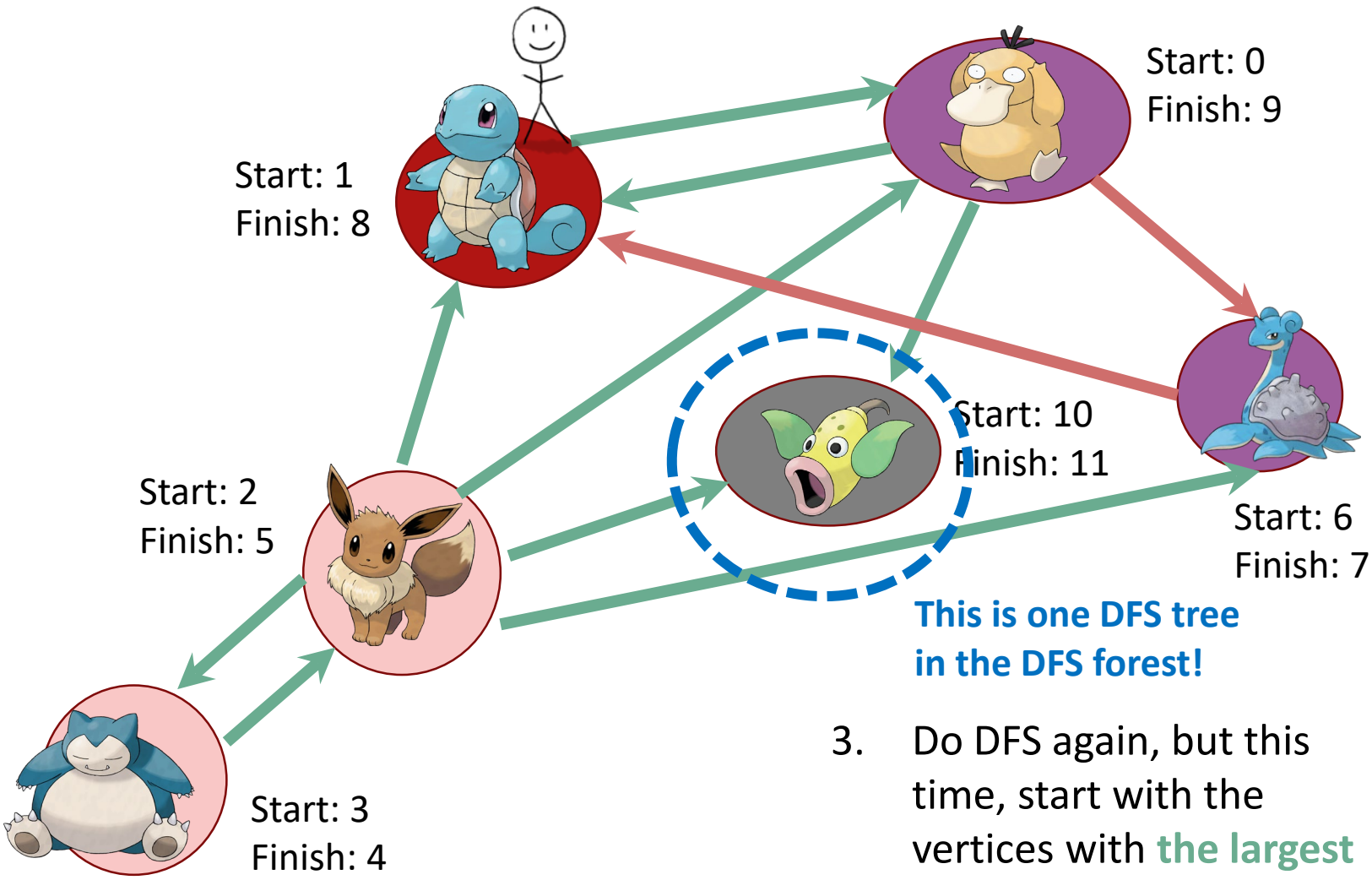


This is one DFS tree in the DFS forest!

3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

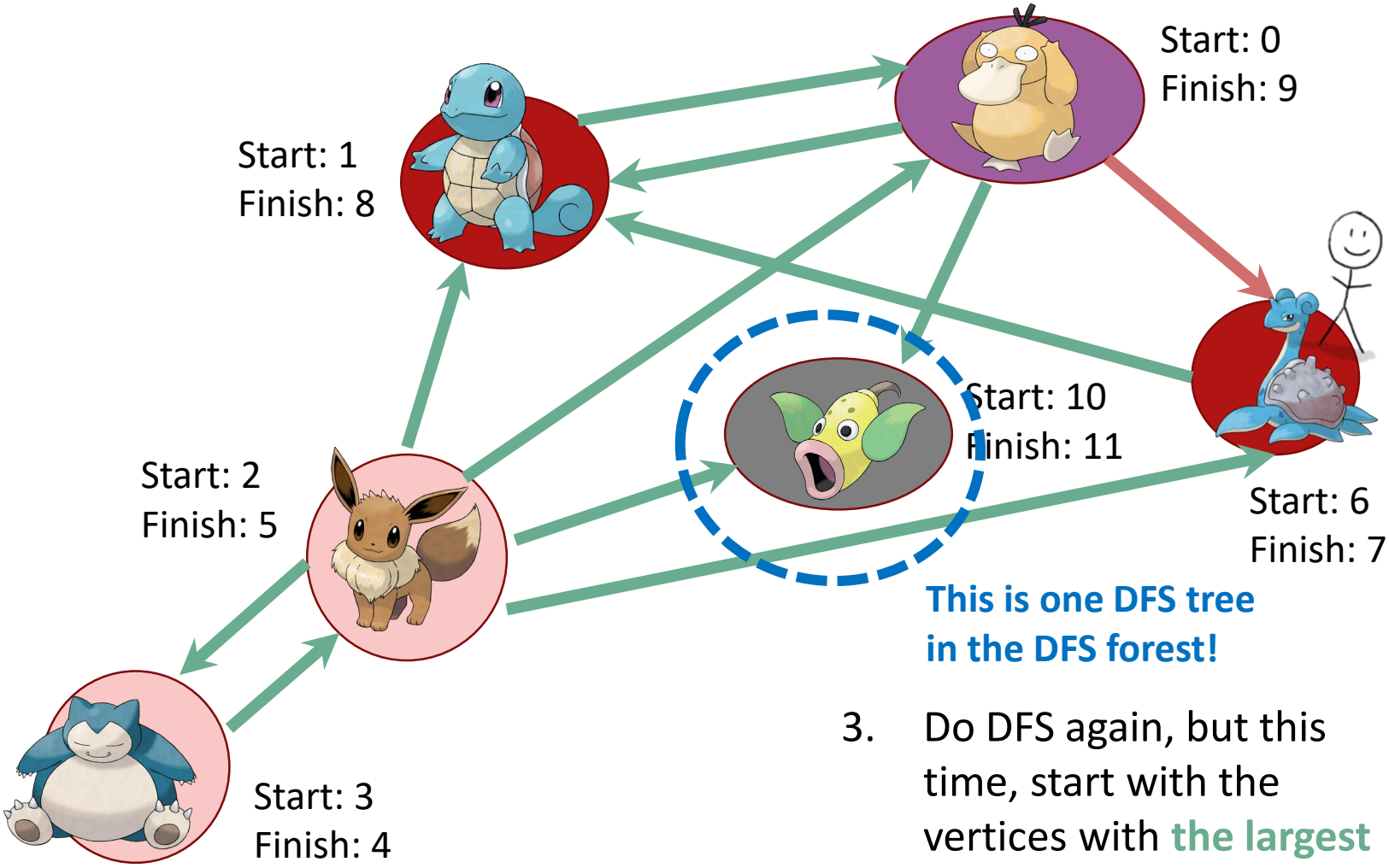


This is one DFS tree in the DFS forest!

3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

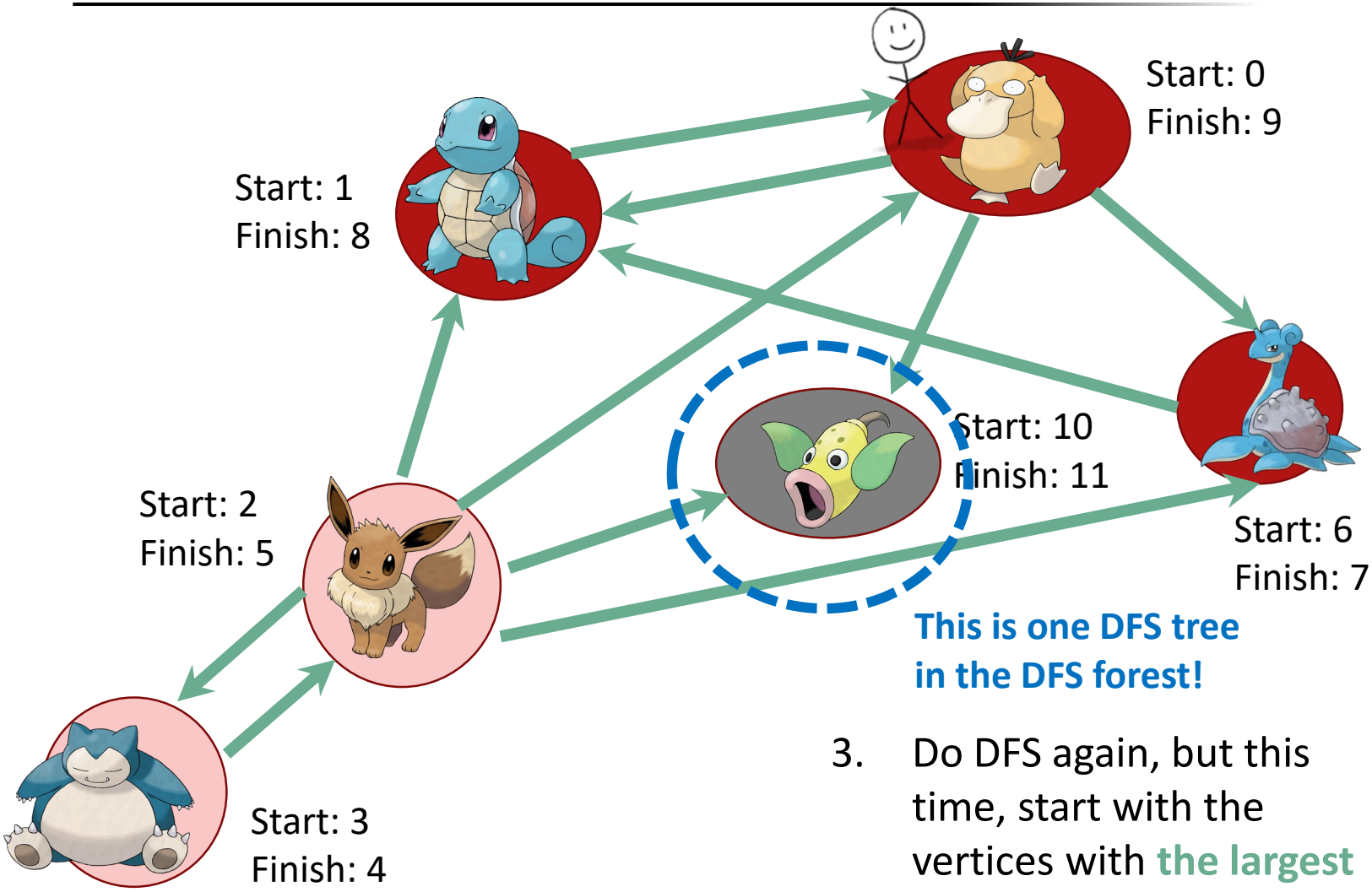


This is one DFS tree in the DFS forest!

3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

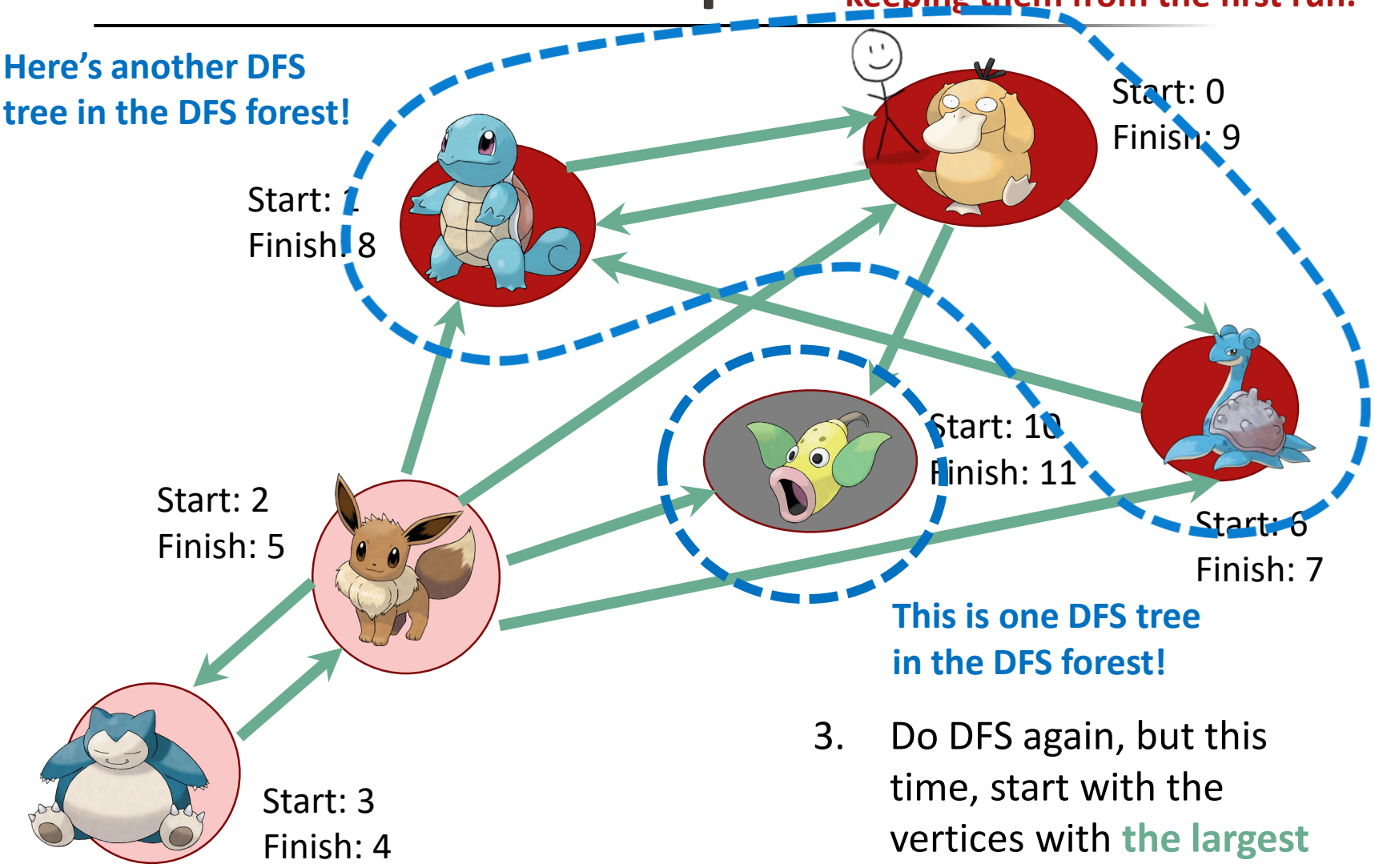


This is one DFS tree in the DFS forest!

3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

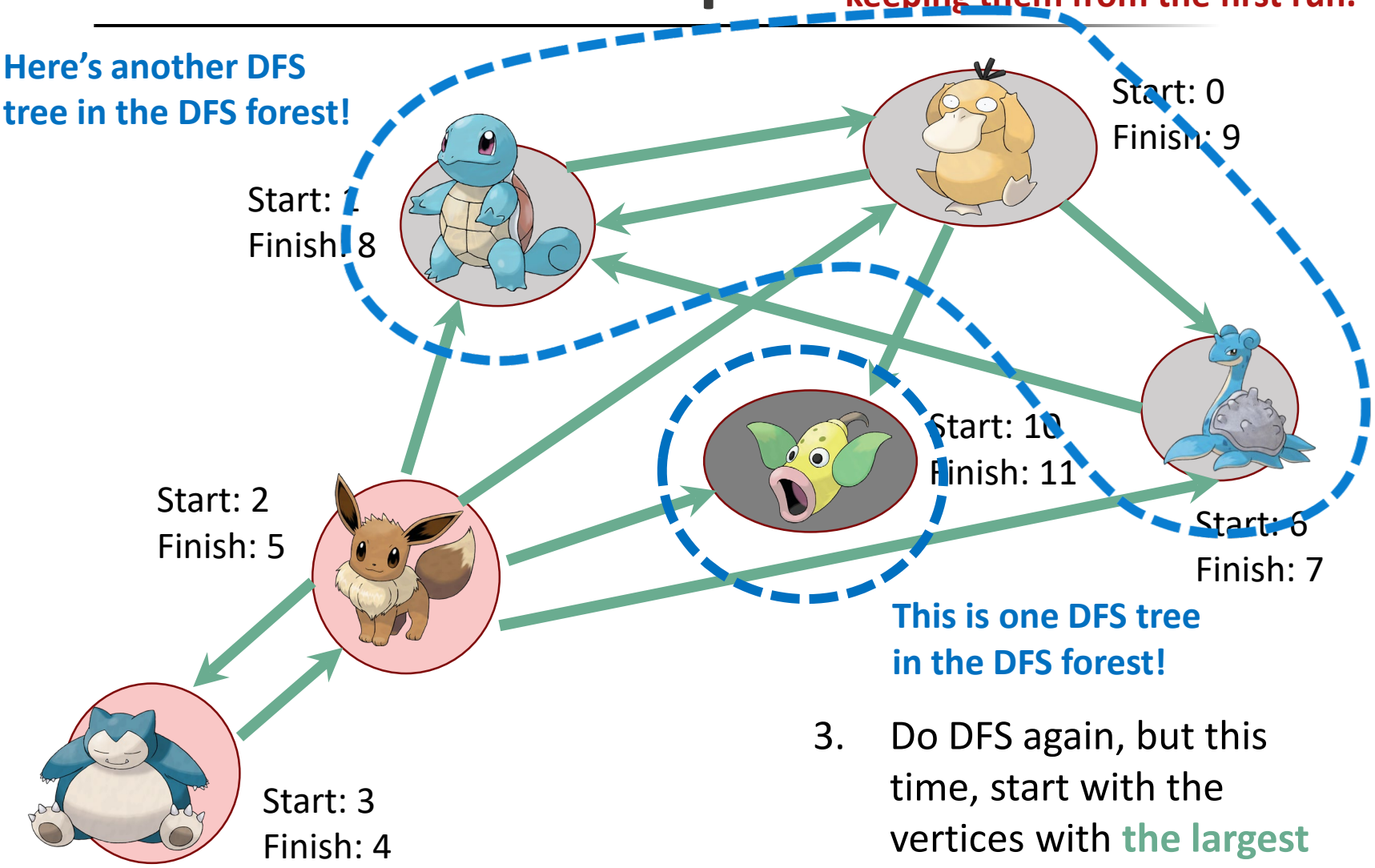
Notice that I'm not changing the start and finish times – I'm keeping them from the first run.



3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

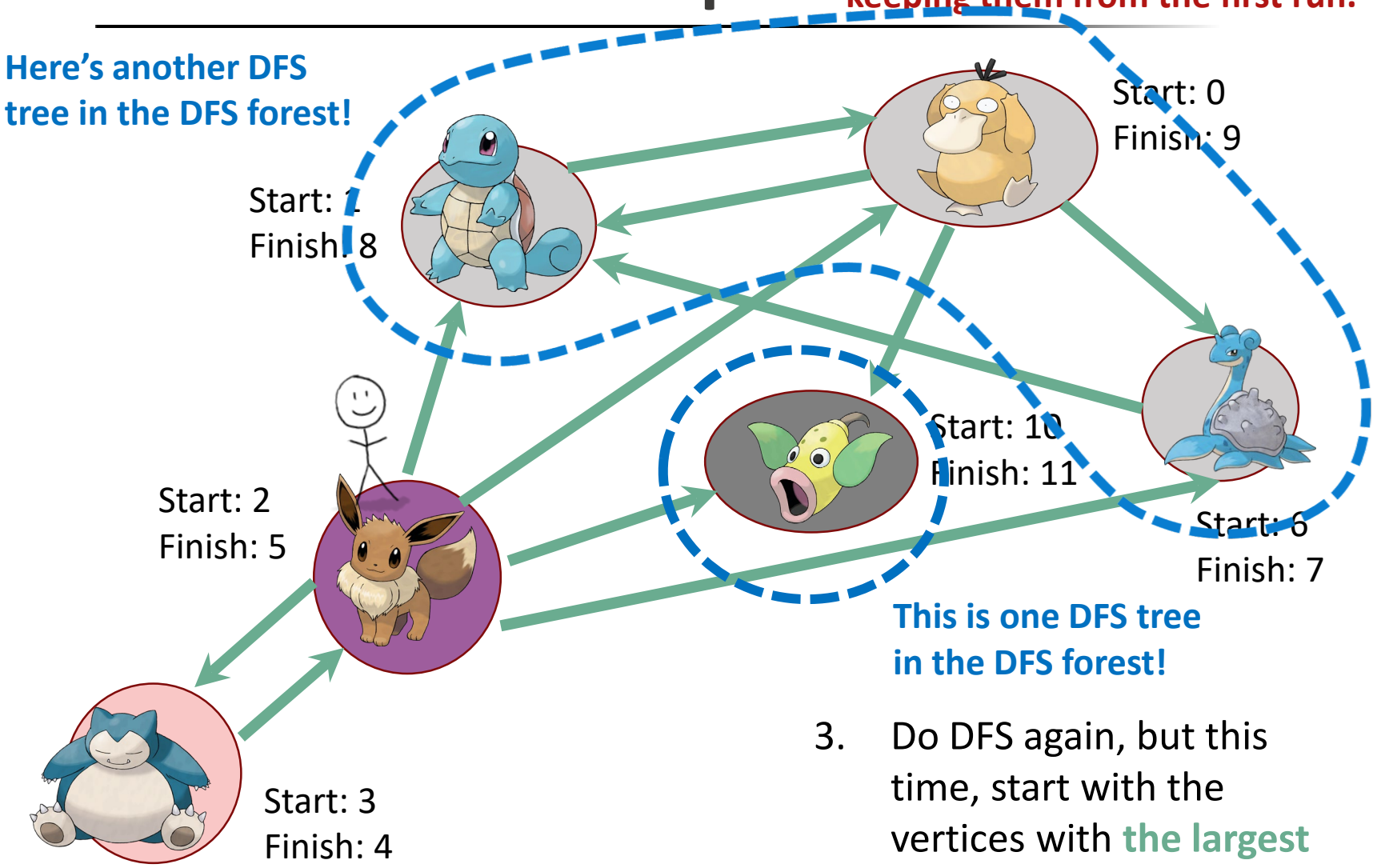
Notice that I'm not changing the start and finish times – I'm keeping them from the first run.



3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

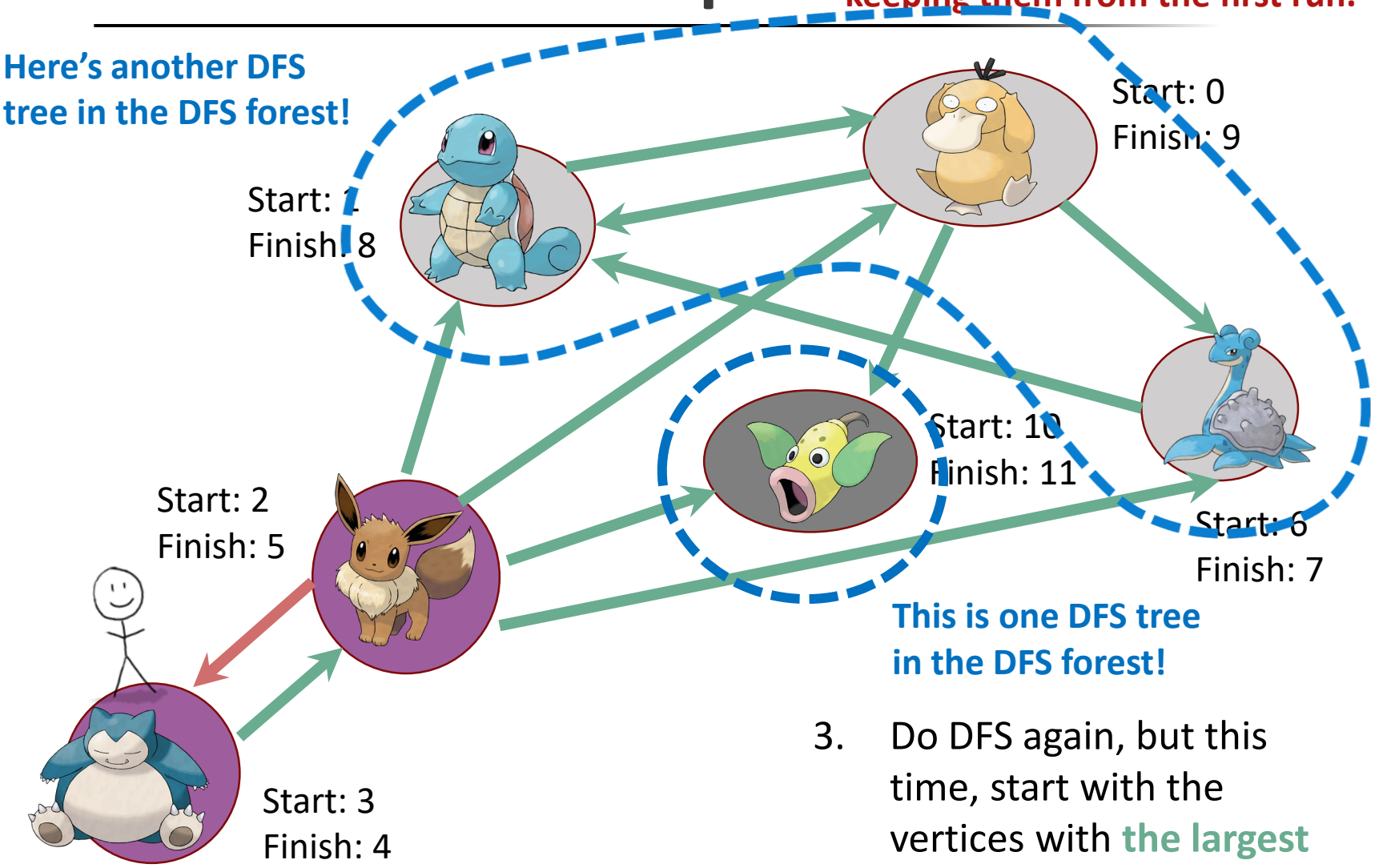
Notice that I'm not changing the start and finish times – I'm keeping them from the first run.



3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

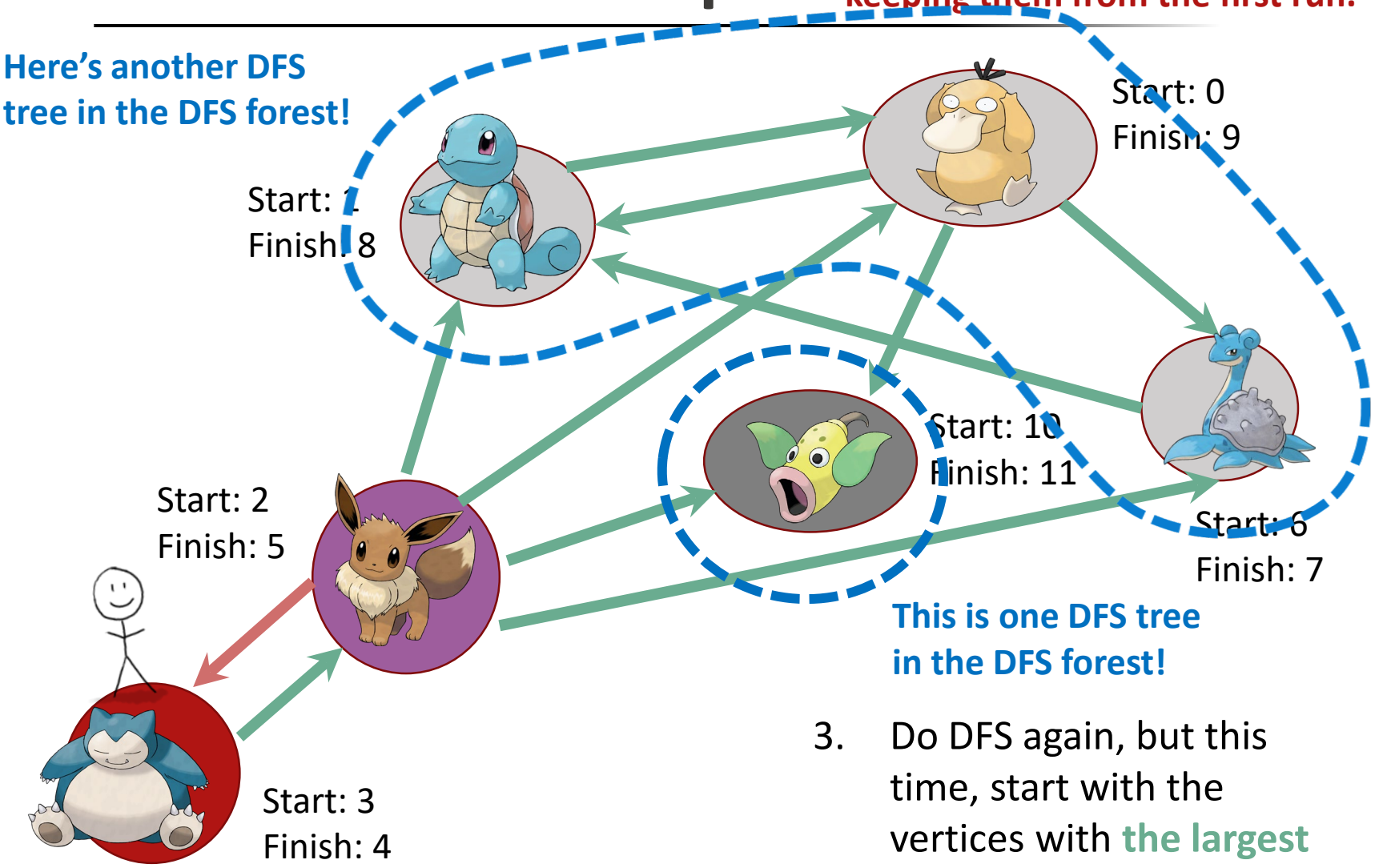
Notice that I'm not changing the start and finish times – I'm keeping them from the first run.



3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

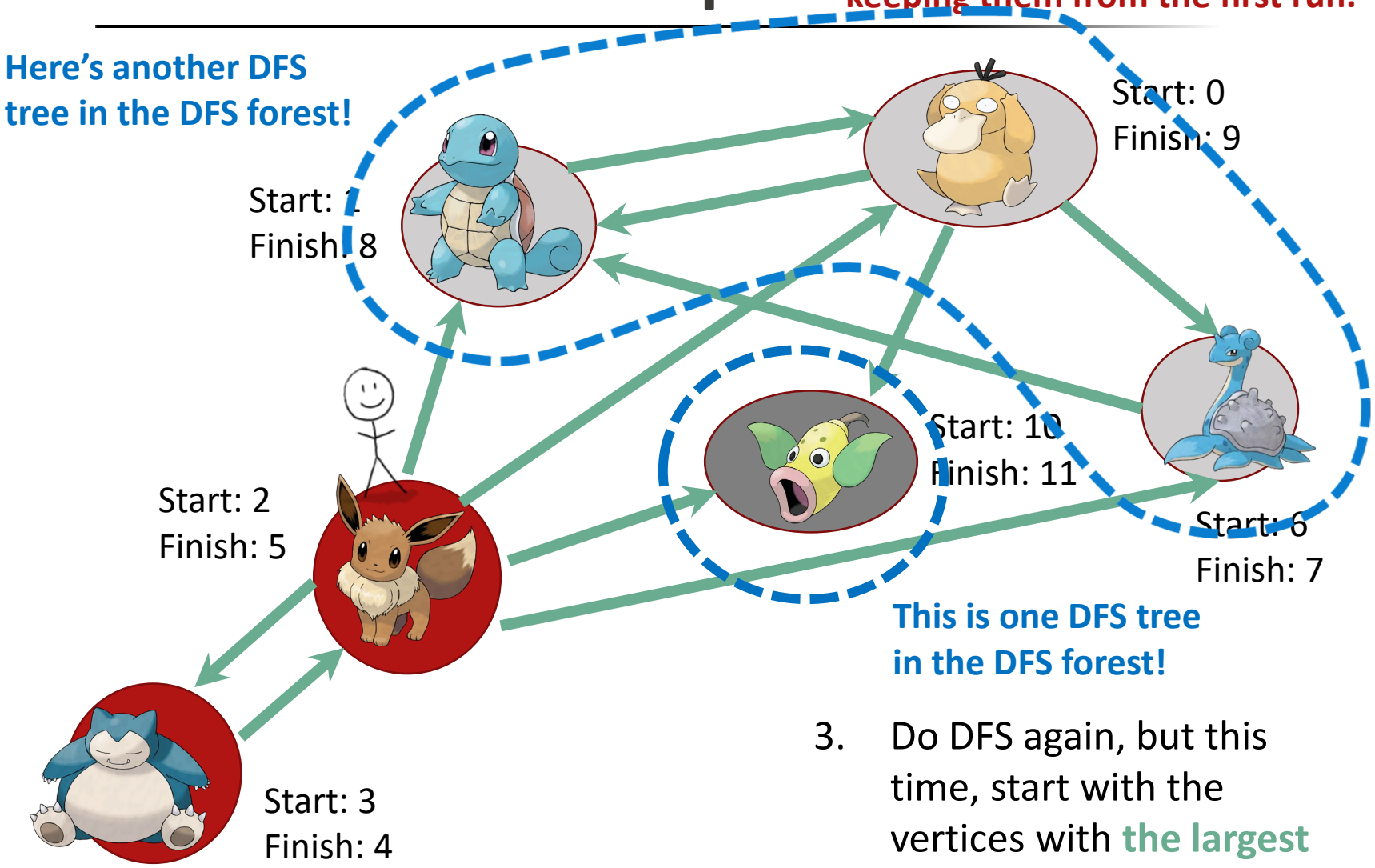
Notice that I'm not changing the start and finish times – I'm keeping them from the first run.



3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

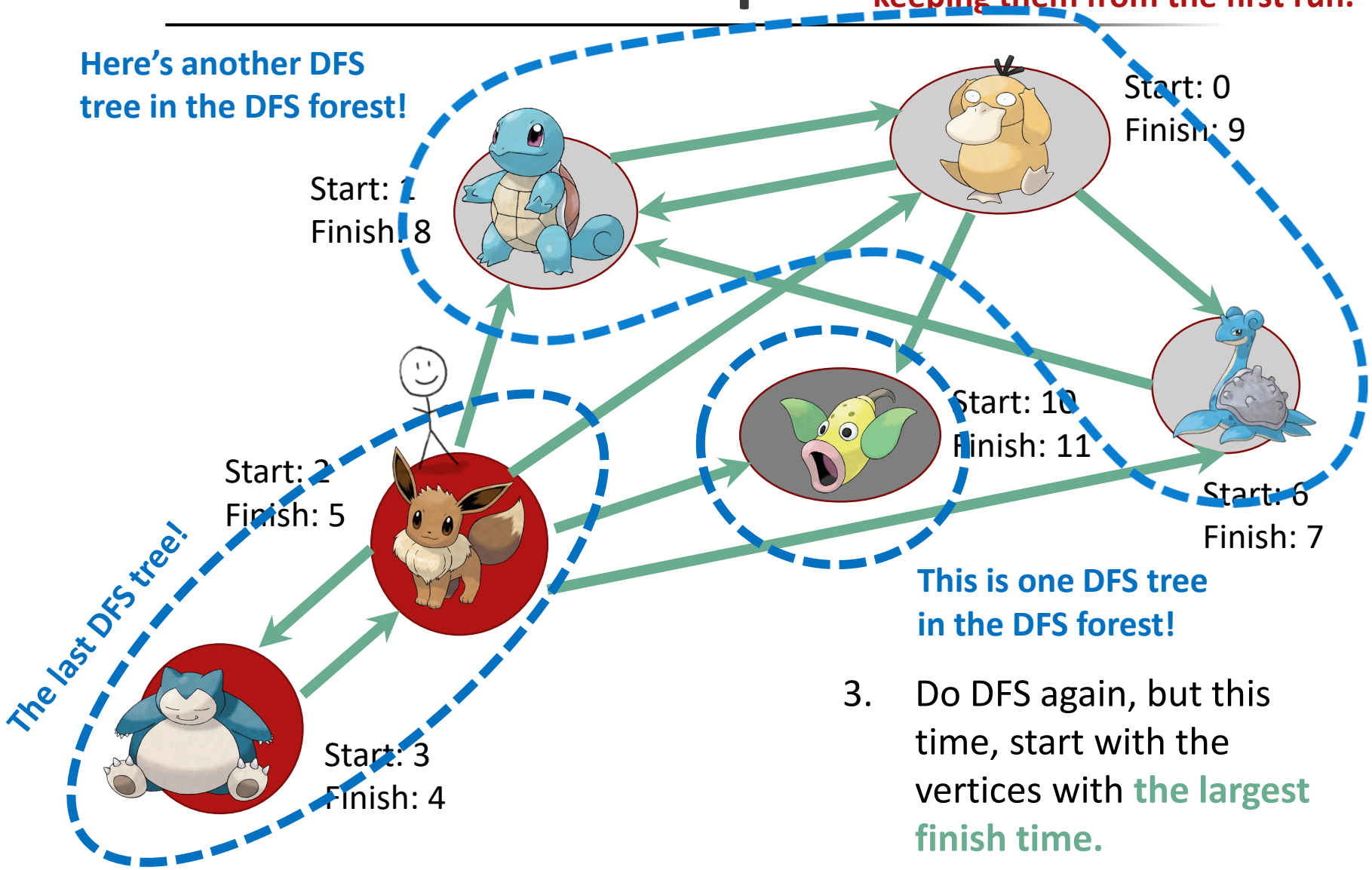
Notice that I'm not changing the start and finish times – I'm keeping them from the first run.



3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

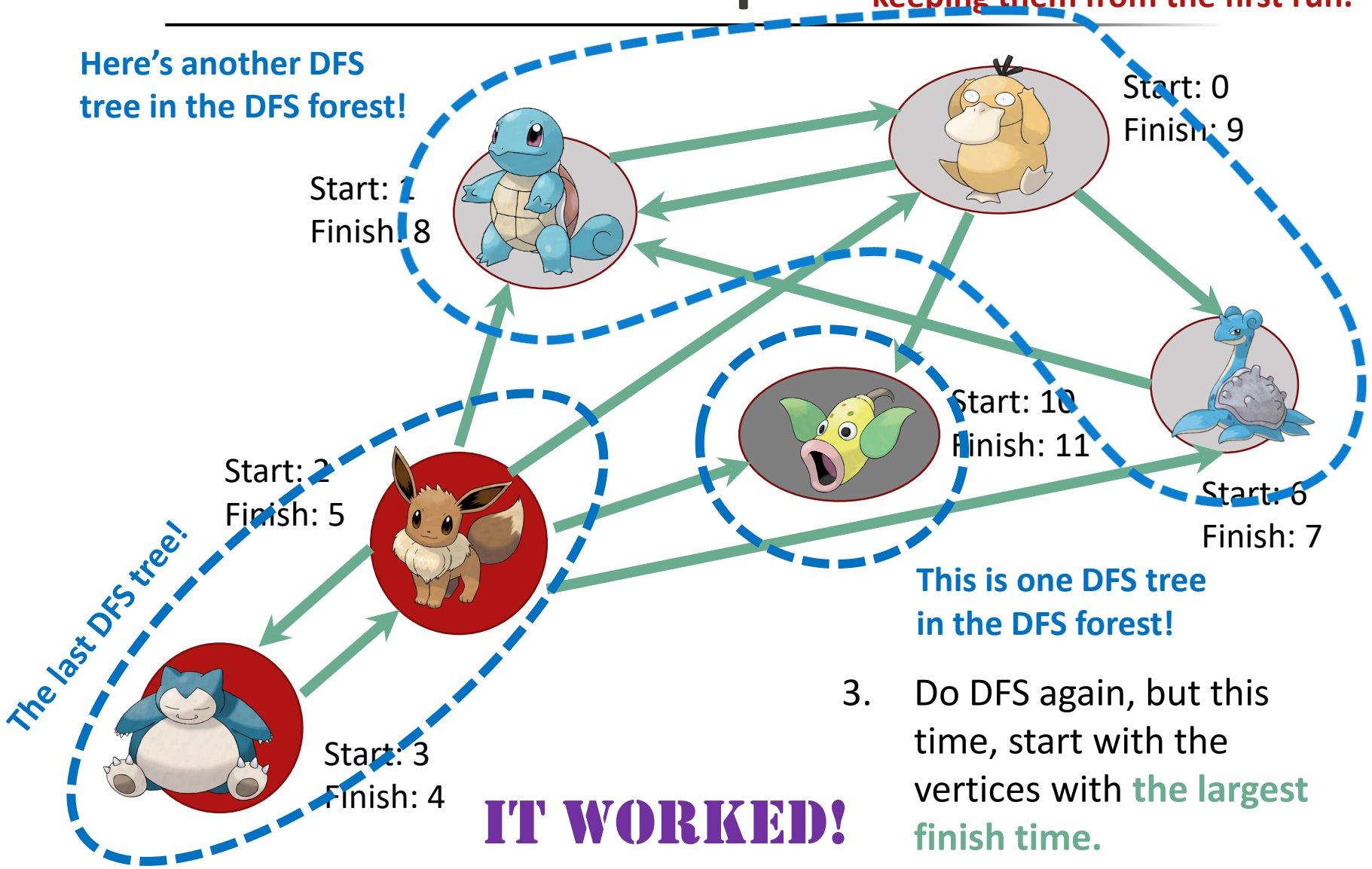
Notice that I'm not changing the start and finish times – I'm keeping them from the first run.



3. Do DFS again, but this time, start with the vertices with **the largest finish time.**

Example

Notice that I'm not changing the start and finish times – I'm keeping them from the first run.

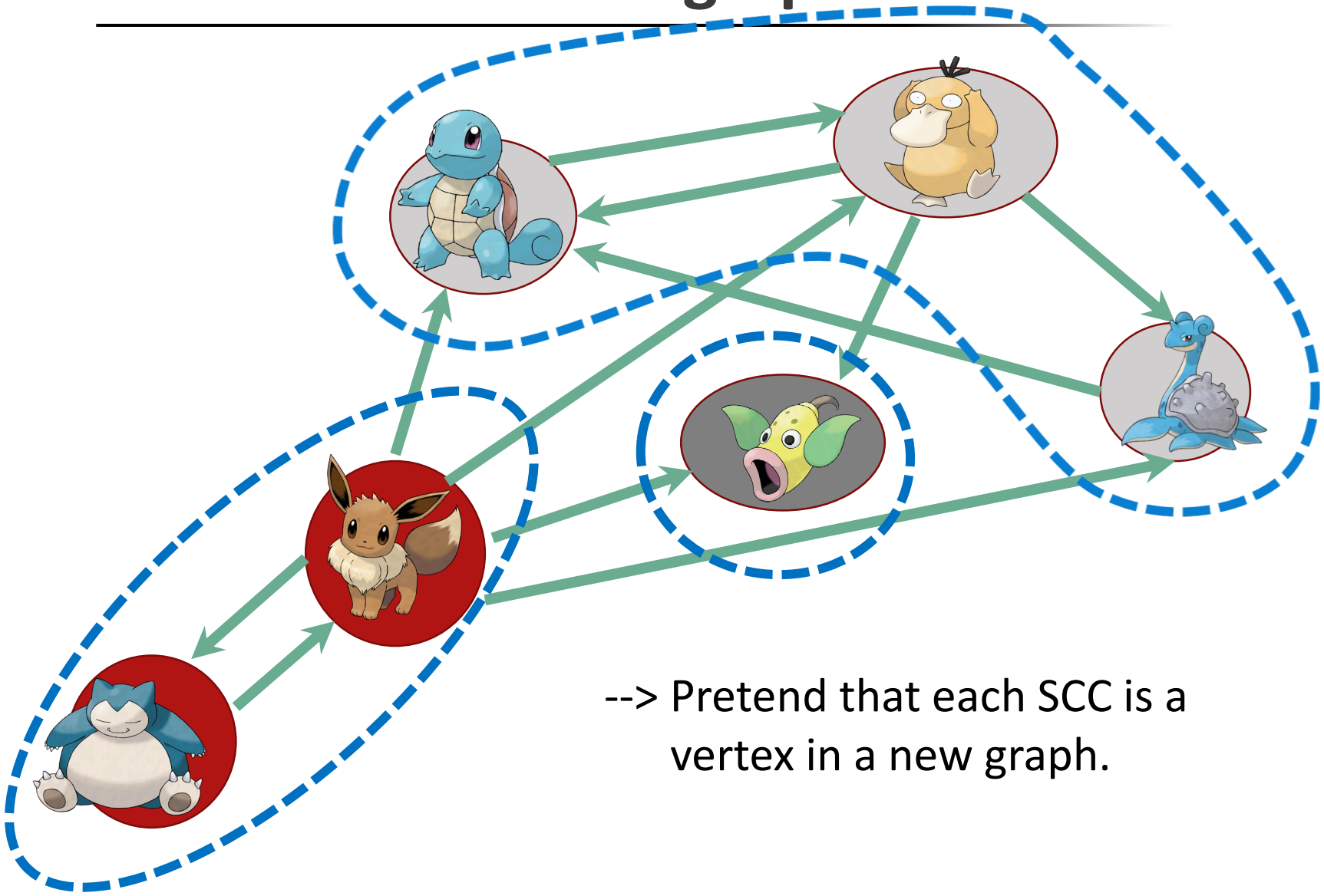


IT WORKED!

3. Do DFS again, but this time, start with the vertices with the largest finish time.

One question

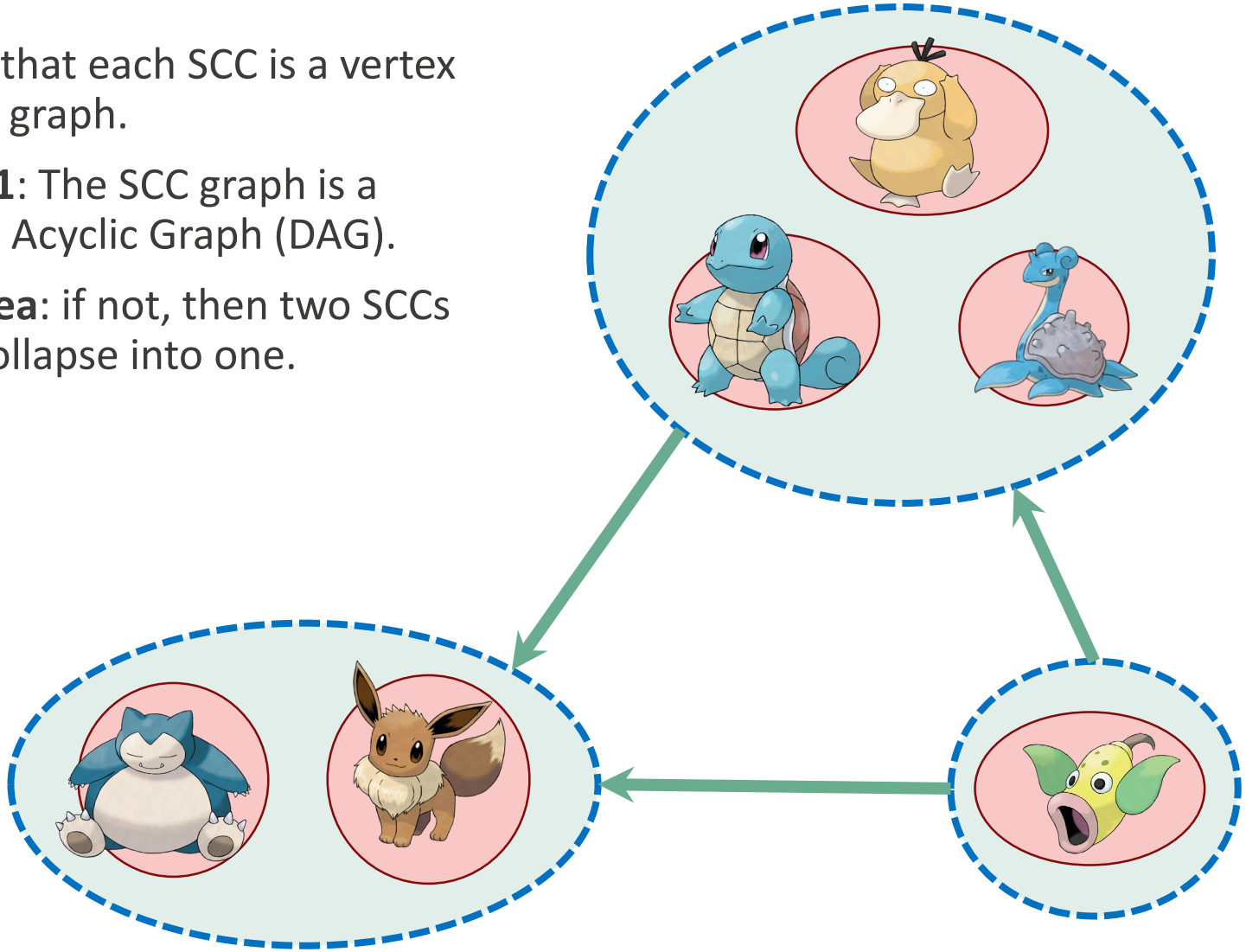
The SCC graph



--> Pretend that each SCC is a vertex in a new graph.

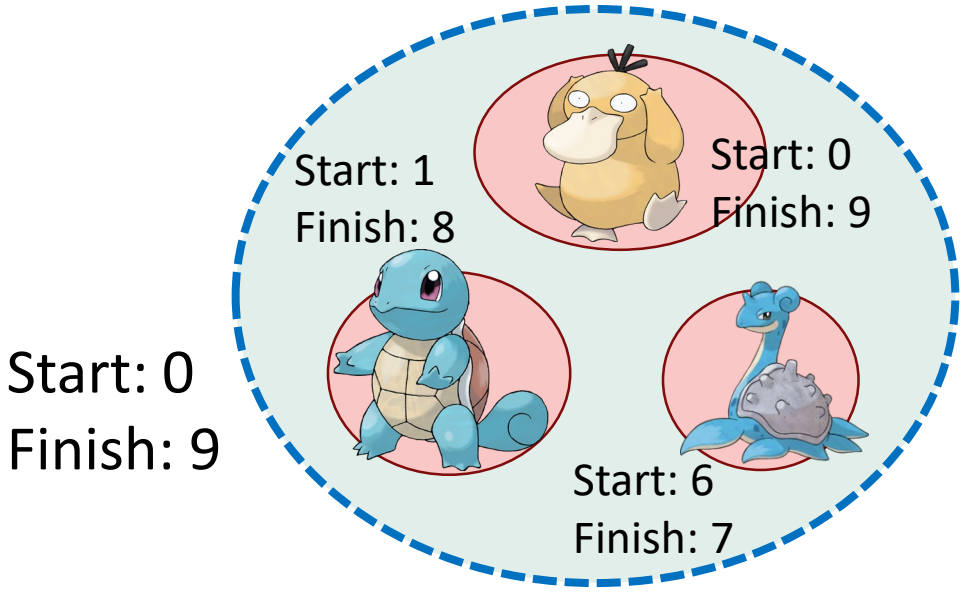
The SCC graph

- Pretend that each SCC is a vertex in a new graph.
- **Lemma 1**: The SCC graph is a Directed Acyclic Graph (DAG).
- **Proof idea**: if not, then two SCCs would collapse into one.



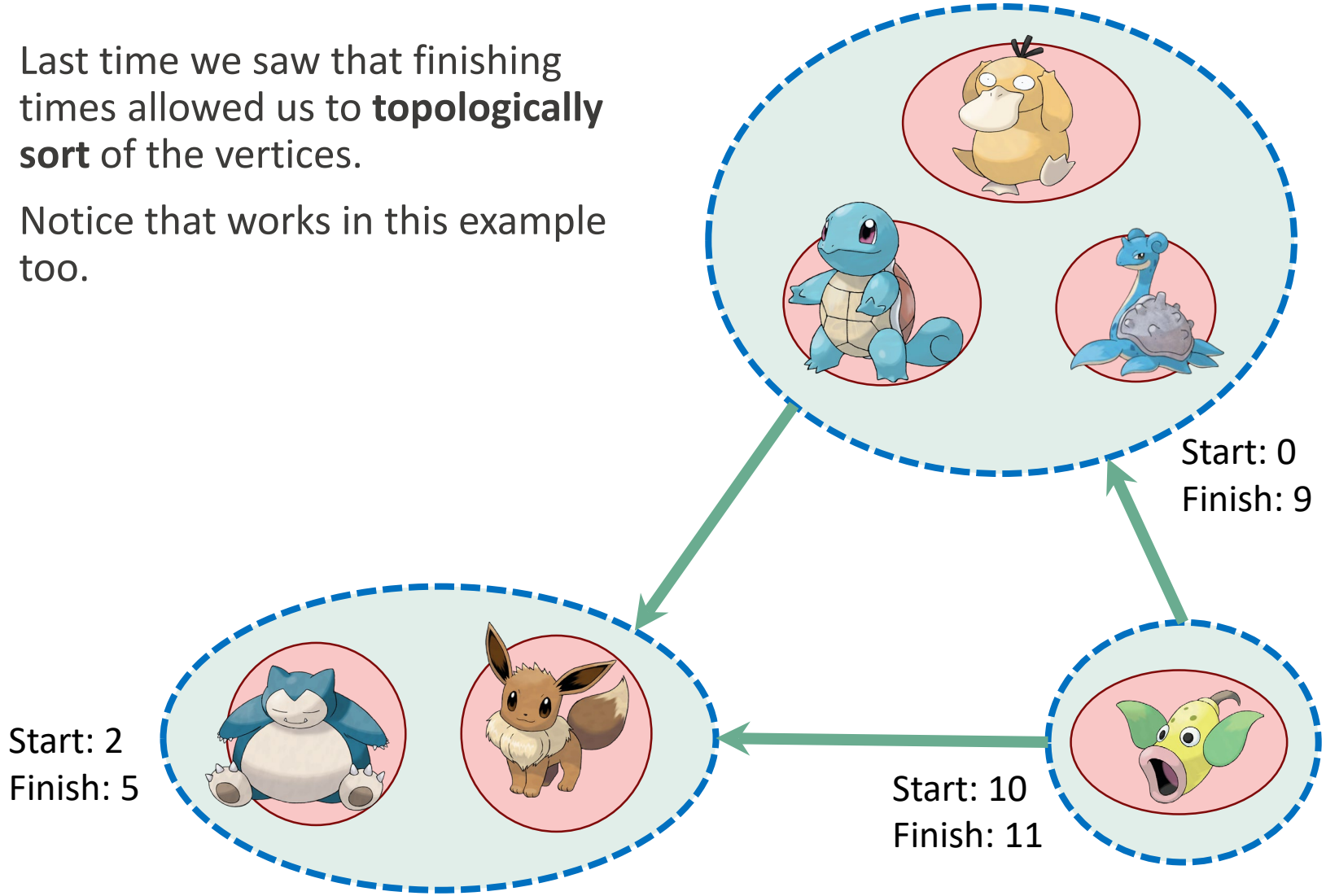
Starting and finishing times in a SCC

- Definition:
  - The **finishing time** of a SCC is the largest finishing time of any element of that SCC.
  - The **starting time** of a SCC is the smallest starting time of any element of that SCC.



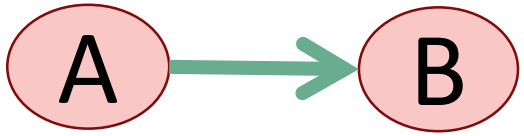
SCC DAG

- Last time we saw that finishing times allowed us to **topologically sort** of the vertices.
- Notice that works in this example too.



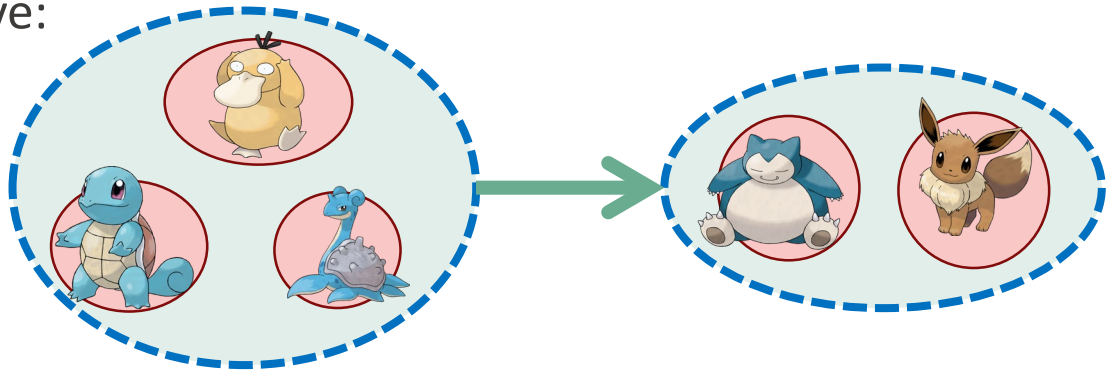
Let's make this idea formal

- As we saw last time,
  - **Claim:** In a DAG, we'll always have:



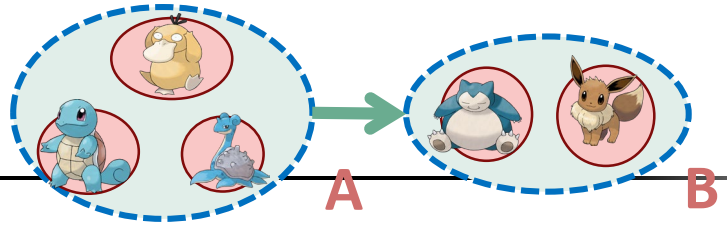
finish: [larger] finish: [smaller]

- Same thing, in the SCC DAG.
  - **Claim:** we'll always have:



finish: [larger] finish: [smaller]

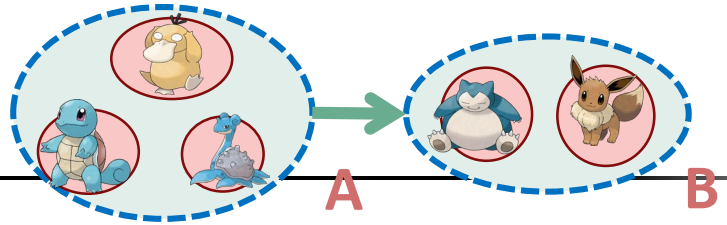
Proof idea



We want to show
A.finish > B.finish.

- **Two cases:**
  - We reached **A** before **B** in our first DFS.
  - We reached **B** before **A** in our first DFS.

Proof idea

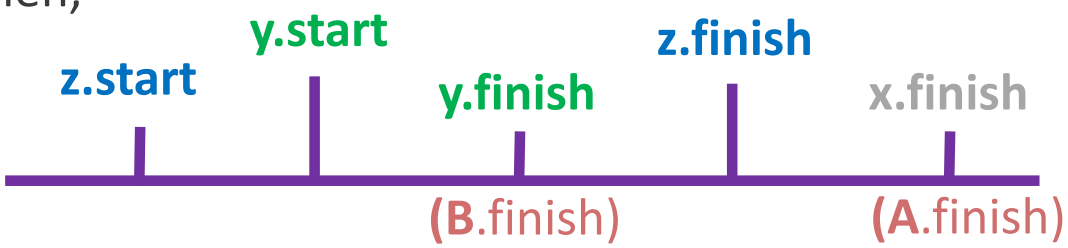


We want to show
A.finish > B.finish.

- **Case 1:** We reached **A** before **B** in our first DFS.
- Say that,
  - **x** has the largest finish in **A**;
  - **y** has the largest finish in **B**;
  - **z** was discovered first in **A**;

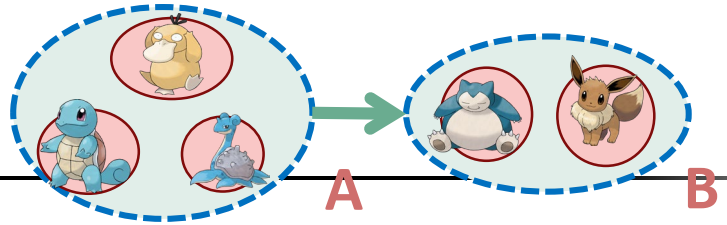
So:
A.finish = **x**.finish
B.finish = **y**.finish
x.finish ≥ **z**.finish

- Then, Reach **A** before **B**
  - => We will discover **y** via **z**
    - (Because we find z for the first time, and A is all connected.)
  - => **y** is a descendant of **z** in the DFS forest.
- Then,



aka,
A.finish > B.finish

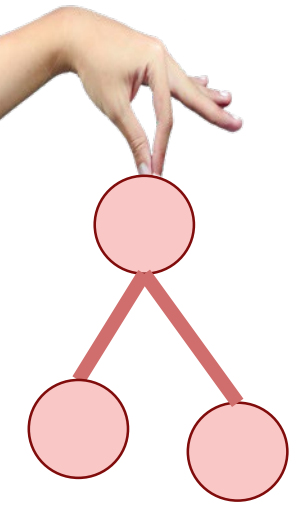
Proof idea



We want to show
A.finish > B.finish.

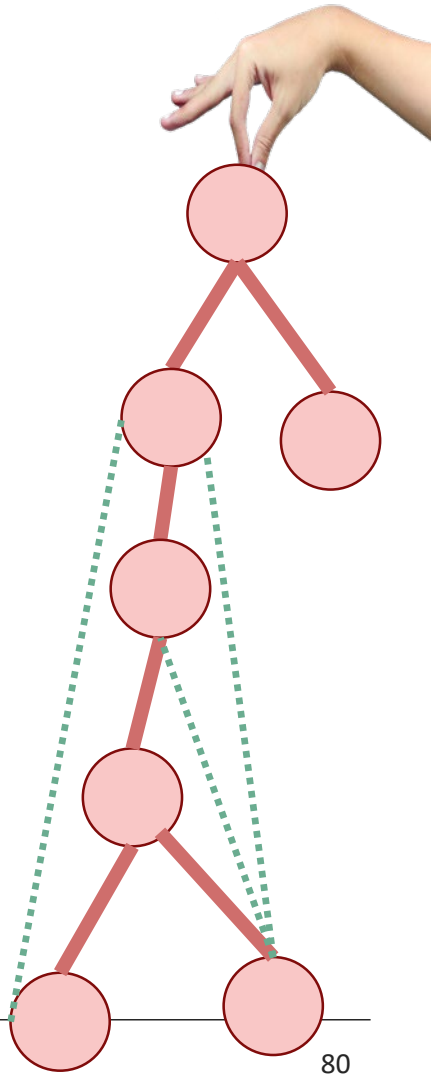
- **Case 2:** We reached **B** before **A** in our first DFS.
  - There are no paths from B to A
    - (because the SCC graph has no cycles)
  - So we completely finish exploring B and never reach A.
  - A is explored later after we restart DFS.

DFS tree

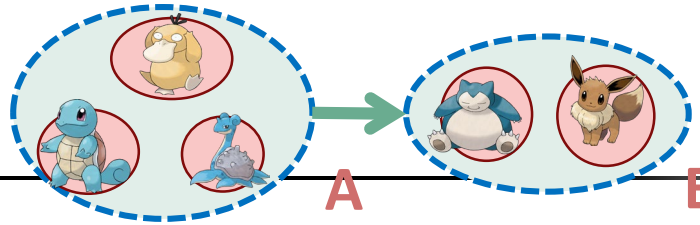


aka,
A.finish > B.finish

DFS tree



Proof



We want to show
A.finish > B.finish.

- Two cases:
  - We reached **A** before **B** in our first DFS.
  - We reached **B** before **A** in our first DFS.

- In either case:

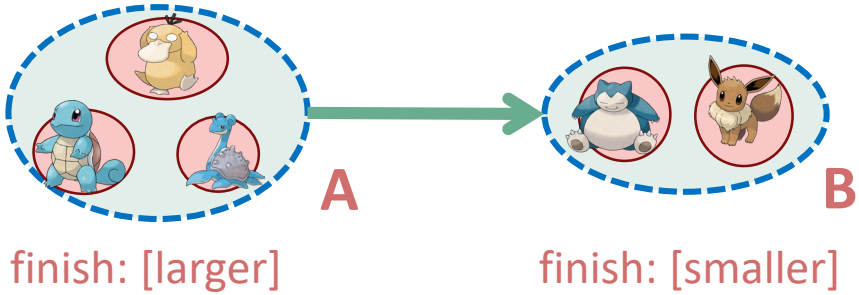
A.finish > B.finish

which is what we wanted to show.

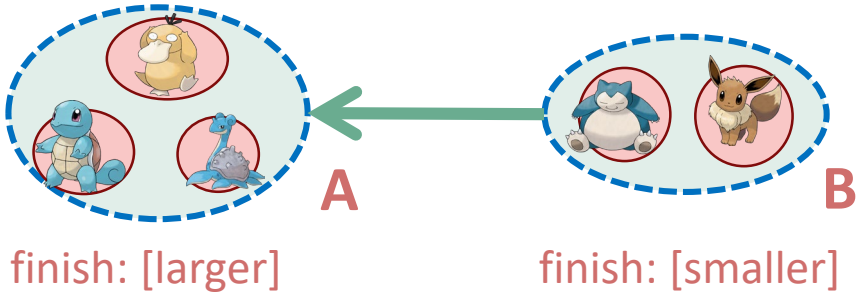
Notice: this is exactly the same two-case argument
that we did earlier, just with the SCC DAG!

This establishes

- **Lemma 2.**
  - If there is an edge like this:
  - Then A.finish > B.finish

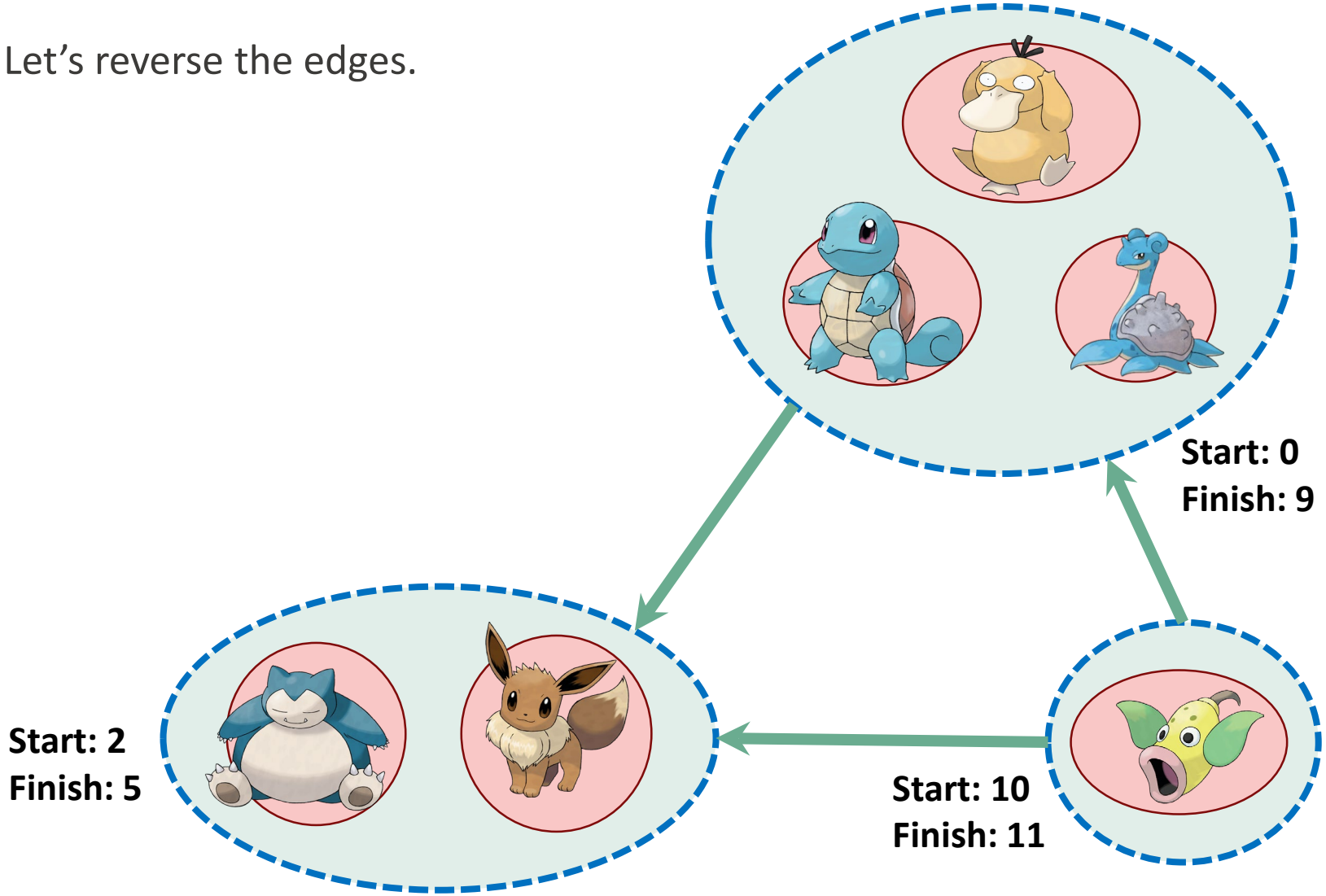


- **Corollary 1.**
  - If there is an edge like this in the **reversed graph**:
  - Then A.finish > B.finish



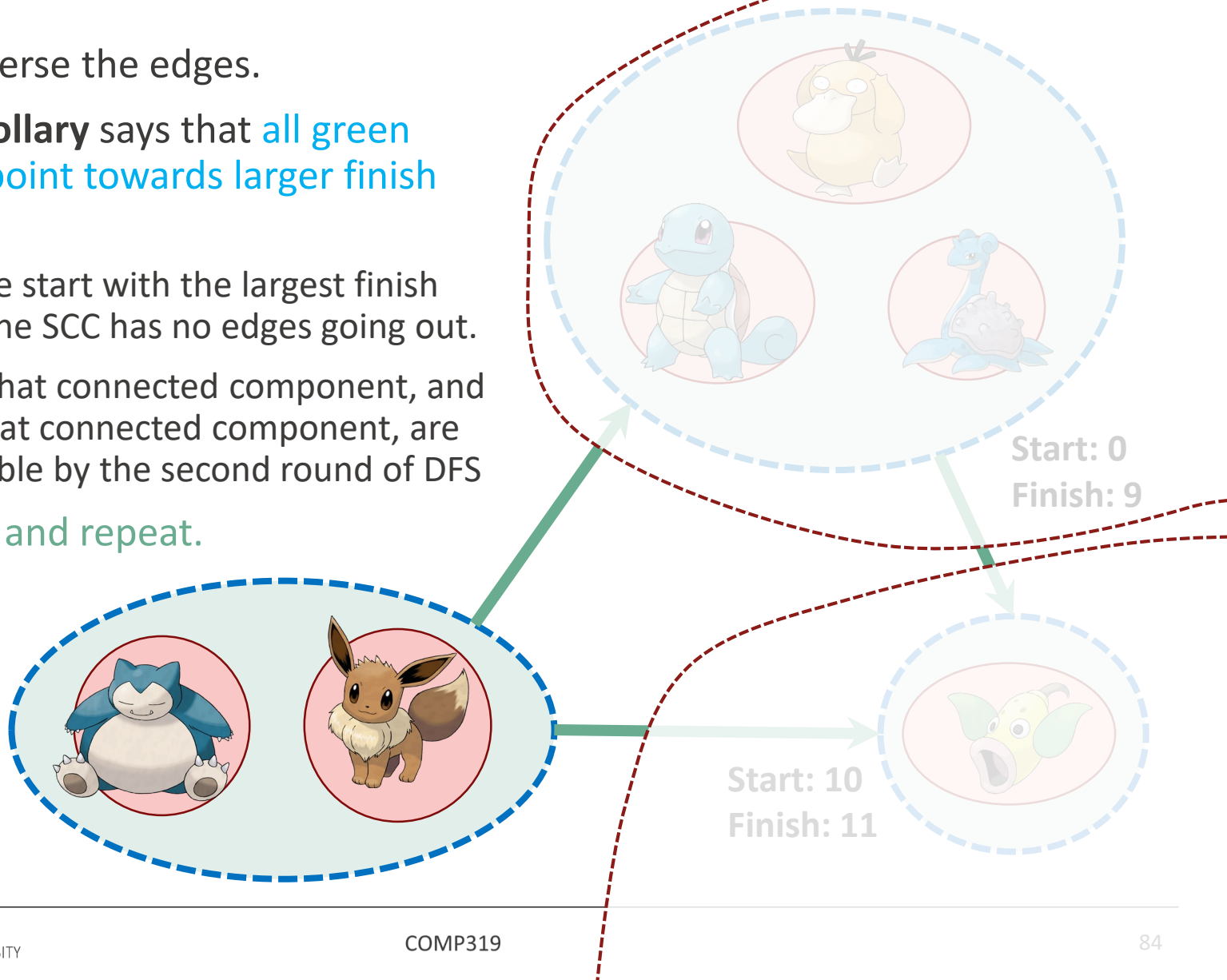
Now we see why this works.

- Let's reverse the edges.



Now we see why this works.

- Let's reverse the edges.
- The **Corollary** says that all green arrows point towards larger finish times.
  - So if we start with the largest finish time, the SCC has no edges going out.
  - Thus, that connected component, and only that connected component, are reachable by the second round of DFS
- Remove and repeat.



Formally, we prove it by induction

- **Theorem:** The algorithm we saw before will correctly identify strongly connected components.
- **Inductive hypothesis:**
  - The first t trees found in the second (reversed) DFS forest are the t SCCs with the largest finish times.
  - Moreover, what's left unvisited after these t trees have been explored is a DAG on the un-found SCCs.
- **Base case:** $(\boldsymbol{t = 0})$
  - The first 0 trees found in the reversed DFS forest are the 0 SCCs with the largest finish times. (TRUE)
  - Moreover, what's left unvisited after 0 trees have been explored is a DAG on all the SCCs. (TRUE by Lemma 1.)

Inductive step

- Assume by induction that the first t trees are the last-finishing SCCs, and the remaining SCCs form a DAG.
- Consider the $(t+1)^{st}$ tree produced, suppose the root is **x**.
  - Suppose that **x** lives in the SCC **A**.
  - Then **A**.finish > **B**.finish for all remaining SCCs **B**.
    - This is because we chose **x** to have the largest finish time.
  - Then there are no edges leaving **A** in the remaining SCC DAG.
    - This follows from the Corollary.
  - Then DFS started at **x** recovers exactly **A**.
    - It doesn't recover any more since nothing else is reachable.
    - It doesn't recover any less since A is strongly connected.
    - (Notice that we are using that A is still strongly connected when we reverse all the edges).
- So the $(t+1)^{st}$ tree is the SCC with the $(t+1)^{st}$ biggest finish time.

Conclusion

- **Theorem:** The algorithm we saw before will correctly identify strongly connected components.
- **Inductive hypothesis:**
  - The first t trees found in the second (reversed) DFS forest are the t SCCs with the largest finish times.
  - Moreover, what's left unvisited after these t trees have been explored is a DAG on the un-found SCCs.
- **Base case:** [done]
- **Inductive step:** [done]
- **Conclusion:**
  - The second (reversed) DFS forest contains all the SCCs as its trees.
  - (This is the first bullet of IH when t = #SCCs)

We can find SCCs in time O(n + m)

- `Algorithm:`
  1. Do DFS.
     - Choose starting vertices in any order.
     - Keep track of finishing times.
  2. Reverse all the edges in the graph.
  3. Do DFS again to create DFS forest.
     - This time, start vertices in the reverse order of the finishing times that they had from the first DFS run.
  4. The SCCs are the different trees in the second DFS forest.

Recall: DFS takes O(n + m) time.

- **Runtime?**
  - We run DFS twice, and do nothing much.
  - So, finding SCCs takes O(n + m) time.

Recap

- **Depth First Search** reveals a very useful structure.
  - We saw last week that this structure can be used to do Topological Sorting in time O(n + m)
  - Today we saw that it can also find Strongly Connected Components in time O(n + m)

- Next time:
  - BFS and Dijkstra's algorithm!


Any Question?