CS 3510 — Algorithms, Spring 2022

Lecture 12: Intro to Graph Algorithms

February 22, 2022

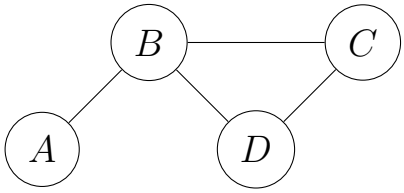
Lecturer: Frederic Faulkner *Scribe: Aditya Diwakar*

Exam 2 is next Thursday. It will cover Dynamic Programming (Lectures 7-11). The review lecture will be next Tuesday and HW2B is due Friday.

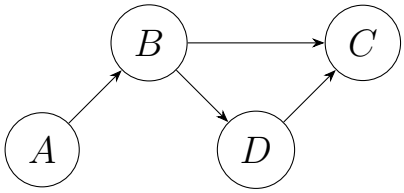
1 Graph Algorithms

First, what is a graph? A graph is something that has edges and vertices (V, E). There are two types of graphs:

- Undirected graph (all edges go both ways between vertices)



- Directed graph (edges go from one vertex to another)



As this is an algorithms class, we care about to store our graphs. There are two ways to store these graphs both with benefits.

1. Adjacency Matrix which is a matrix that has a 1 in entry i, j if there is an edge from $i \to j$. In the first graph shown above, the adjacency matrix would be

$$G = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

   An adjacency matrix for an undirected graph will be symmetrical along the diagonal as an edge going from $i \to j$ will also go from $j \to i$.

2. Adjacency list which is an array of lists where there is a list for each vector that contains the vertices reachable from v. For example with the above example, we can write:

$$G = \begin{cases} A : [B] \\ B : [C, D] \\ C : \emptyset \\ D : [C] \end{cases}$$

   which shows that we can reach B from A and C, D from B, etc.

Which one should we use? What is the difference? One way to look at it is by using an argument about space complexity. With an adjacency matrix, it takes up $\mathcal{O}(|V|^2)$ space due to the design of the matrix.

With an adjacency list, each vertex shows up once and each edge appears also once giving a total space bound of $\mathcal{O}(|V| + |E|)$.

What about edge lookup? In an adjacency matrix, it takes $\mathcal{O}(1)$ time to look up whether an edge exists as we can simply check the entry i, j to see if there is an edge $i \to j$.

With an adjacency list, we need to search through the vertices that can be reached from a certain vertex. There can be a total of $|V|$ vertices to look through, and even using something like storing the edges in a sorted manner, then it costs $\mathcal{O}(\log |V|)$.

So, although edge lookup is fast, we are wasting space with the matrix representation. So which is better? First, note there are two types of graphs:

1. Sparse Graph: a graph with a small number of edges relative to $|V|$
2. Dense Graph: a graph with $|E| >> |V|$ (significantly more edges than $|V|$

Hence, since a dense graph has significantly more edges than vertices, we can use an adjacency matrix as we aren't saving much space by using an adjacency list (and edge lookups are slower). For a sparse graph, an adjacency list may be better.

1.1 DFS on Undirected Graph

Now, we can start looking at algorithms! Given an undirected graph and a vertex v, what other vertices can you reach from v? In other words, what are the components of G? (where a component is a maximally connected subgraph).

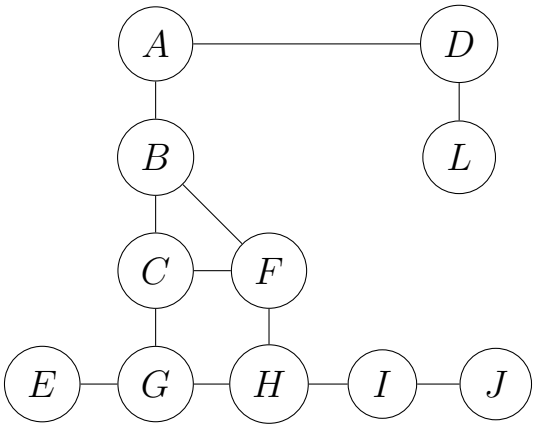
There are 2 ways to iterate through a graph:

1. DFS (depth first search); backed by a stack
2. BFS (breadth first search); backed by a queue

Today, we will investigate DFS! We can write an explore algorithm using DFS:

```
1 Function explore(v):
2     visited(v) ← true
3     for (v, x) ∈ E do
4         if ¬visited(x) then
5             explore(x)
```

An example of running this algorithm is below:



The explore method would run as follows:

Function Call	Stack	Visited
`explore(A)`	A, B	$\{A\}$
`explore(B)`	A, B, C	$\{A, B\}$
`explore(C)`	A, B, C, F	$\{A, B, C\}$
`explore(F)`	A, B, C, F, H	$\{A, B, C, F\}$
`explore(H)`	A, B, C, F, H, G	$\{A, B, C, F, H\}$
`explore(G)`	A, B, C, F, H, G, E	$\{A, B, C, F, H, G\}$
`explore(E)`	A, B, C, F, H, G	$\{A, B, C, F, H, G, E\}$
`explore(G)`	A, B, C, F, H	$\{A, B, C, F, H, G, E\}$
`explore(H)`	A, B, C, F, H, I	$\{A, B, C, F, H, G, E\}$
`explore(I)`	A, B, C, F, H, I, J	$\{A, B, C, F, H, G, E, I\}$
`explore(J)`	A, B, C, F, H, I	$\{A, B, C, F, H, G, E, I, J\}$
`explore(I)`	A, B, C, F, H	$\{A, B, C, F, H, G, E, I, J\}$
`explore(H)`	A, B, C, F	$\{A, B, C, F, H, G, E, I, J\}$
`explore(F)`	A, B, C	$\{A, B, C, F, H, G, E, I, J\}$
`explore(C)`	A, B	$\{A, B, C, F, H, G, E, I, J\}$
`explore(B)`	A	$\{A, B, C, F, H, G, E, I, J\}$
`explore(A)`	A, D	$\{A, B, C, F, H, G, E, I, J\}$
`explore(D)`	A, D, L	$\{A, B, C, F, H, G, E, I, J, D\}$
`explore(L)`	A, D	$\{A, B, C, F, H, G, E, I, J, D, L\}$
`explore(D)`	A	$\{A, B, C, F, H, G, E, I, J, D, L\}$
`explore(A)`	`Done`	$\{A, B, C, F, H, G, E, I, J, D, L\}$

Since it is possible that G has multiple components (where some vertex is not reachable by another vertex), we we can define another method `DFS` that solves the problem of graphs not being connected (`explore` can only explore things that are reachable from the starting vertex v)

```
1 Function DFS(G = (V, E)):
2     for v ∈ V do
3         if ¬visited(v) then
4             explore(v)
```

We can also modify the DFS algorithm to label what component each vertex corresponds to. We could make a modification to the DFS algorithm that initializes some global variable and increments each time DFS iterates again through V and sees a vertex not previously seen.

Then, the explore method could label each vertex with that corresponding global variable value. Why does this work? When we run DFS on an undirected graph, each call of `explore` explores an entire component at once. Hence, when itertting through V and we see a vertex that is not in the visited set already, this must belong to a brand new component.

Either way, the runtime of DFS is $\mathcal{O}(|V| + |E|)$ as we will visit each vertex once and each edge between vertices once as well, which is represented by $|V|$ and $|E|$ respectively.

1.2 DFS on Directed Graph

Although we cannot as easily discuss components with a directed graph (we will discuss strongly connected components after the exam), we can use DFS to determine if a graph is acylic (this has some cool properties!).

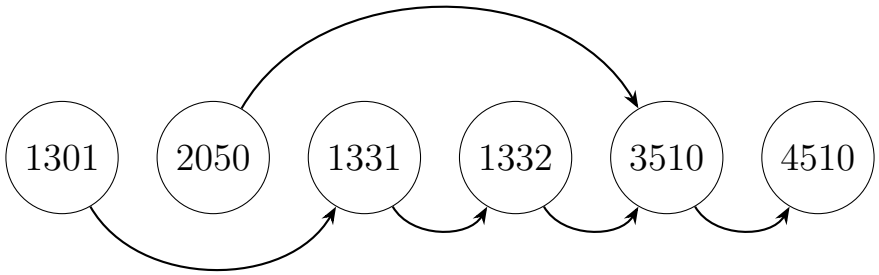
Before talking about DFS, why is it useful for a graph to be acyclic?

1.2.1 Class Scheduling

An example of a topic that naturally has a directed acyclic graph is course pre-requirements. For example, we know the following prereqs:

$$1331 \to 1332$$
$$1301 \to 1331$$
$$1332 \to 3510$$
$$2050 \to 3510$$
$$3510 \to 4510$$

Since this is directed and acylic, we can run a topological sort to get the following graph which can be very useful as we now can make a schedule of classes that we can take in a specific order:



The topologically sorted graph has a lot of nice properties nice such as all the edges pointing to the right (nothing goes back to the left). Now, let's generalize this! How do we first find that a tree is acyclic?

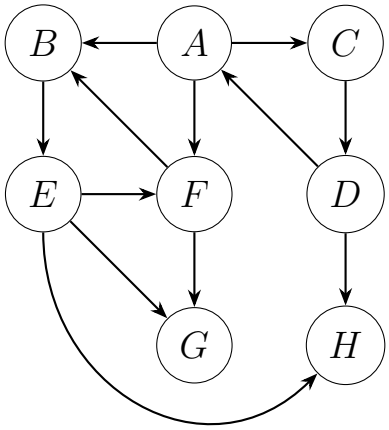
We can modify our original `explore` algorithm with the following:

```
1 Function explore(v):
2     visited(v) ← true
3     v_pre = clock()
4     for (v, x) ∈ E do
5         if ¬visited(x) then
6             explore(x)
7     v_post = clock()
```

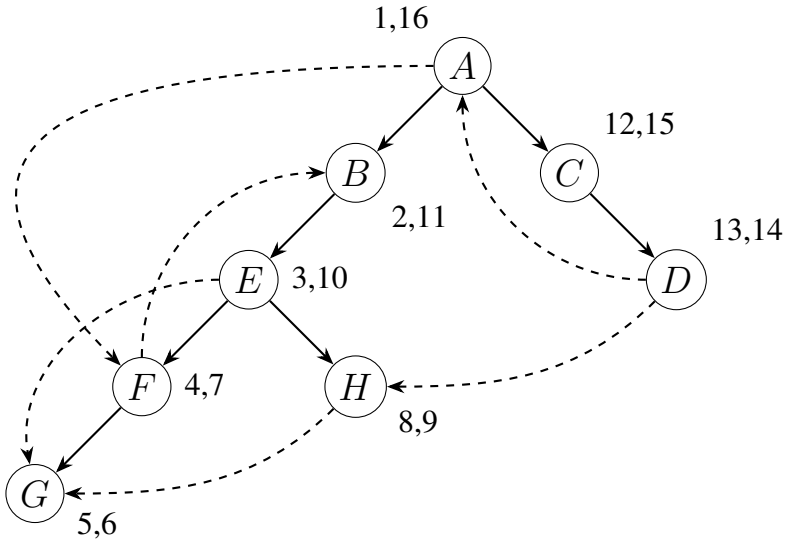
This code uses a `clock` method which can be defined as

```
1 Function clock():
2     clock ← clock + 1 return clock
```

Keep in mind, this new `explore` method is coupled with a new DFS method that also initializes a global `count` variable. This modified DFS essentially records numbers associated with each vertex. For any vertex v discovered during `explore`, it will find values $Pre(v)$ and $Post(v)$. Let's do an example:



We can then generate the following DFS tree, see the table on the next page for how this is generated.



Function Call	Pre/Post Numbers	Clock	Stack
`explore(A)`	$pre = 1$	2	A, B
`explore(B)`	$pre = 2$	3	A, B, E
`explore(E)`	$pre = 3$	4	A, B, E, F
`explore(F)`	$pre = 4$	5	A, B, E, F, G
`explore(G)`	$pre = 5$	6	A, B, E, F, G
`explore(G)`	$\boxed{pre = 5, post = 6}$	7	A, B, E, F
`explore(F)`	$\boxed{pre = 4, post = 7}$	8	A, B, E
`explore(E)`	$pre = 2$	8	A, B, E, H
`explore(H)`	$pre = 8$	9	A, B, E, H
`explore(H)`	$\boxed{pre = 8, post = 9}$	10	A, B, E
`explore(E)`	$\boxed{pre = 2, post = 10}$	11	A, B
`explore(B)`	$\boxed{pre = 2, post = 11}$	12	A
`explore(A)`	$pre = 1$	12	A, C
`explore(C)`	$pre = 12$	13	A, C, D
`explore(D)`	$pre = 13$	14	A, C, D
`explore(D)`	$\boxed{pre = 13, post = 14}$	15	A, C
`explore(C)`	$pre = 12, 15$	16	A
`explore(A)`	$\boxed{pre = 1, post = 16}$	17	`Done`

The boxed numbers in the above table are the same pre/post numbers from the DFS tree presented above (and shows how these numbers are arrived at).

Within this graph, there are 3 types of edges:

1. Forward edge: ancestor to a descendant such as $B \to G$ or $A \to F$
2. Backward edge: descendant to an ancestor such as $F \to B$
3. Cross edge: an edge between two nodes that are not related such as $D \to H$

Within these edges, we have certain properties for pre and post numbers.

- If $X \to Y$ is a forward edge, $Pre(X) < Pre(Y) < Post(Y) < Post(X)$
- If $X \to Y$ is a backward edge, $Pre(Y) < Pre(X) < Post(X) < Post(Y)$
- If $X \to Y$ is a cross edge, $Pre(Y) < Post(Y) < Pre(X) < Post(X)$

In particular, this is useful because if (u, v) is an edge and $Post(v) > Post(u)$ then (u, v) is a backward edge.

Theorem. *If DFS finds a backwards edge, G has a cycle.*

How do we actually determine if DFS finds a backwards edge? We can run DFS first and collect pre/post numbers and for each edge $(u, v) \in E$ then we check if $Post(v) > Post(u)$. If so, then this is a backwards edge.

1.3 Topological Sorting

We are interested in being able to topologically sort a DAG (directed acyclic graph). Since there is a DAG, then the graph is acylic meaning we can sort the vertices by their post number and put highest post number to the left.

This is additional $\mathcal{O}(|V| \log |V|)$ work since we are sorting the post numbers of which there are $|V|$ total work.