●●●● Not Frequent 0/6

Introduction to Graphs

Authors: Darren Yao, Benjamin Qi
Contributors: Nathan Gong, Ryan Chou, Kevin Sheng, Nikhil Chatterjee

What graphs are.

Language: Python Edit This Page

TABLE OF CONTENTS

Introduction

Graphs can be used to represent many things, from images to wireless signals, but one of the simplest analogies is to a map. Consider a map with several cities and bidirectional roads connecting the cities. Some problems relating to graphs are:

1. Is city A connected to city B? Consider a region to be a group of cities such that each city in the group can reach any other city in said group, but no other cities. How many regions are in this map, and which cities are in which region? (USACO Silver)
2. What's the shortest distance I have to travel to get from city A to city B? (USACO Gold)

For USACO Bronze, it suffices to learn the basics of how graphs are represented (usually **adjacency lists**).

RESOURCES

CSA	★ **Introduction to Graphs**	interactive	⋮
CSA	★ **Graph Representations**	interactive - adjacency lists and matrices	⋮
CSA	★ **Graph Editor**	use this tool to visualize your own graphs	⋮
CPH	★ **11 - Basics of Graphs**	graph terminology, representation	⋮
IUSACO	**10.1 to 10.3 - Graph Theory**	graph basics and representation, trees	⋮
PAPS	**6.4 - Graphs**	adjacency matrices, lists, maps	⋮

Constructing Adjacency Lists

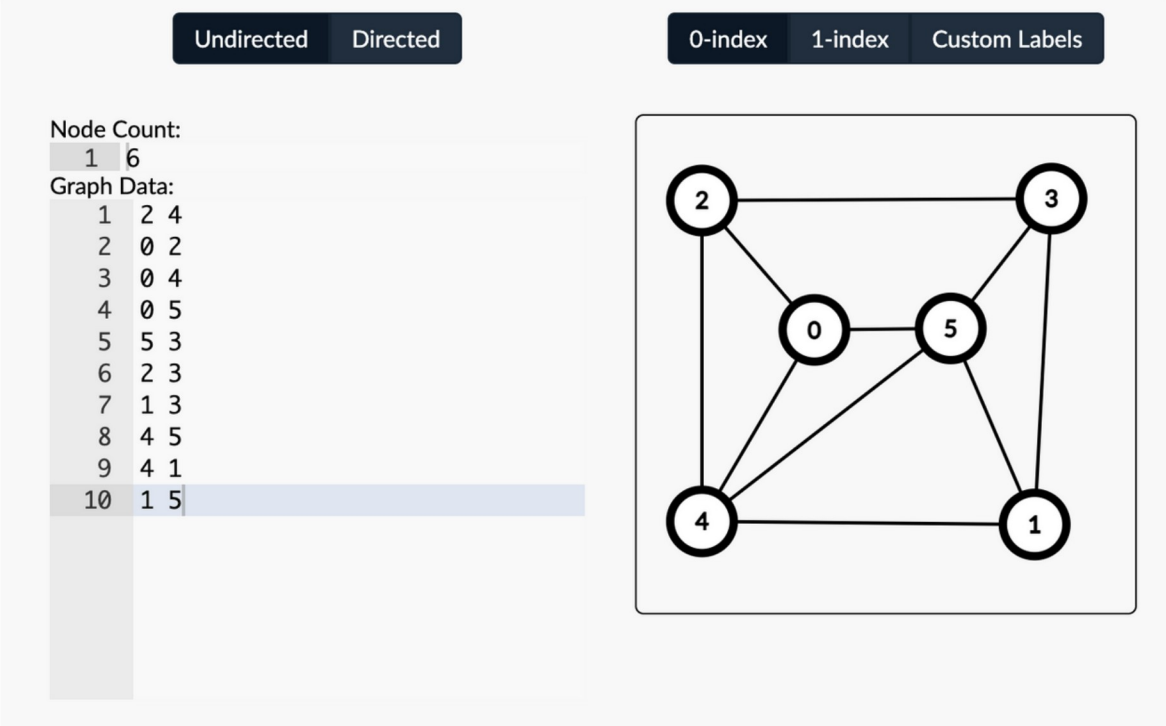
Graphs are often given as input in the following format:

- The first line will contain the number of nodes N and the number of edges M.
- Then follow M lines, each containing a pair of integers specifying an edge of the graph.

For example, the undirected graph given in the **CSAcademy resource from above** would be represented as the following input:

```
6 10
2 4
0 2
0 4
0 5
5 3
2 3
1 3
4 5
4 1
1 5
```

and could be visualized in the **CSAcademy graph editor**:



The following code represents the graph using adjacency lists. Once we have the graph in this representation, it is easy to print the number of neighbors of a node, or iterate over the neighbors of a node.

Copy PYTHON

```python
N, M = map(int, input().split())
adj = [[] for _ in range(N)]

for i in range(M):
    u, v = map(int, input().split())
    adj[u].append(v)
    adj[v].append(u)

u = 1
# print number of vertices adjacent to u
print("deg(u) =", len(adj[u]))
# print all edges with u as an endpoint
for v in adj[u]:
```

```
14      print("{" + str(u) + ", " + str(v) + "}")
```

Output:

```
deg(u) = 3
{1, 3}
{1, 4}
{1, 5}
```

What Does a Bronze Graph Problem Look Like?

All of the problems below fall into at least one of the following two categories:

- The graph's structure is special (it's a tree, path, or a cycle).
- To solve the problem, all you need to do is iterate over the adjacency list of every vertex.

In addition, knowledge of **Silver-level graph topics** is usually helpful but not strictly required to solve the problem.

Livestock Lineup

Livestock Lineup

Bronze - Hard

Focus Problem – try your best to solve this problem before continuing! *View Internal Solution*

While the intended solution is to brute force all possible permutations of the cows in $\mathcal{O}(N \cdot C!)$ time, we can solve the problem in just $\mathcal{O}(C)$ time if we represent the constraints using a graph.

Solution Using Graphs

> **Warning!**
>
> This explanation, and the implementation that follows, assume that all constraints in the input describe distinct pairs of cows. If this assumption did not hold, you would first need to remove duplicate constraints.

Notice that since the input is guaranteed to be valid, we're always going to end up with "chains" of cows that we can arrange as we please. Using the sample given in the problem, we'd get a "chain" representation like this:

Note that cows that are not part of any chain can be considered their own chains of length 1 for implementation purposes.

With this representation in mind, we can iterate through the cows in lexicographical (alphabetical) order. When we visit a cow that could be a possible start of a chain (a cow that has at most one required neighbor), we repeatedly go through its neighbors, adding the cows we visit to the ordering, until we hit an end.

Implementation

Time Complexity: $\mathcal{O}(C)$

Copy PYTHON

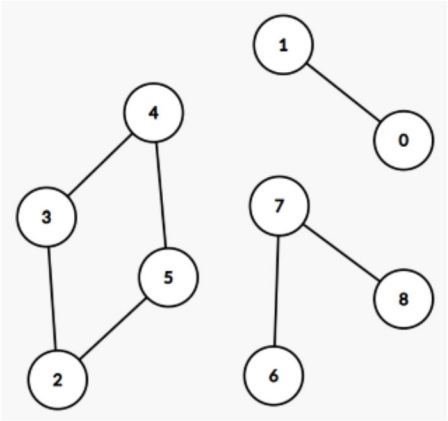
```python
COWS = sorted(
    ["Bessie", "Buttercup", "Belinda", "Beatrice", "Bella", "Blue", "Betsy", "Sue"]
)

cow_inds = {c: i for i, c in enumerate(COWS)}
```

```
6
7  neighbors = [[] for _ in range(len(COWS))]
8  with open("lineup.in") as read:
9      for _ in range(int(read.readline())):
10         words = read.readline().strip().split()
11
12         # Convert the names to their index in the list
13         cow1 = cow_inds[words[0]]
14         cow2 = cow_inds[words[-1]]
15         neighbors[cow1].append(cow2)
16         neighbors[cow2].append(cow1)
17
18 order = []
19 added = [False for _ in range(len(COWS))]
20 for c in range(len(COWS)):
21     """
22     Check that:
23     1. This cow hasn't already been added yet.
24     2. This cow could be the possible start of a chain.
25     """
26     if not added[c] and len(neighbors[c]) <= 1:
27         added[c] = True
28         order.append(c)
29
30         # If the chain length > 1, we keep on going
31         if len(neighbors[c]) == 1:
32             prev = c
33             at = neighbors[c][0]
34             while len(neighbors[at]) == 2:
35                 added[at] = True
36                 order.append(at)
37                 a, b = neighbors[at]
38                 at, prev = b if a == prev else a, at
39
40             # Add the final element
41             added[at] = True
42             order.append(at)
43
44 with open("lineup.out", "w") as out:
45     for c in order:
46         print(COWS[c], file=out)
```

Check Your Understanding

How many connected components are in the following graph?



1. 1

2. 2