UCLA CS35L

Wednesday

Rewriting History

git commit --amend can change the last commit content and commit message.

Just git add or remove the appropriate changes and call **git commit --amend**

Then rewrite the commit message and save/exit the editor dialog box.

Warning: Only rewrite the history when the changes do not affect commit that have been pushed to the remote.

https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History

Rewriting History

In order to rewrite the last few commits, use the git interactive rebase, e.g.

```
git rebase -i HEAD~3
```

will rebase the last 3 commits onto the 4th oldest commit.

You will see an editor open up with texts like the ones in the next slide.

Beware the the commits are listed in the reverse order to that of git log.

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
```

Three Trees

HEAD
Last commit snapshot, next parent. HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch.

Index
Proposed next commit snapshot. The index is your proposed next commit. We've also been referring to this concept as Git's "Staging Area" as this is what Git looks at when you run git commit.

Working Directory
The actual directories with files that you can modify with an editor.

https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified

git reset

If HEAD refers to the master branch,

and there are three commits on the master branch **eb43bf8**, **9e5e6a4**, **38eb946**,

with **38eb946** being the most recent,

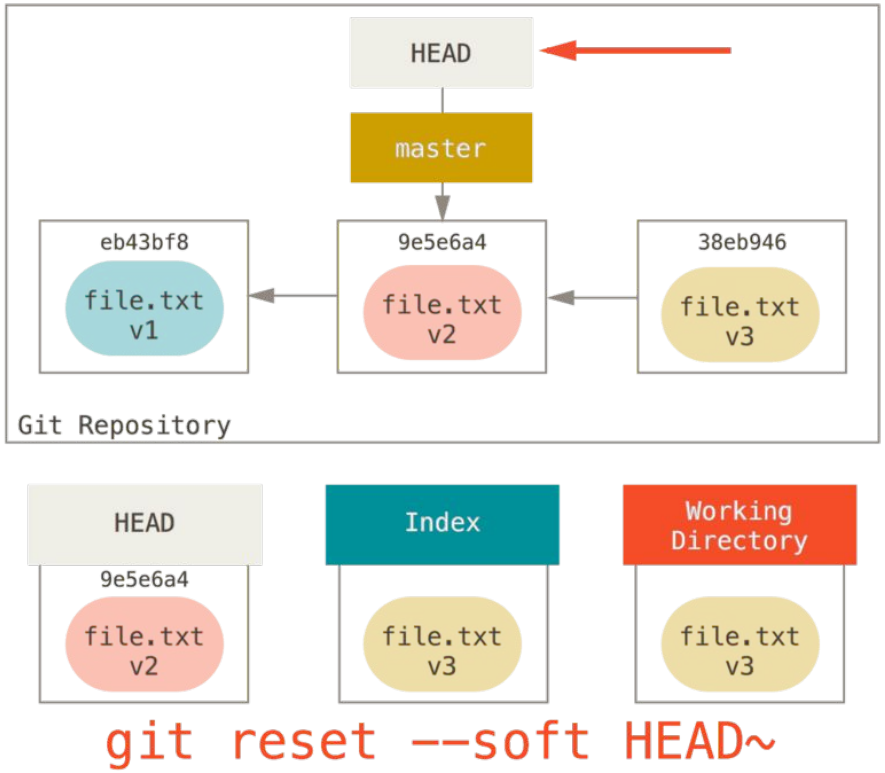
then **`git reset 9e5e6a4`** will perform the following steps.

Step 1 (Move HEAD)

It will make the master point to the commit **9e5e6a4**.

Note that if **`git reset --soft 9e5e6a4`** is used instead, git reset will stop at this step.

At this point, git status will show the changes file.txt v3 as staged, since the index didn't change.



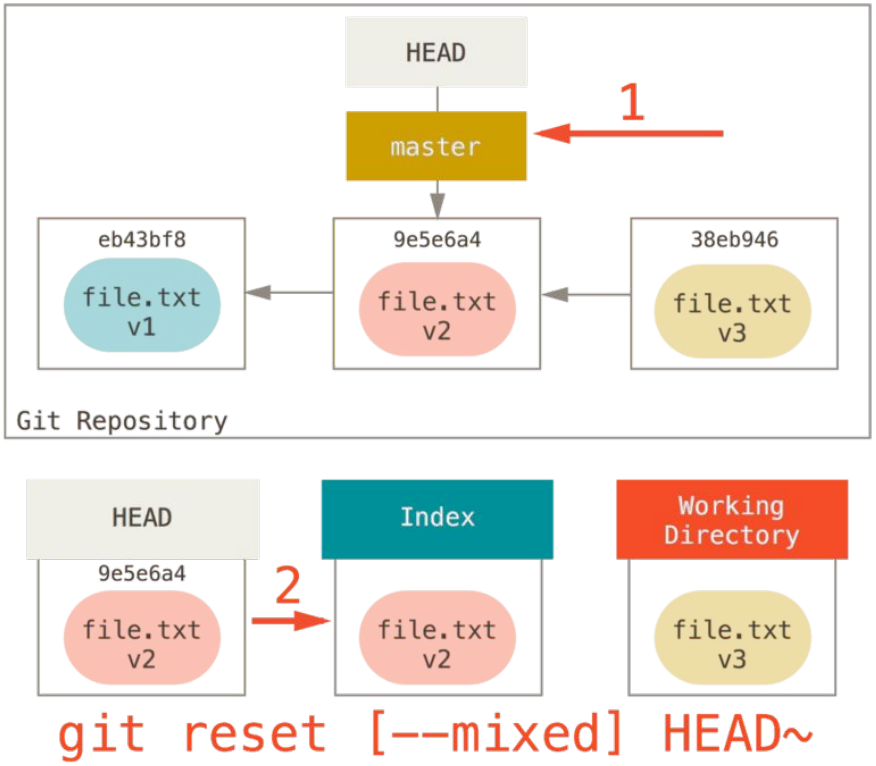
Step 2 (Updating the Index)

The index will be updated to be the same as the state in commit **9e5e6a4** i.e. it also unstaged everything.

However, the working directory wasn't modified, so **file.txt** still has the content of the version 3, i.e. **file.txt v3**, and **`git status`** will show that file.txt as modified.

Note that without the **`--hard`** option, **`git reset`** will stop at this step.

--mixed means stopping at this step, but it's the default anyways.

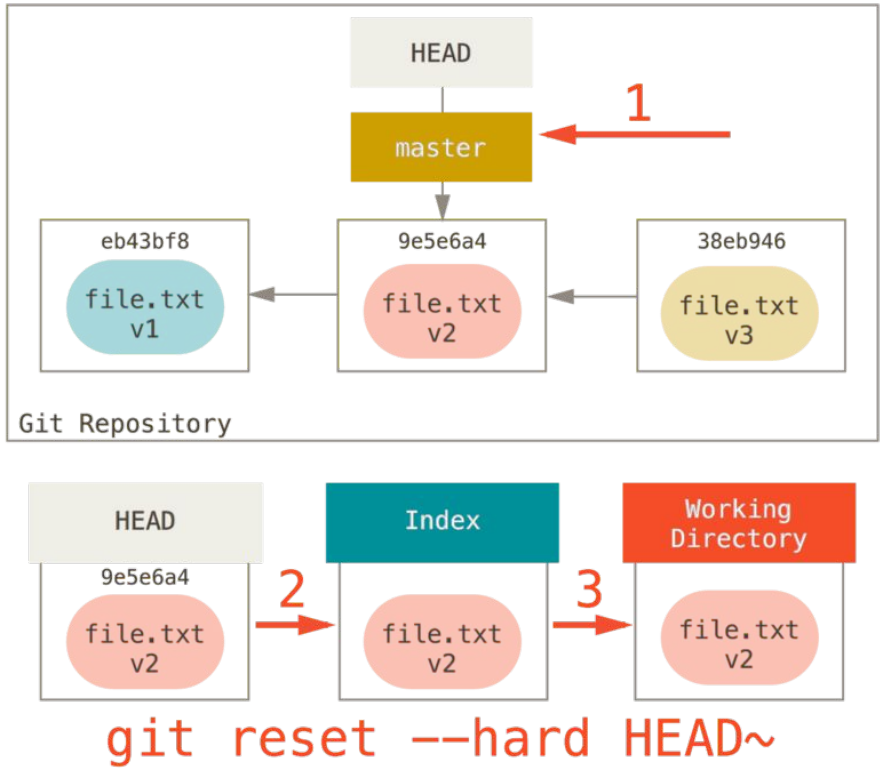


Step 3 Updating the Working Directory (--hard)

`git reset --hard 9e5e6a4`

will reset the changes to the actual files that have been modified from their states in the commit **9e5e6a4**, i.e. the content of **file.txt** will become **file.txt v2**

This is a potentially dangerous operation because you will lose all the changes to **file.txt** at version 3.

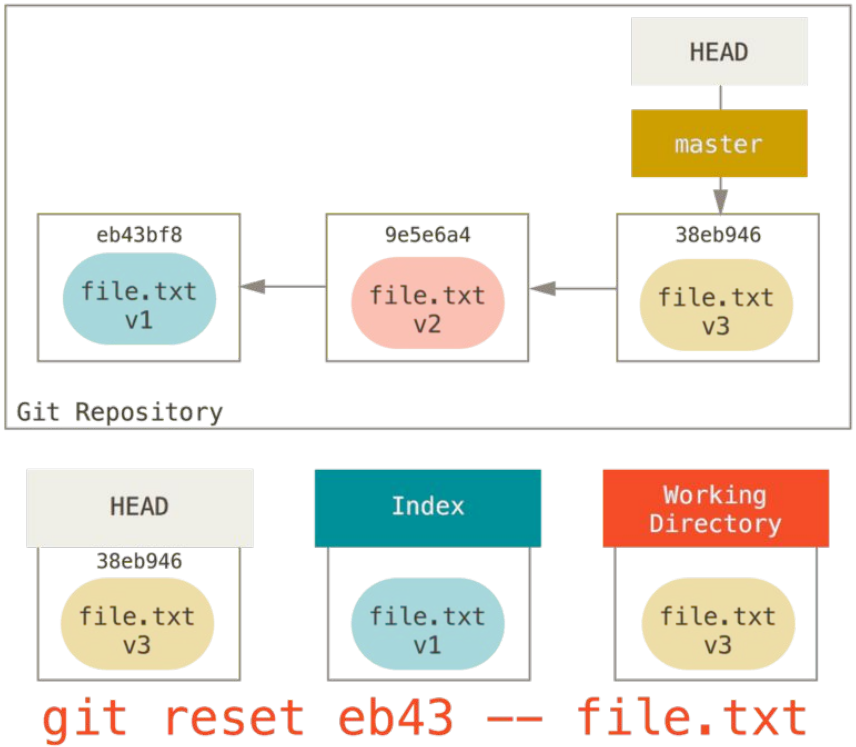


git reset at the File Level

`git reset eb43bf file.txt`

will not move the HEAD pointer.

You can also use the --hard option on this.



git submodule

Sometimes when the source code of a third party repository is needed in your own repository, instead of copy and pasting the source code, git submodule is a cleaner and more reliable solution.

Git submodules would allow you to keep a git repository as a subdirectory inside your own repository, while keeping your commits separate.

https://git-scm.com/book/en/v2/Git-Tools-Submodules

git submodule

To add a new submodule, use **`git submodule add`** with the absolute or relative URL of the project you would like to start tracking, e.g.

```
git submodule add https://github.com/eigenteam/eigen-git-mirror.git
```

A .gitmodules file will be created containing the mapping between the submodule's directory and the URL.

The .gitmodules file should be checked into the version control system.

After clone a repository with submodules or pulling updates from remotes, one has to do

```
git submodule update --init --recursive
```

to make the changes to the submodules happen locally.

Git Plumbing Commands

Plumbing commands refer to a set of git commands that do low-level work and are not usually used for ordinary purposes.

For example, **`git hash-object`** is a plumbing command:

```
echo 'test content' | git hash-object -w --stdin
```

--stdin indicates taking input from stdin, and -w indicates writing the content into a new file in .git/objects

Git Objects

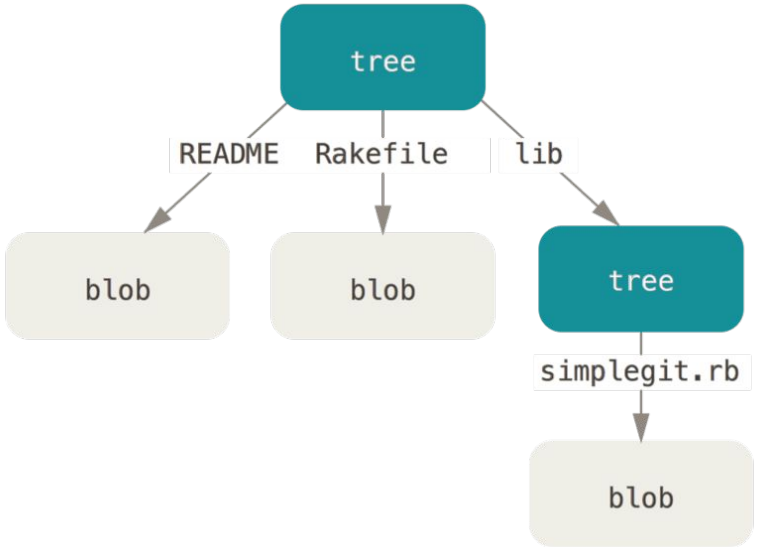
```
echo 'version 1' > test.txt
git hash-object -w test.txt
# returns a hash, say 83baae61804e65cc73a7201a7252750c76066a30

echo 'version 2' > test.txt
git hash-object -w test.txt
# returns another hash, say 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

find .git/objects -type f
# .git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
# .git/objects/83/baae61804e65cc73a7201a7252750c76066a30
# ...

git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30
# prints version 1
git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
# prints version 2

git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
# prints "blob", the type of the file, due to the -t option
# blob is the leaf of the directory tree, representing a regular file.
```

Tree Objects

Another type of git objects is a tree object, which represents directory entries.

For example, each commit will point to a tree object, which is the top directory of the repository when the commit snapshot is taken.

https://git-scm.com/book/en/v2/Git-Internals-Git-Objects



Commit Objects

Each commit is also stored as a git object, which will contain information such as the tree object is points to, the parent commit(s), author, commit message, etc.

Git References

Inside .git/refs/ there are files whose names are aliases to commit hashes.

For example, .git/refs/heads/master will contain the commit hash of the commit pointed to by the master branch.

Another set of references are the remotes.

For example, refs/remotes/origin/master will contain the commit hash pointed to by the master branch in the remote named origin.

Git Refspec

When we do something like **`git remote add origin some-url`**, an entry is created in the **.git/config** file:

```
[remote "origin"]
        url = some-url
        fetch = +refs/heads/*:refs/remotes/origin/*
```

The fetch has the format +<source>:<destination>, where <source> is the references on the remote, and <destination> is where those references will be tracked locally.
The + sign means update the local references even if the remote branch update is not a fast forward, e.g. if originally we have commits **A <-- B (origin/master)(master)**, then somebody viciously changed history on the remote to A <-- C, the + sign says this is okay and update the local references to be

```
A --- B <--(master)
 \
  C <--(origin/master)
```

Git Refspec

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

are all equivalent, because git expands all of them to **`refs/remotes/origin/master`**

Pushing Refspecs

Sometimes we might want to push a local branch/reference to a different reference on the remote. For example, if we want to push the local master branch to the qa/master branch in the origin remote (qa is a namespace, say for the QA team), we can do :
git push origin master:refs/heads/qa/master

To have a permanent mapping for fetch and push, we can update the .git/config file to

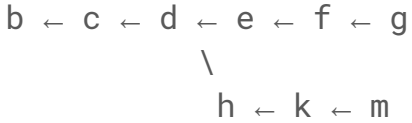
```
[remote "origin"]
        url = some-url
        fetch = +refs/heads/*:refs/remotes/origin/*
        push = refs/heads/master:refs/heads/qa/master
```

Partially Ordered Set (Poset)

A binary relation "<=" on a set S is a partial order if "<=" is

- reflexive, i.e. **a <= a** for all a in S
- antisymmetric, i.e. **a <= b and b <= a implies a = b**
- transitive, i.e. **a <= b, b <= c implies a <= c**

For example, for a rooted tree, if we define a <= b if a is an ancestor of b, then this "<=" is a partial order on the nodes of the tree. Note that in general only a subset of the pairs are comparable, e.g. in the tree



with **b** being the oldest ancestor, we have **c <= f** and **c <= k**, but neither **e <= k** nor **k <= e** is true, since neither is an ancestor of the other.

Total Order

If a partial order can be applied to **every pair** of elements in the set S, then it is a **total order**.

That is, for every pair **a, b,** if either **a <= b**, or **b <= a** or **both**, then "<=" is a total order.

For example, the usual less-than-or-equal-to relation "<=" on the set of integers is a total order.

Topological Order

A **topological order** on a directed acyclic graph (DAG) G = (V, E) is a **total order** on all of its vertices such that if there is a directed edge from **v1** to **v2**, then **v1** < **v2**.

We can view a topological ordering/sorting on G as arranging the vertices on a horizontal line such that all edges go from left to right.

Depth-First Search (DFS) for Both Directed and Undirected Graphs

```
dfs(G = (V, E), s):
    for each vertex u in V:
        u.color = white
        u.discoverer = None
    time = 0
    for each vertex u in G.adj[s]:
        if u.color == white:
            dfs_visit(u)
```

```
dfs_visit(u):
    time = time + 1
    u.discover_time = time
    u.color = gray
    for each v in G.adj[u]:
        if v.color == white:
            v.discoverer = u
            dfs_visit(v)
    u.color = black
    time = time + 1
    u.finish_time = time
```

This is the pseudo code.

- white means undiscovered
- gray means being processed
- black means finished processing

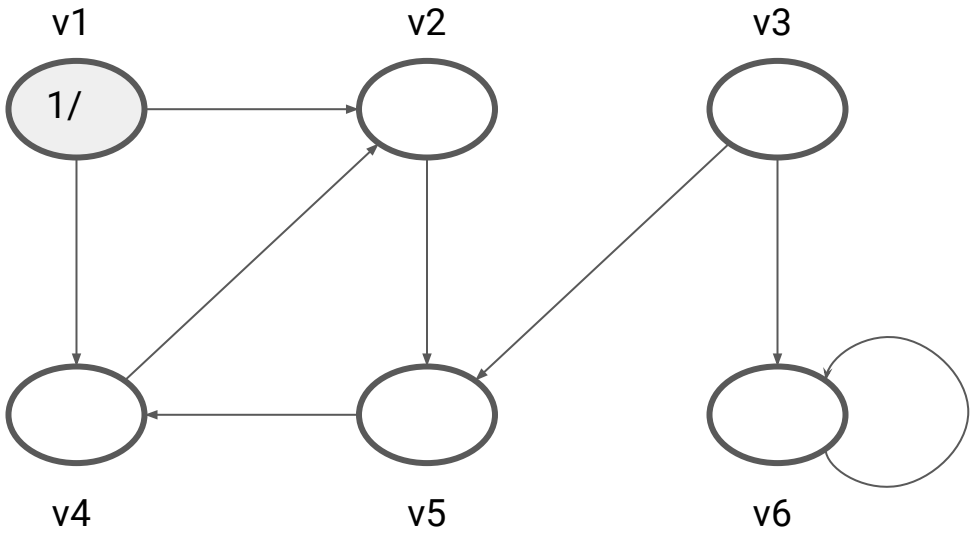
G.adj[**u**] is the set of vertices reachable from **u**.

A stack can (and probably should) be used to implement DFS instead of recursion.

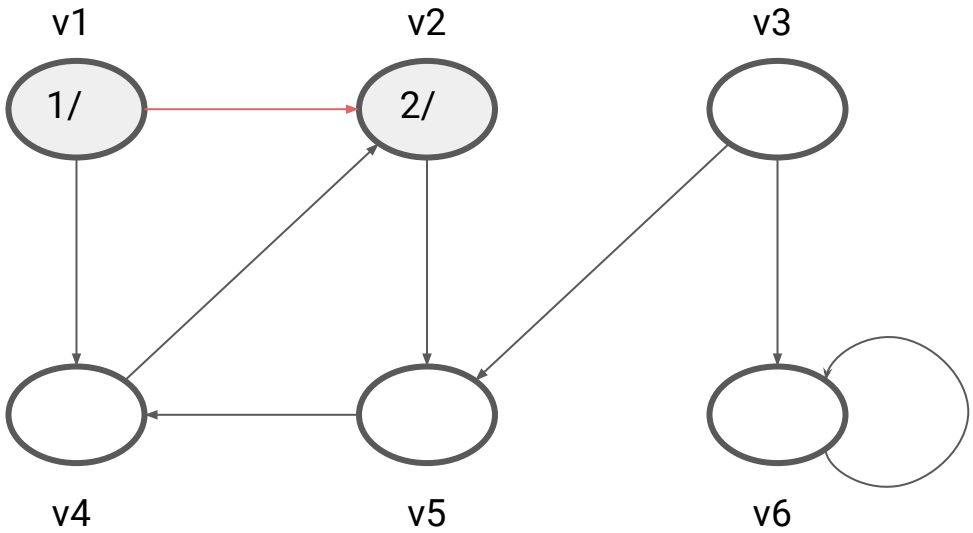
In fact, a stack should be used in Python to implement DFS due to the interpreter's limit on recursion depth.

DFS Example on a Directed Graph

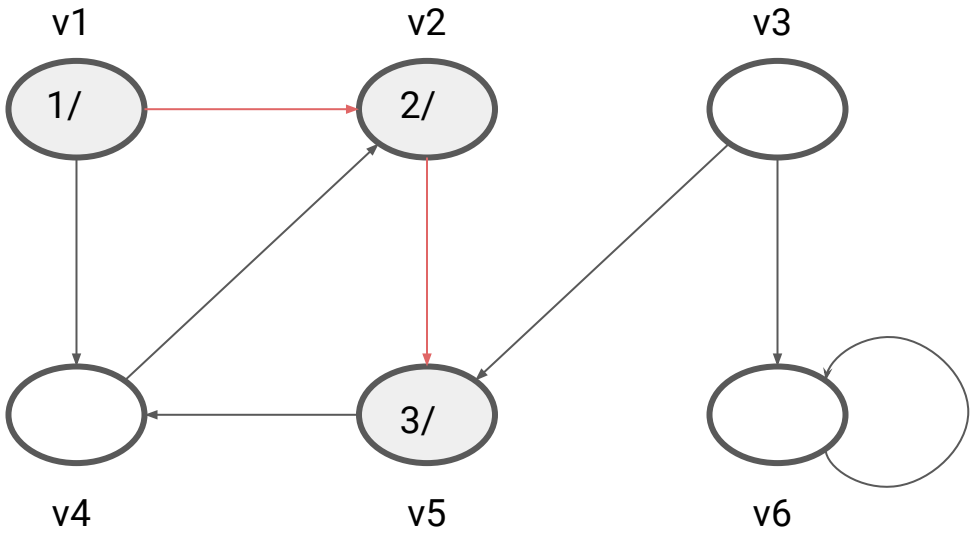
DFS(G, v1)



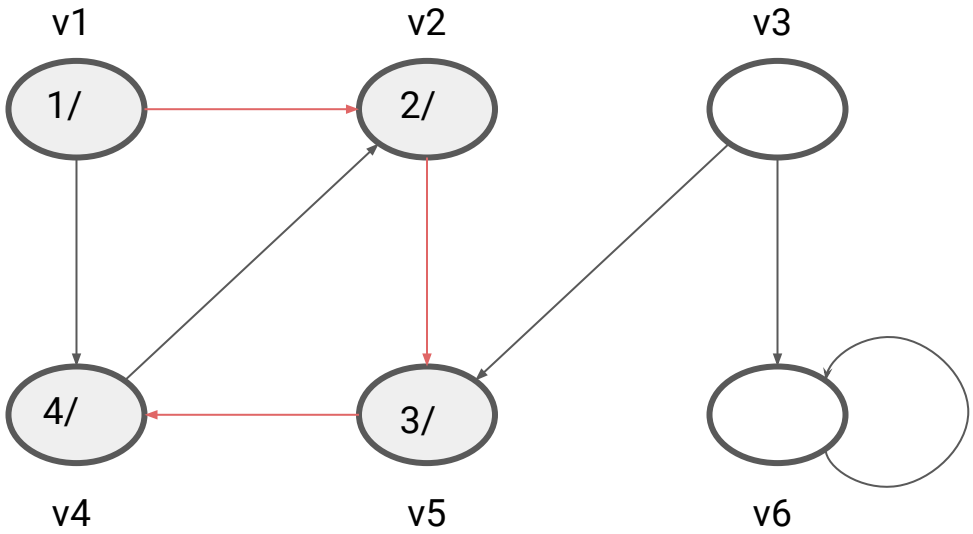
DFS Example on a Directed Graph



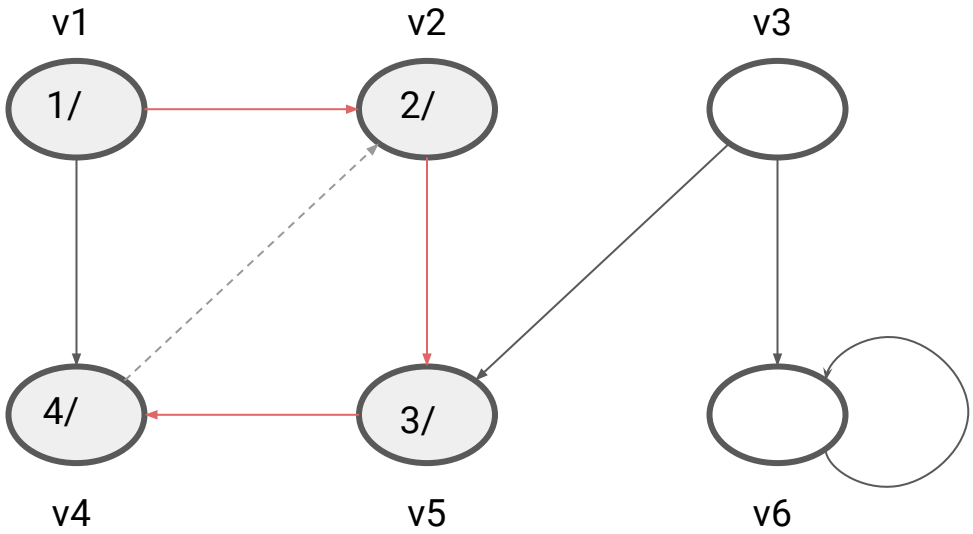
DFS Example on a Directed Graph



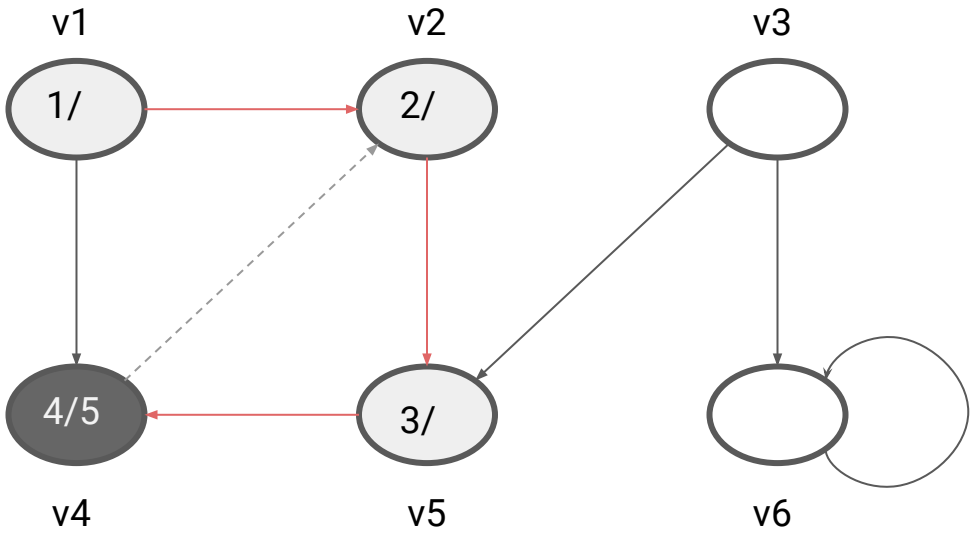
DFS Example on a Directed Graph



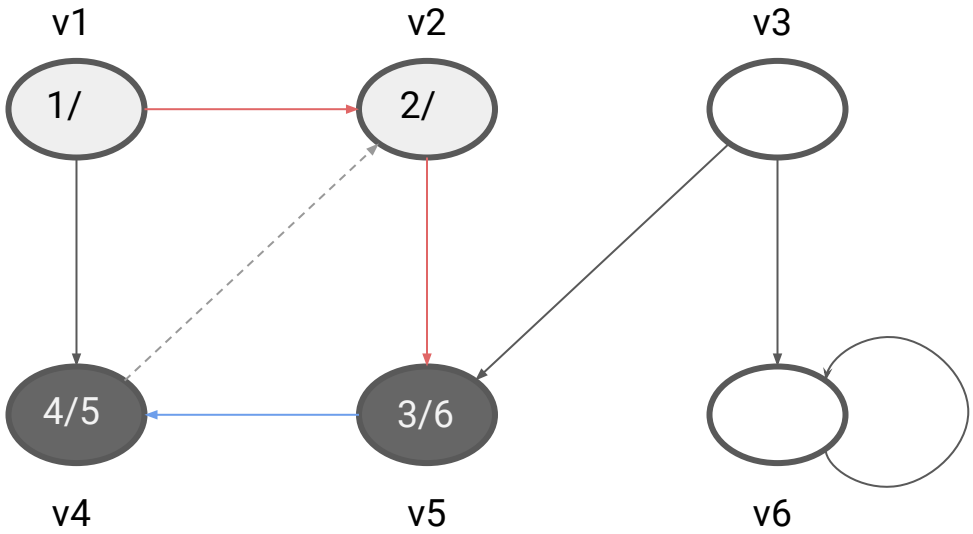
DFS Example on a Directed Graph



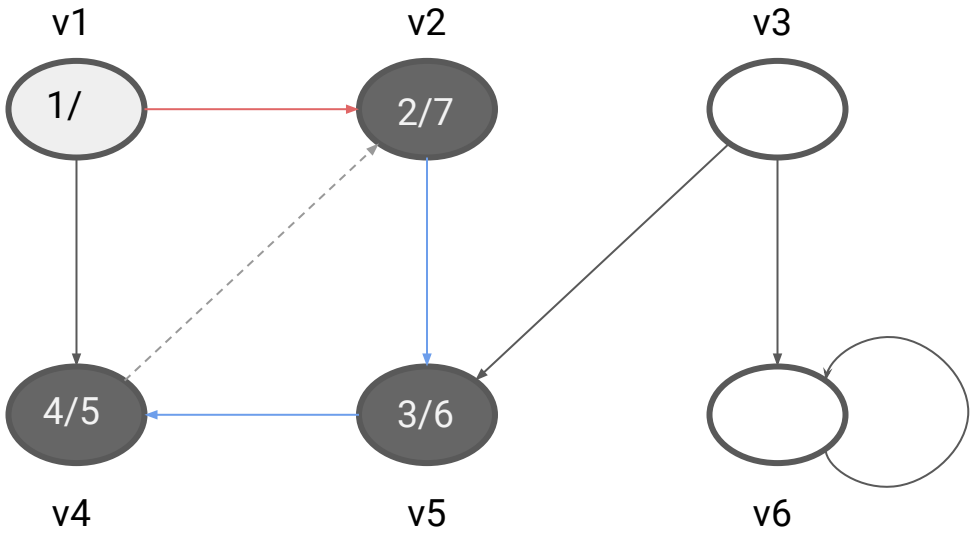
DFS Example on a Directed Graph



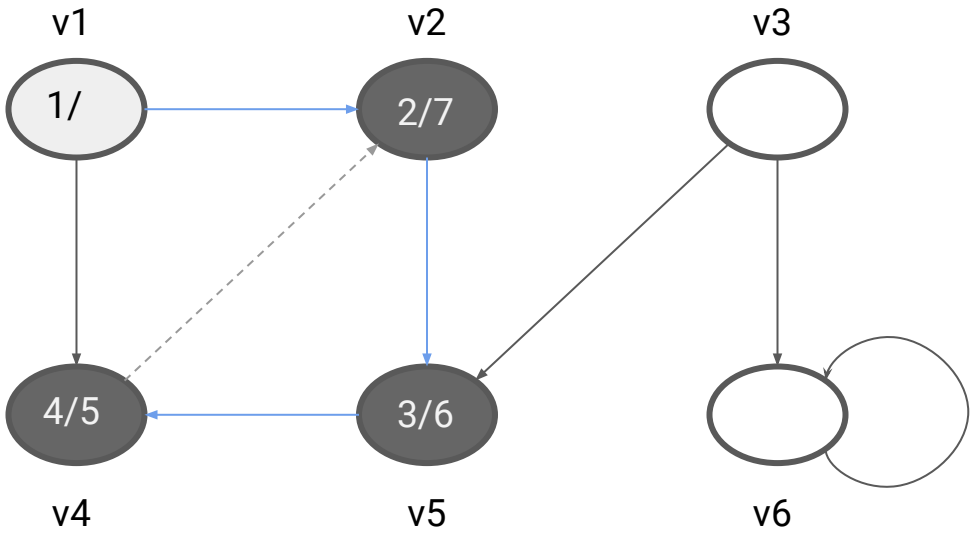
DFS Example on a Directed Graph



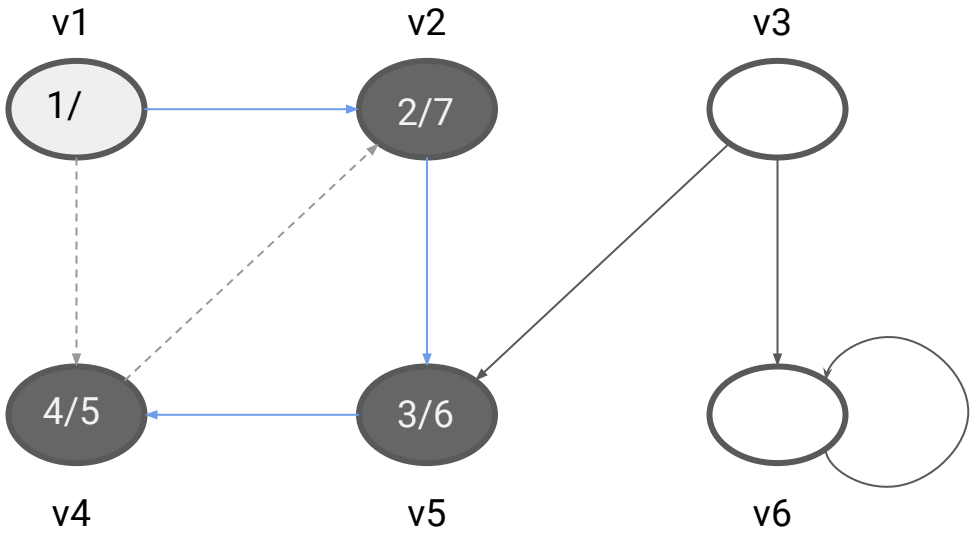
DFS Example on a Directed Graph



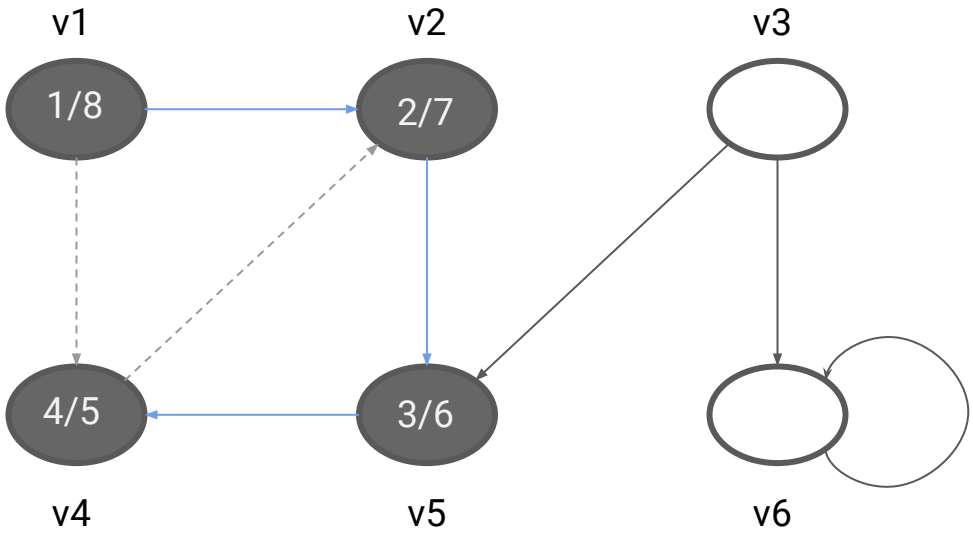
DFS Example on a Directed Graph



DFS Example on a Directed Graph

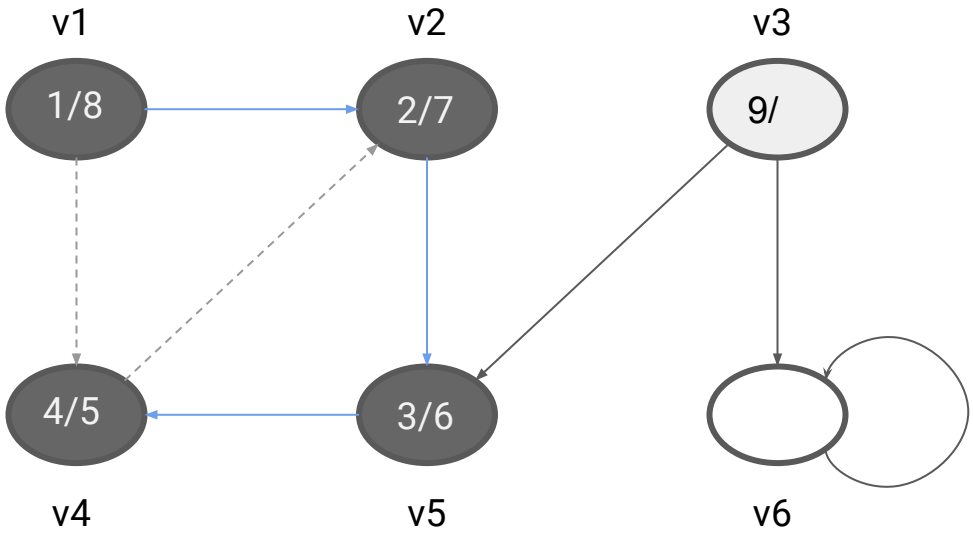


DFS Example on a Directed Graph

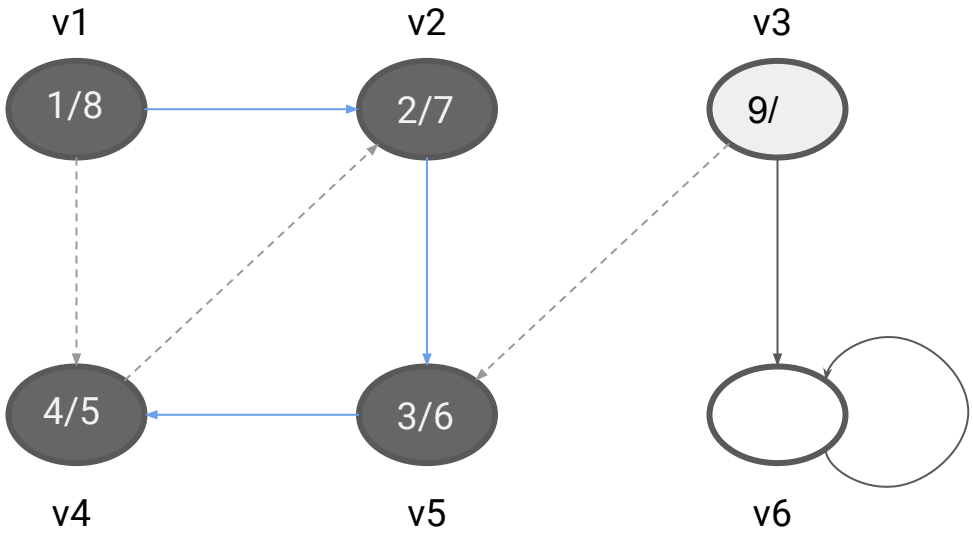


DFS Example on a Directed Graph

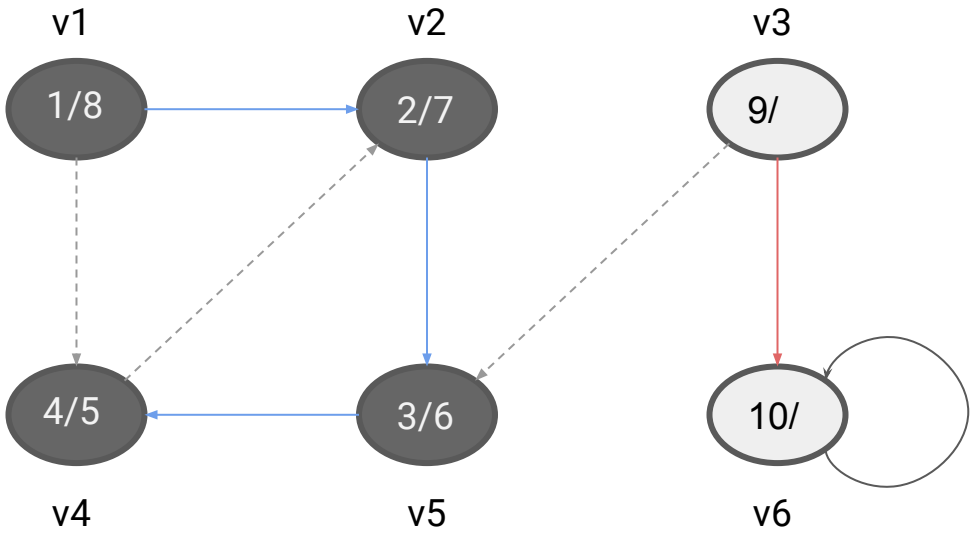
DFS(G, v3)



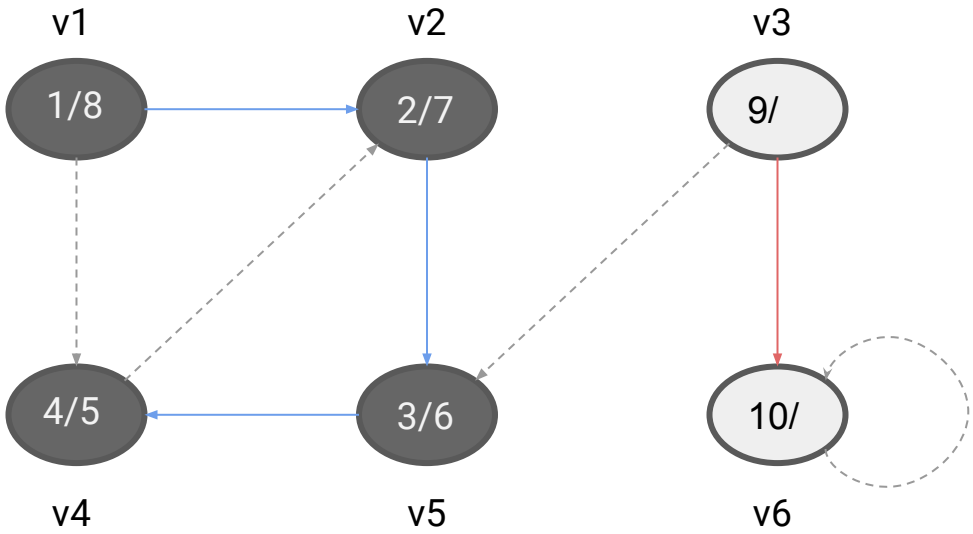
DFS Example on a Directed Graph



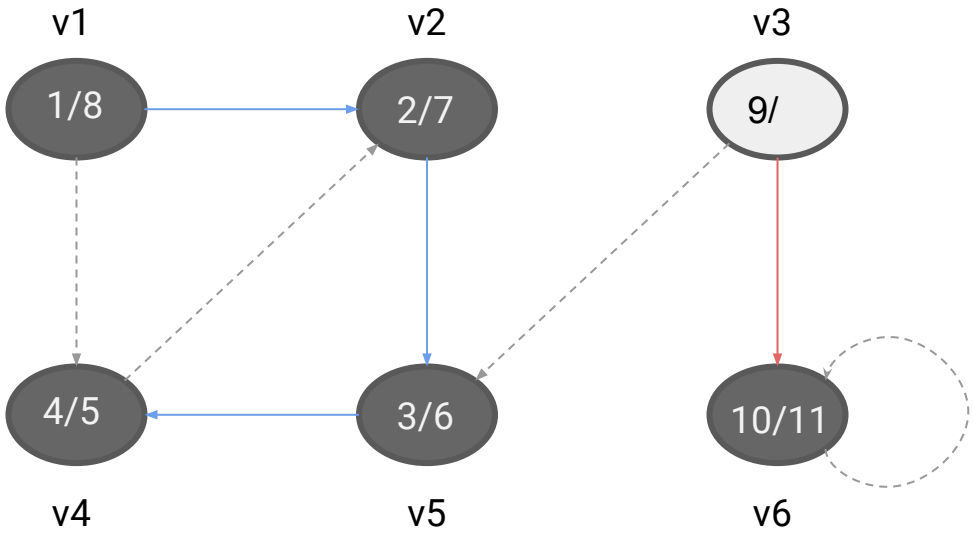
DFS Example on a Directed Graph



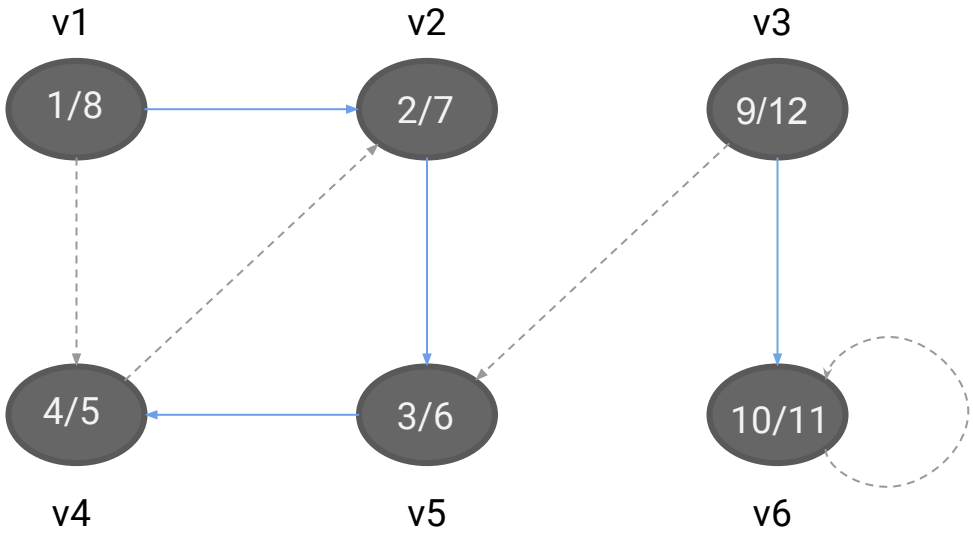
DFS Example on a Directed Graph



DFS Example on a Directed Graph



DFS Example on a Directed Graph

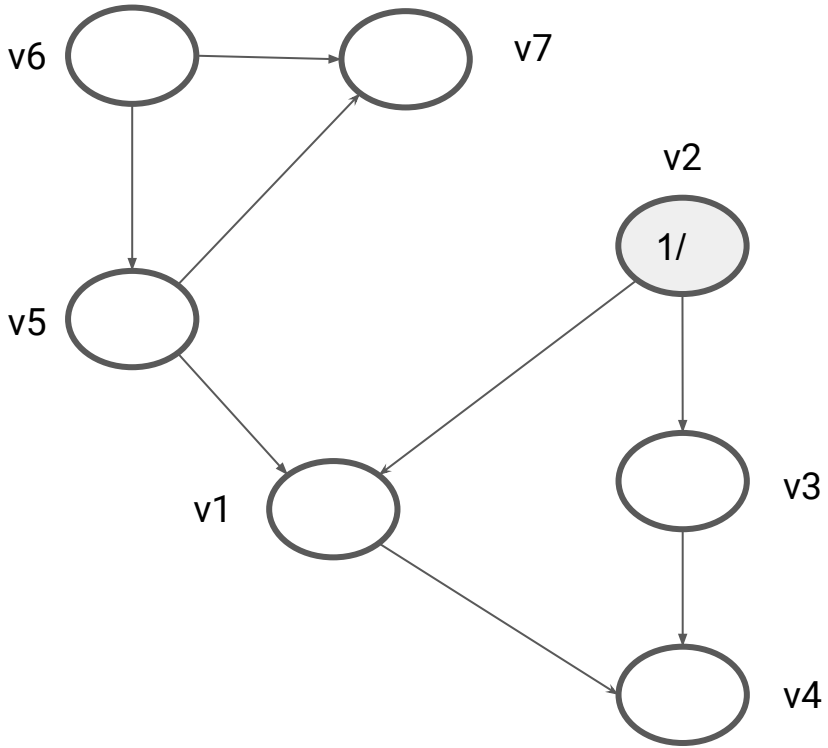


Topological Sort

One way to perform a topological sort on a DAG G is to run DFS on G, and as each vertex is finished being processed, i.e. turned black, append it onto a FIFO queue.

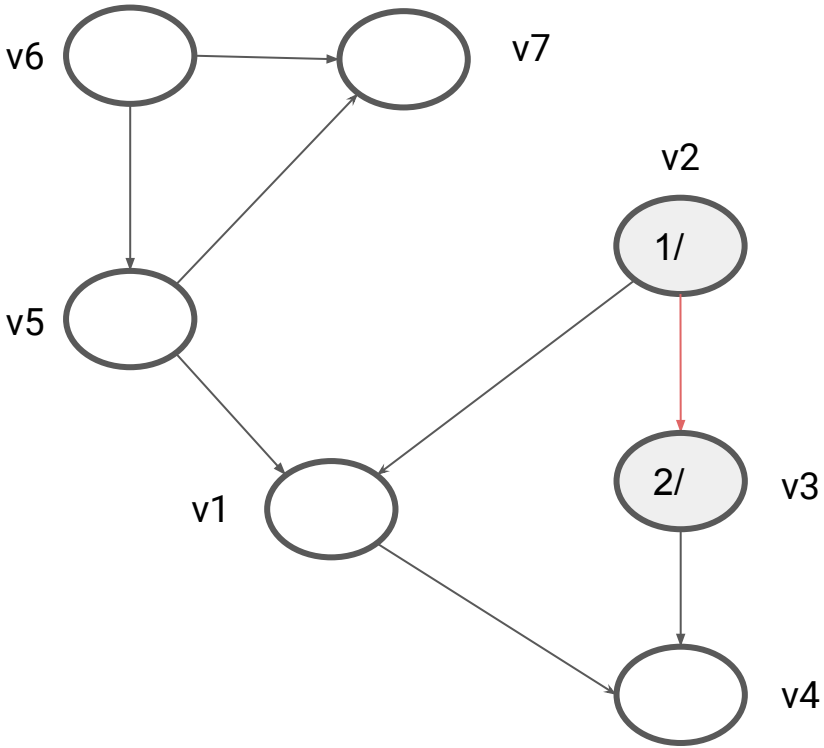
The resulting queue will contain a topological ordering of the vertices, where the first out element is the smallest and so on.

Topological Sort Example



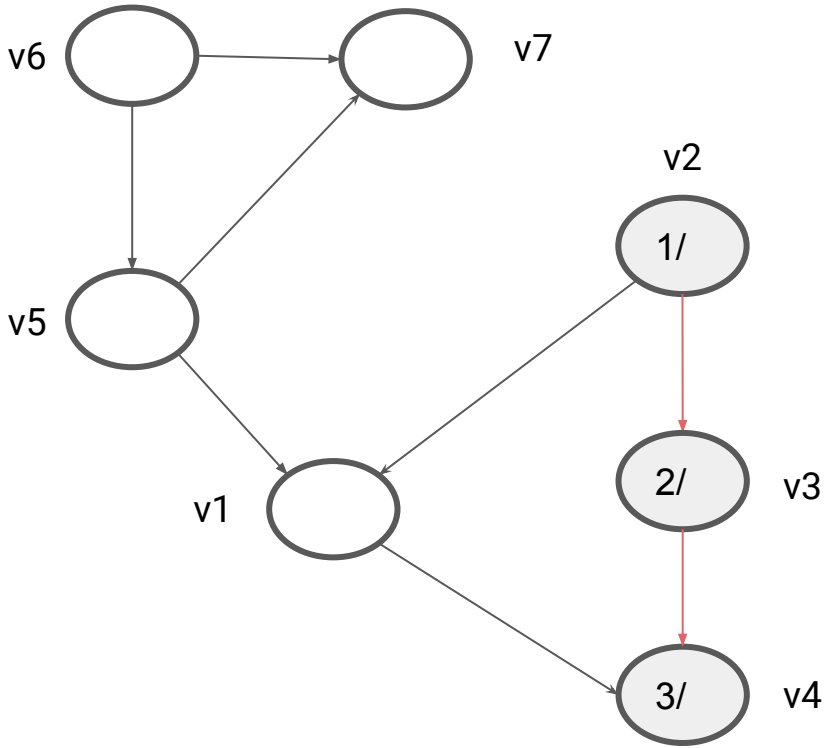
topologically ordered vertices:
[ ]

Topological Sort Example



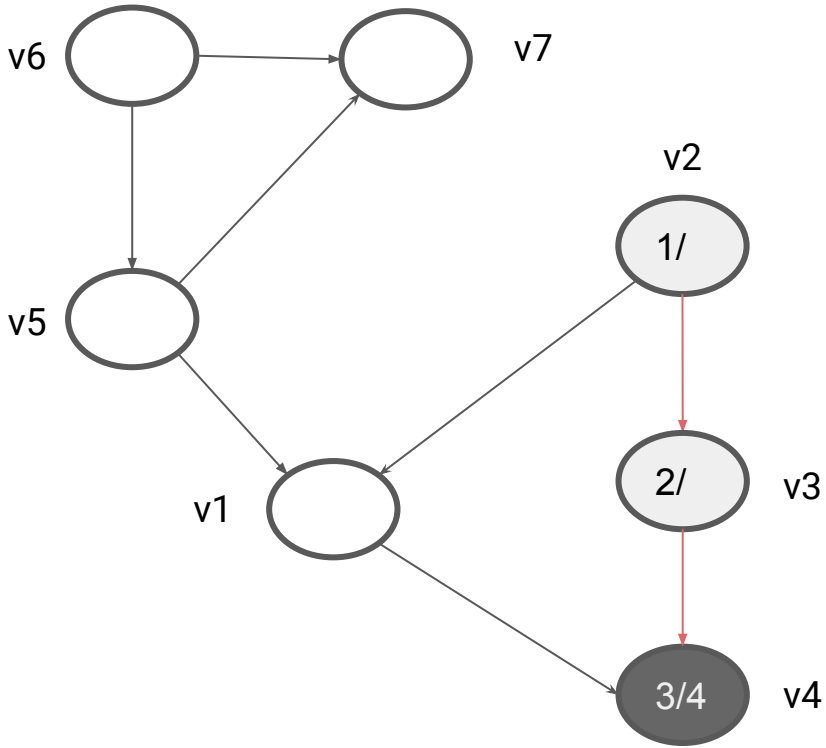
topologically ordered vertices:
[ ]

Topological Sort Example



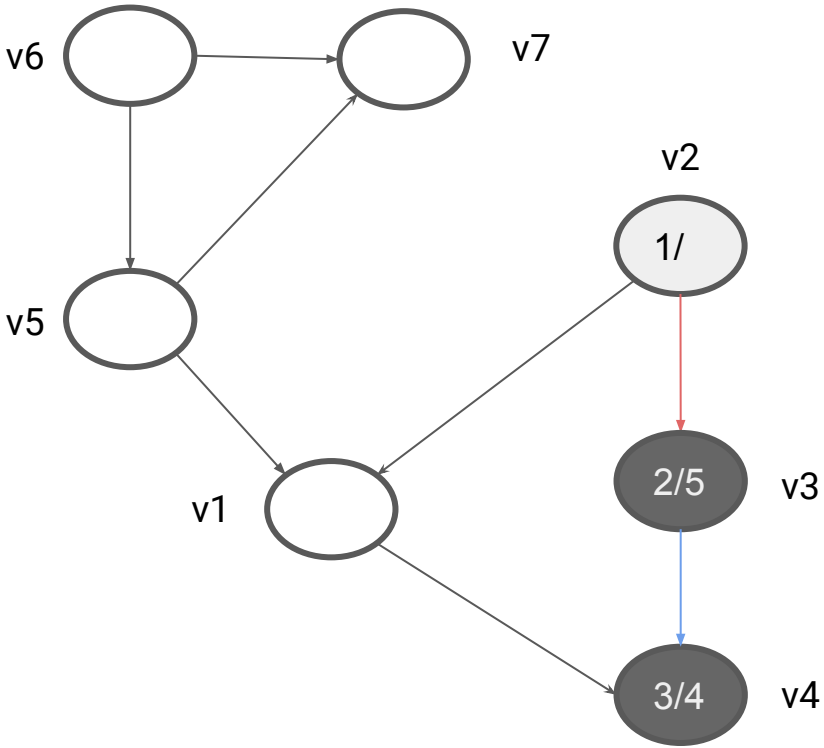
topologically ordered vertices:
[ ]

Topological Sort Example



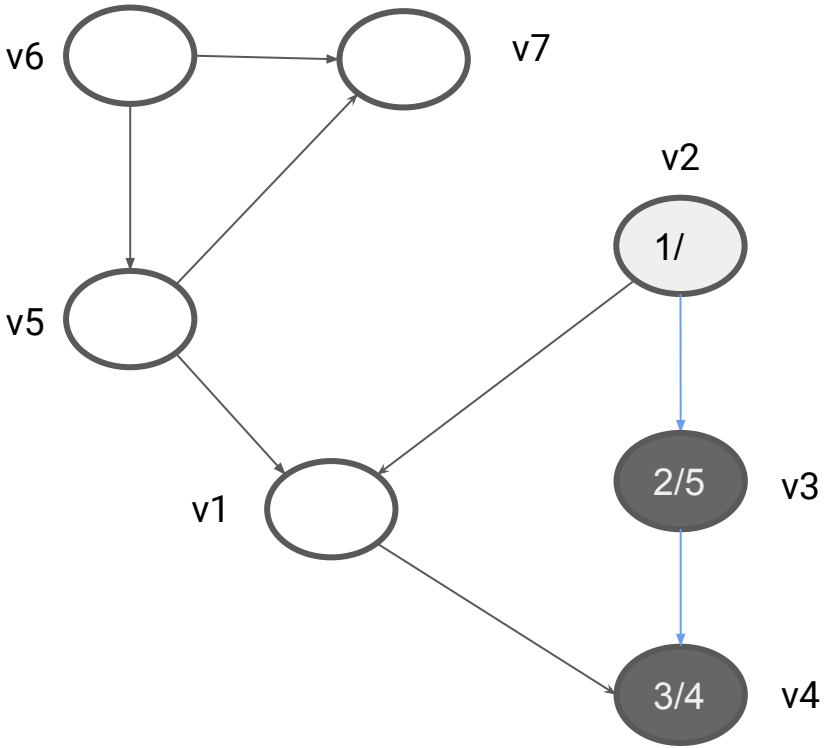
topologically ordered vertices:
[v4]

Topological Sort Example



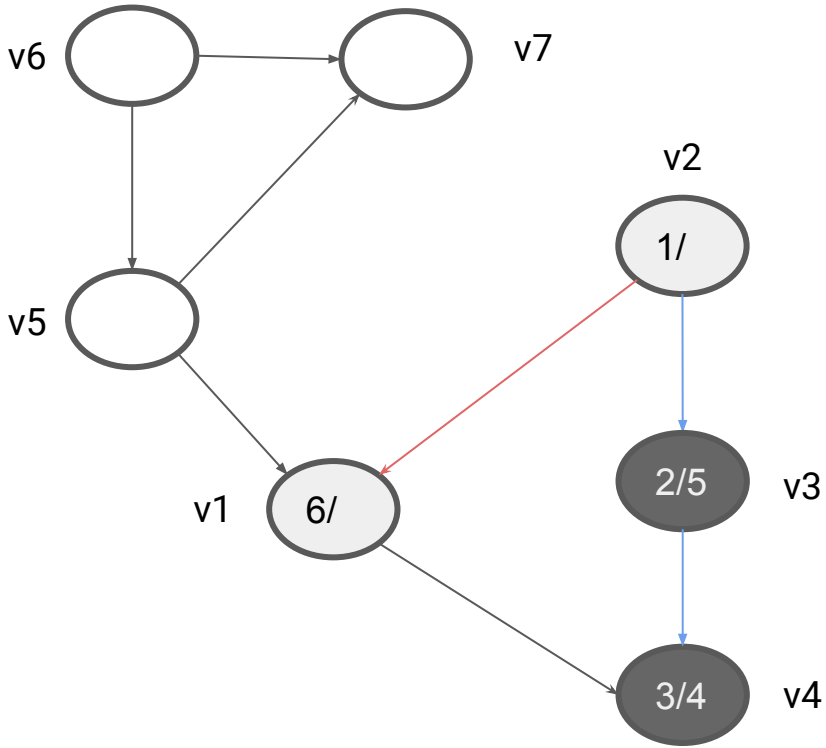
topologically ordered vertices:
`[v4, v3]`

Topological Sort Example



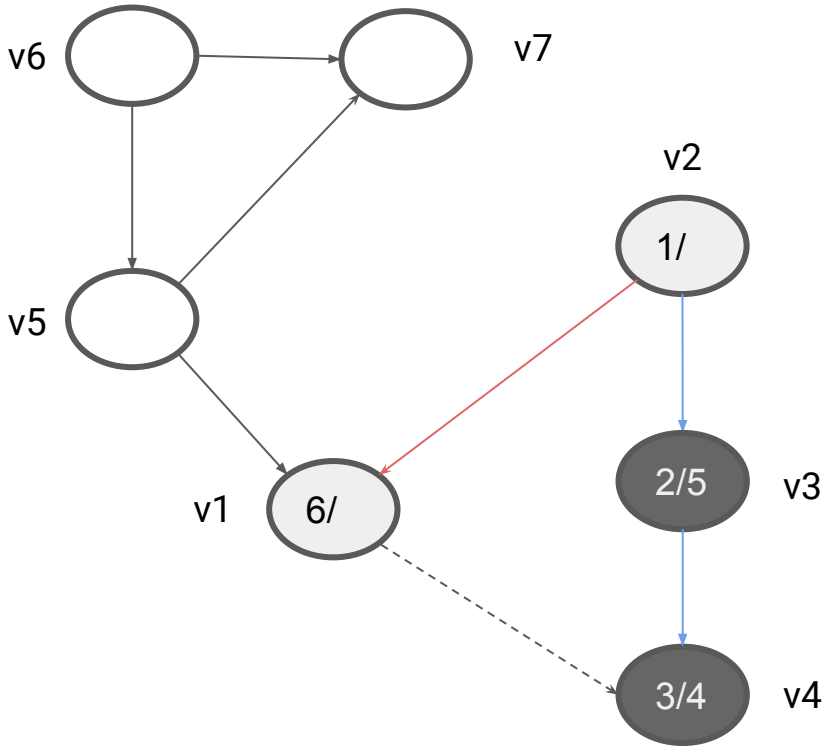
topologically ordered vertices:
`[v4, v3]`

Topological Sort Example



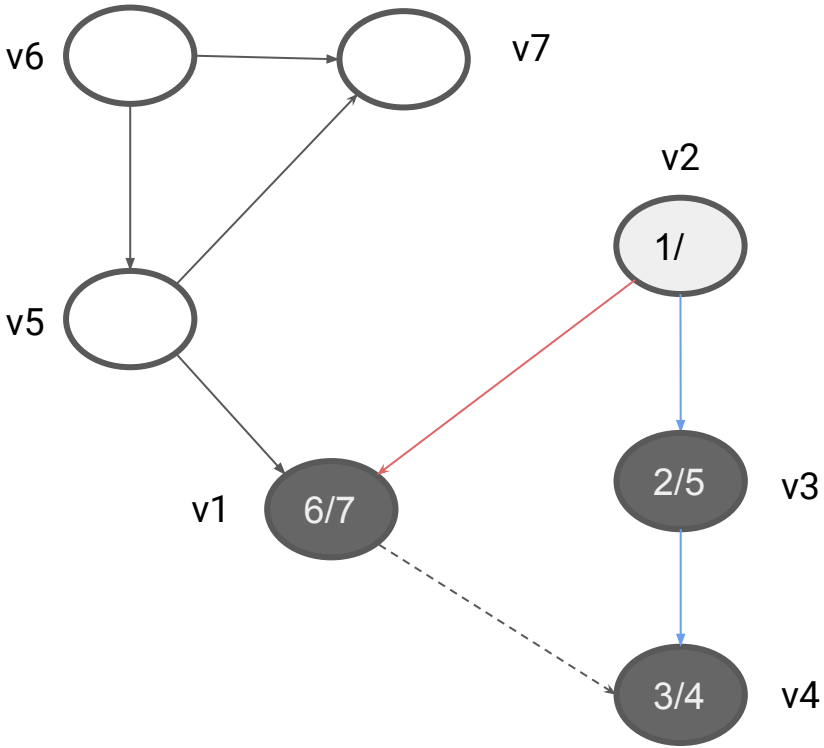
topologically ordered vertices:
`[v4, v3]`

Topological Sort Example



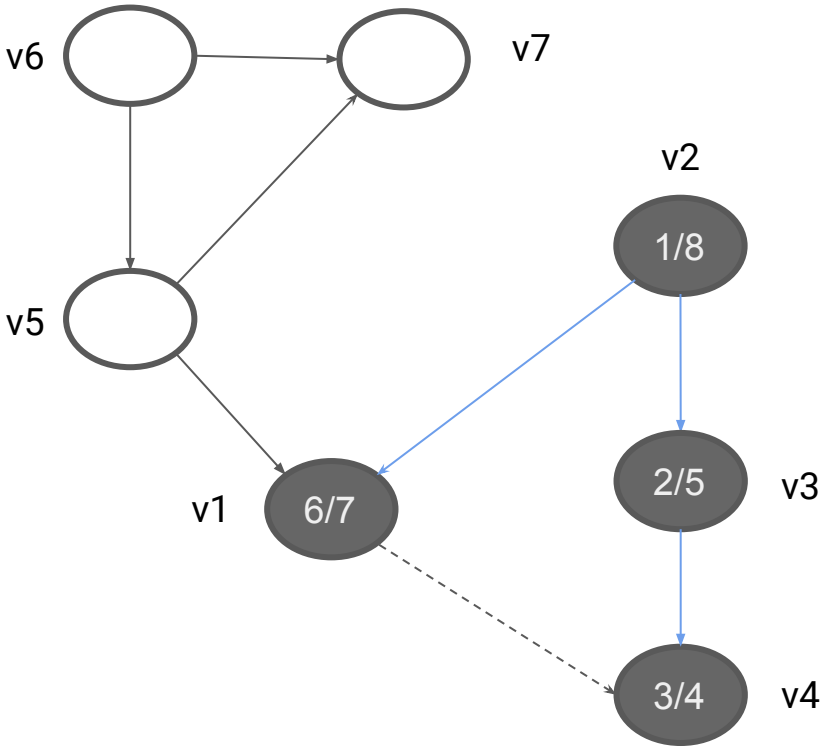
topologically ordered vertices:
`[v4, v3]`

Topological Sort Example



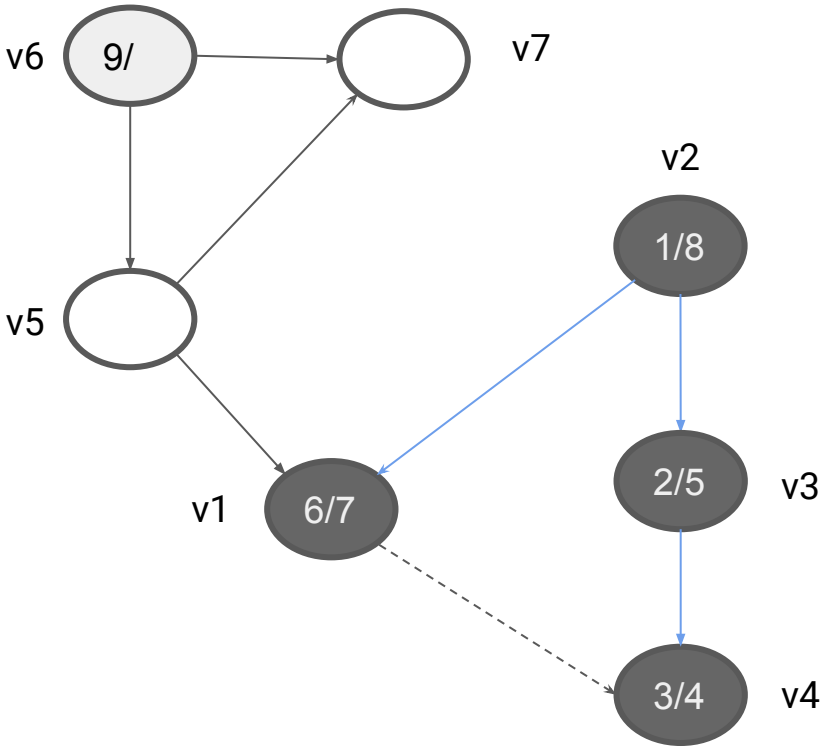
topologically ordered vertices:
`[v4, v3, v1]`

Topological Sort Example



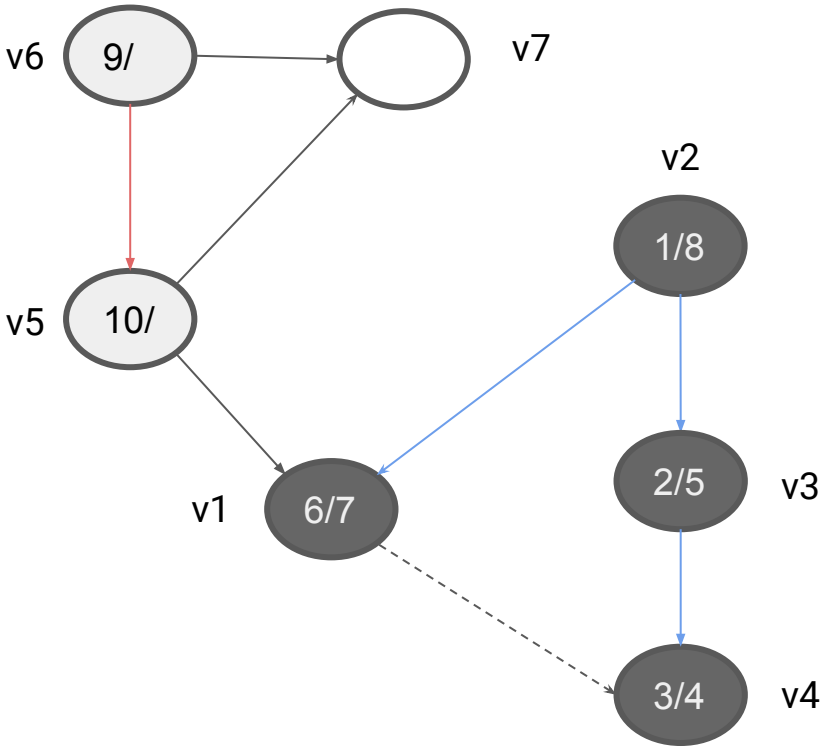
topologically ordered vertices:
`[v4, v3, v1, v2]`

Topological Sort Example



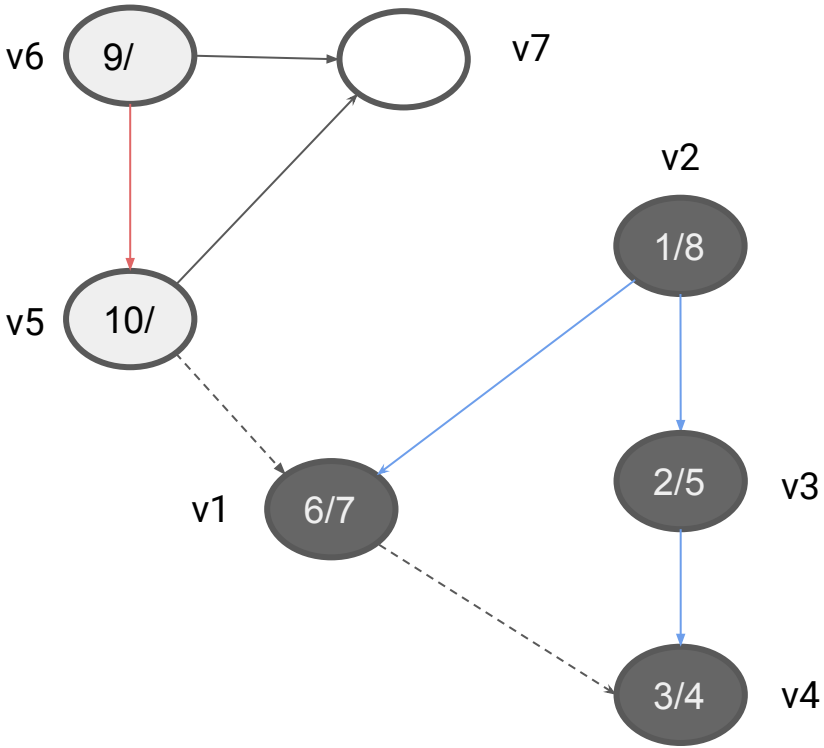
topologically ordered vertices:
`[v4, v3, v1, v2]`

Topological Sort Example



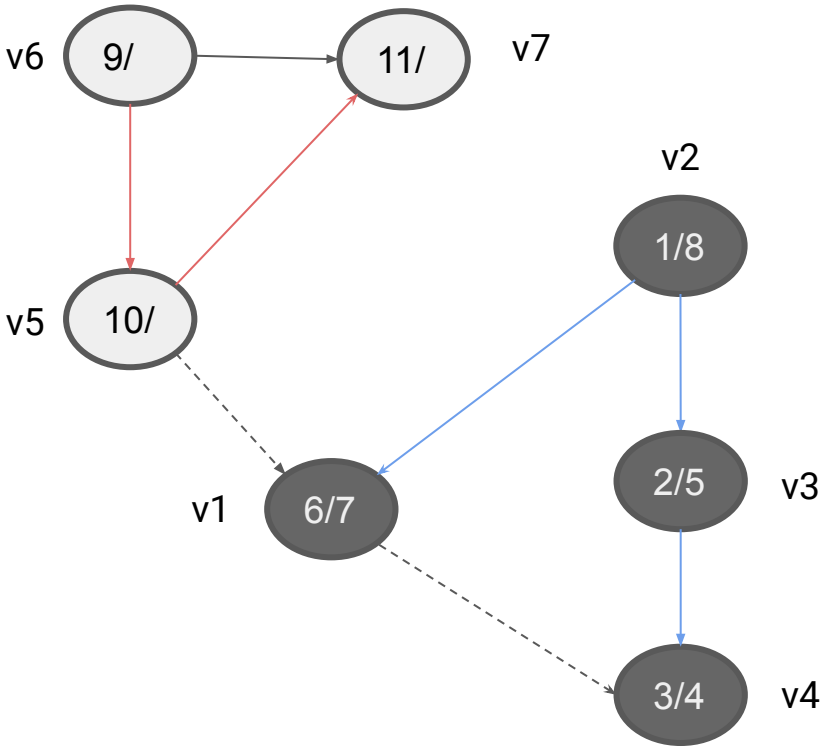
topologically ordered vertices:
`[v4, v3, v1, v2]`

Topological Sort Example



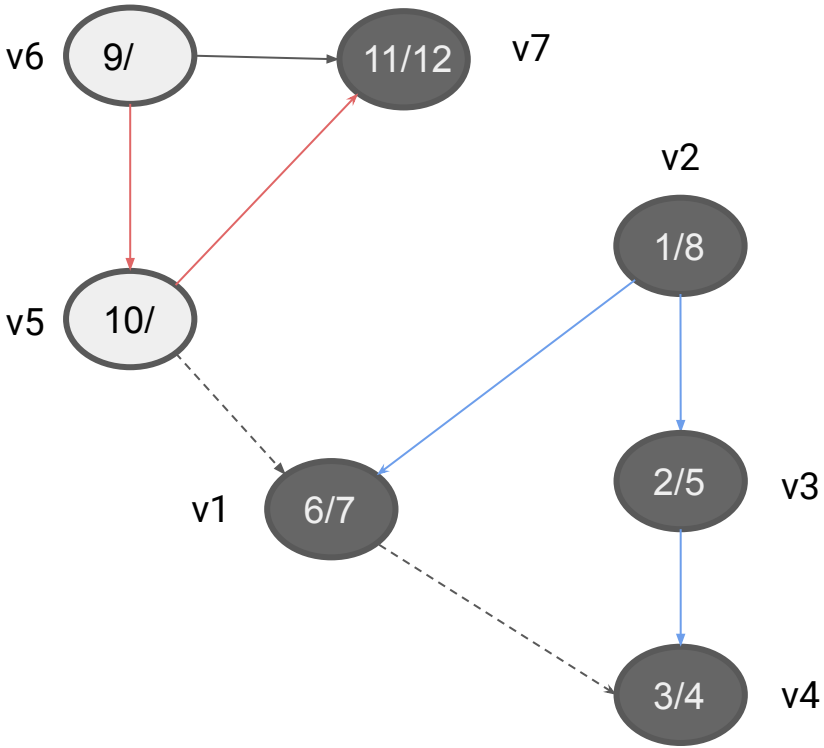
topologically ordered vertices:
`[v4, v3, v1, v2]`

Topological Sort Example



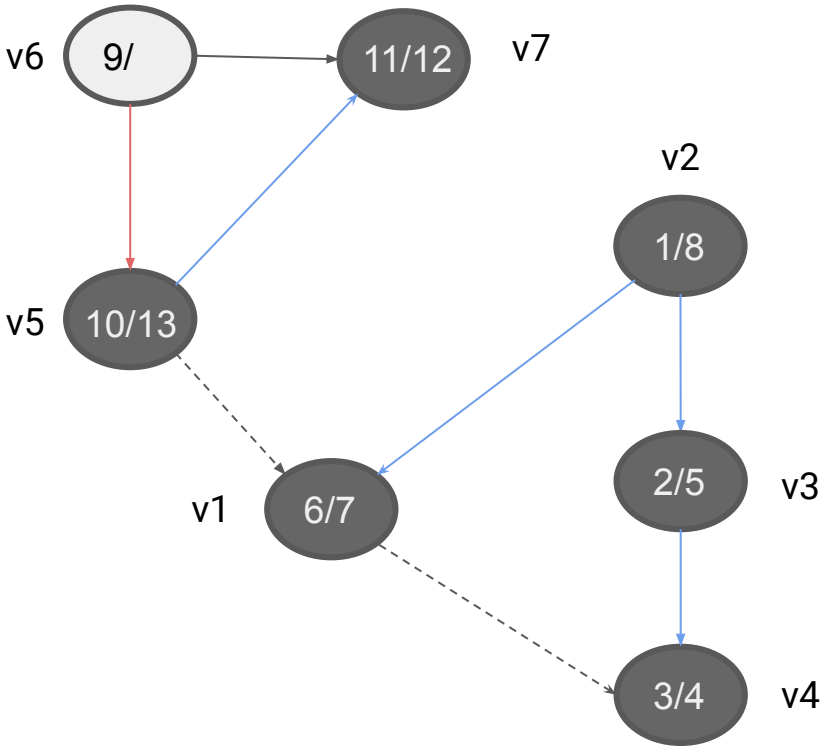
topologically ordered vertices:
`[v4, v3, v1, v2]`

Topological Sort Example



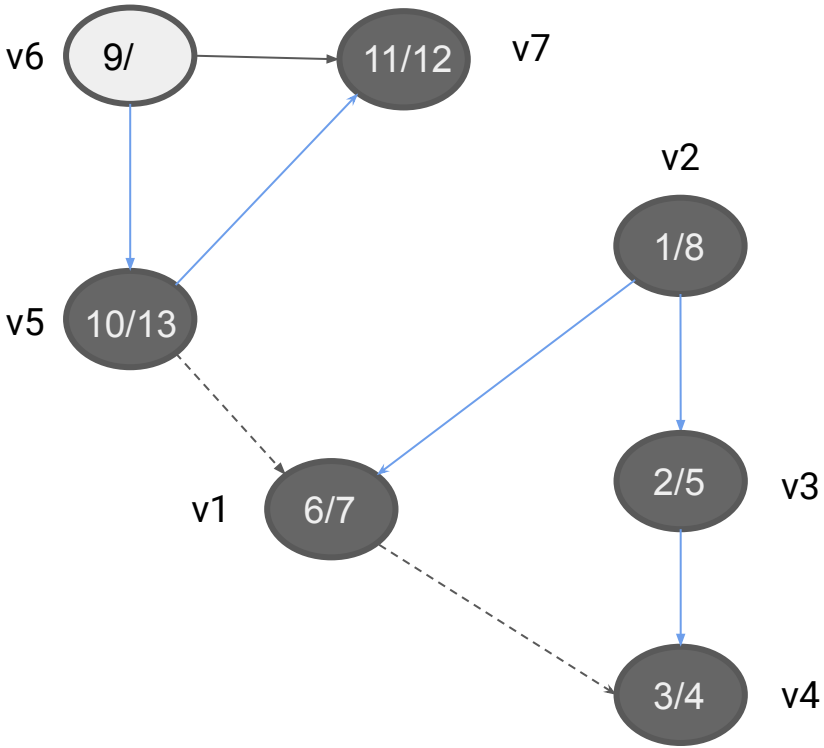
topologically ordered vertices:
`[v4, v3, v1, v2, v7]`

Topological Sort Example



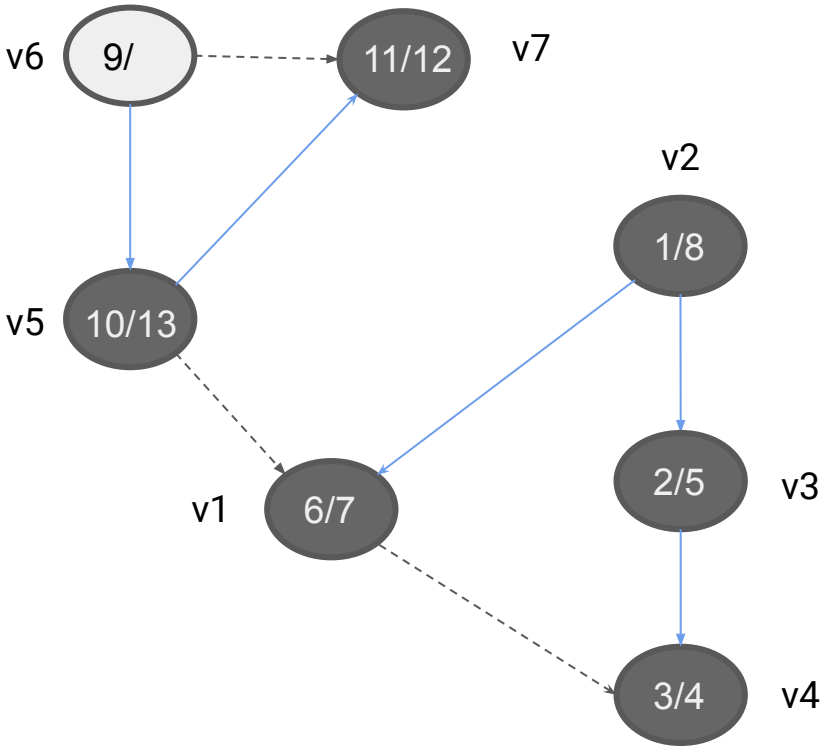
topologically ordered vertices:
`[v4, v3, v1, v2, v7, v5]`

Topological Sort Example



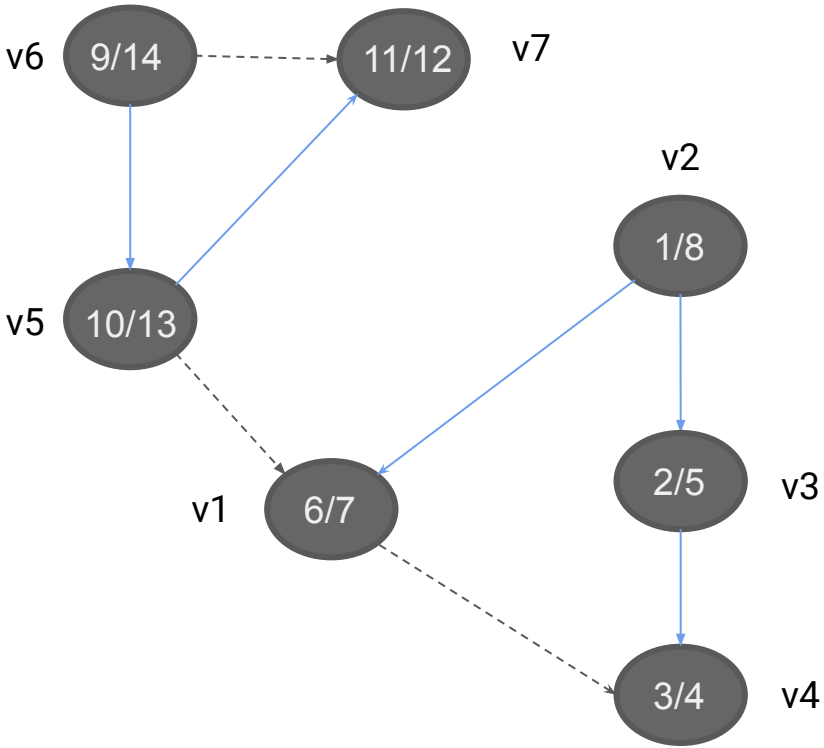
topologically ordered vertices:
`[v4, v3, v1, v2, v7, v5]`

Topological Sort Example



topologically ordered vertices:
`[v4, v3, v1, v2, v7, v5]`

Topological Sort Example



topologically ordered vertices:
`[v4, v3, v1, v2, v7, v5, v6]`

Python DFS Implementation Using a Stack

Depending on your need, each element on the stack can be a vertex coupled with some extra info.

For example, if you want to keep track of the path, something like this will help:

```
stack = [(vertex, [vertex])]
visited = set()
while stack:
    v, path = stack.pop()
    if v in visited:
        continue
    visited.add(v)
    do something...
    for child in v.adjacent_vertices:
        if child not in visited:
            stack.append((child, path + [child]))
        do something...
```