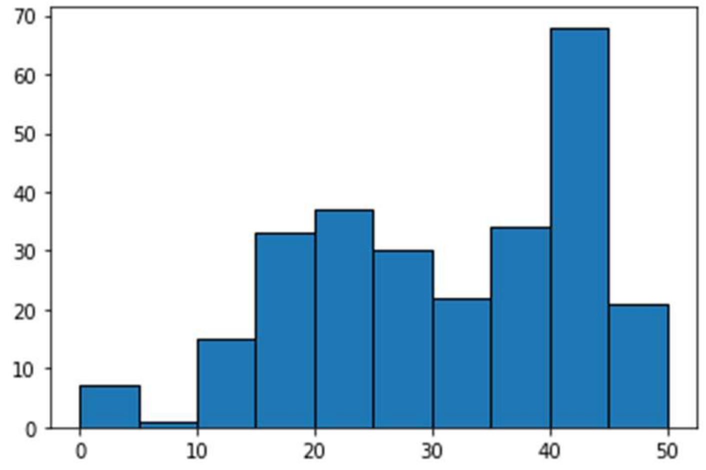
Admin

- Teaching evaluation is closing soon, do it on NuKu.
- Test1 marks are released
- Mean total: 30.6   Q1: 13, Q2: 14,   Q3: 3
- If you want to check the marking, go to CO358
- No hand back



Admin

- Week 6:
  - Term test 2
  - In-person marking of Assign 1
    - Your signed up tutorial time
    - CO241
- Week 7
  - Assign2 is due

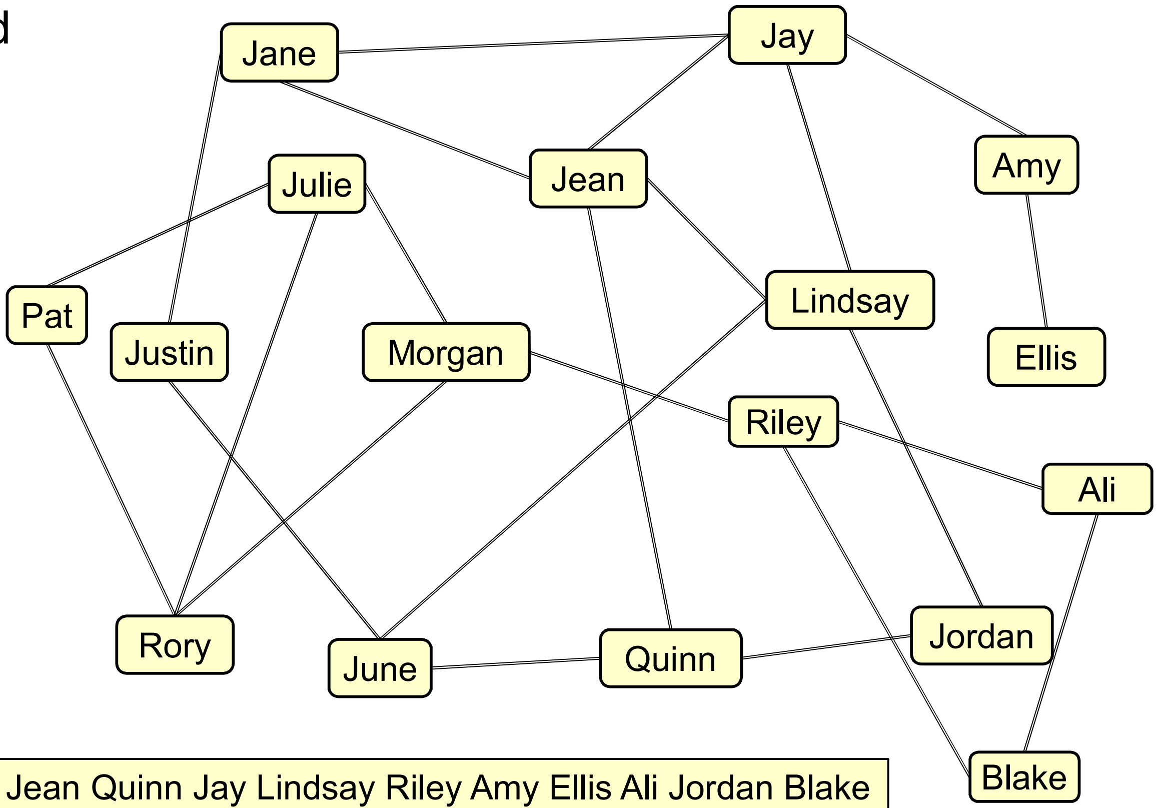
- Time management during the break

Assign 2

- Part 1, A*
  - Most of the graph structure is done: multi-graph, directed, only outgoing edges
  - Add walking edges, remove walking edges
  - A*: edge length is straight-line distance between two stops, heuristic: straight line distance to goal
- Part 2
  - Set up the UI
  - Modify the graph: single graph, directed, both in edges and out edges
  - Assume you can found a map: node with its component no
  - Draw it using different colour and display the map
- Part 3
  - UI,
  - Graph: single graph, undirected, do not care much about edges, only care neighbour nodes
  - Assume you find a list of nodes,
  - Draw them using different colours and display them
- Part4: do everything if you can.

Connected Components

- If the graph is not connected, how do we find the connected components?



Collection of nodes:

Pat Justin Jane Rory Julie Morgan June Jean Quinn Jay Lindsay Riley Amy Ellis Ali Jordan Blake

Connected Components (Undirected Graphs)

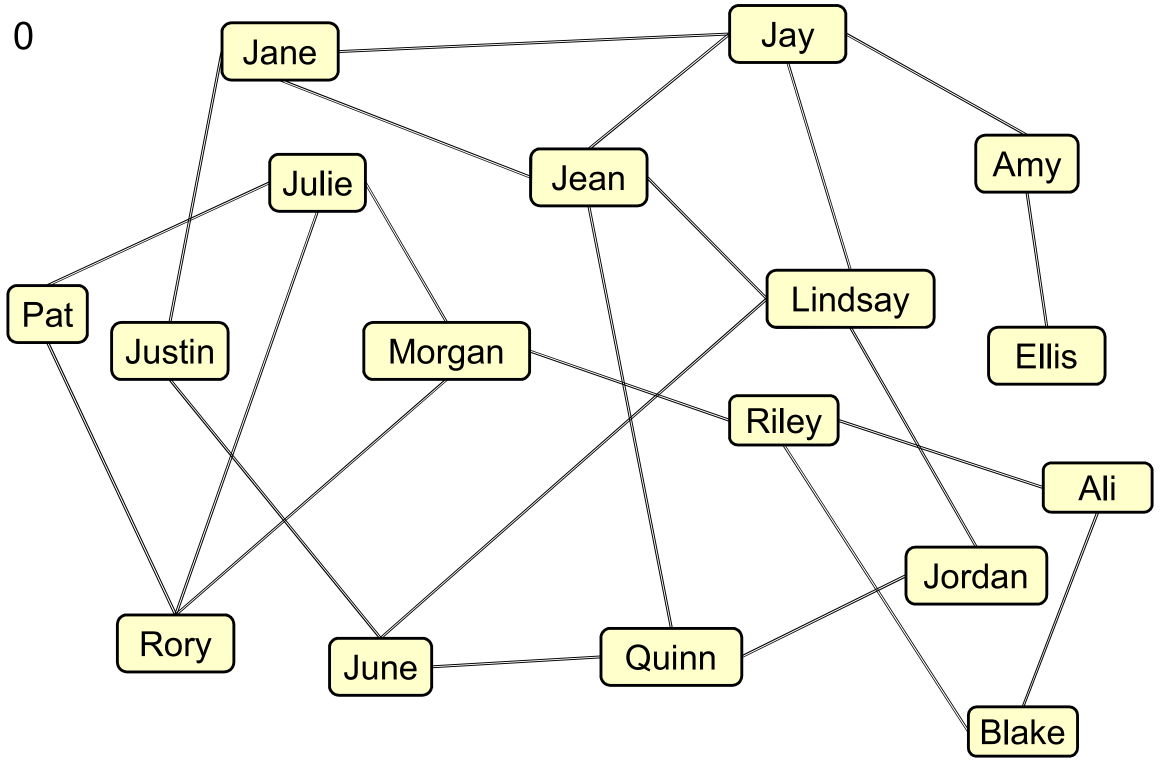
- Goal: Label each node of an undirected graph with the id of its component.
- Assume that for all nodes, node.component is initially -1.

```
ConnectedComponents (graph)
    componentNum ← 0;
    for each node in nodes:
        if node.component = -1:                 // ie, node is not visited
            TraverseComponent(node, componentNum)
            componentNum++

TraverseComponent(node, componentNum)
    node.component ← componentNum
    for each neighbour of node:
        if  neighbour.component = -1:               // ie, neighbour is not visited
            TraverseComponent(neighbour, componentNum)
```

Example of ConnectedComponents

componentNum: 0

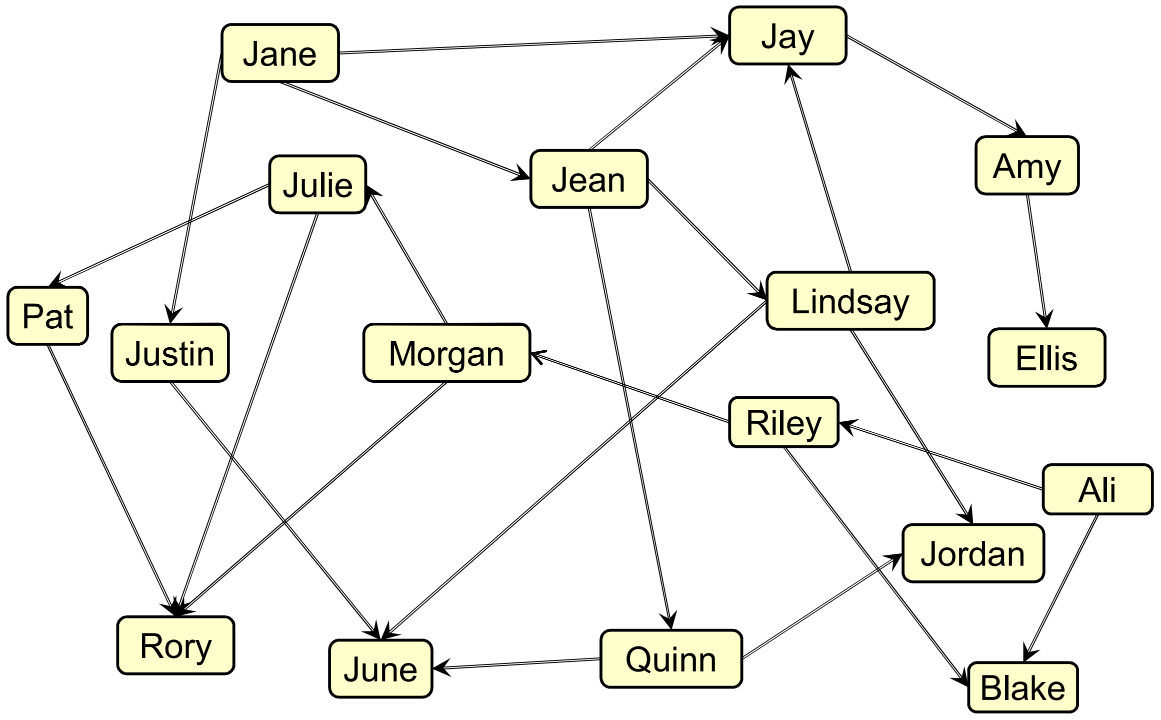


Collection of nodes:

Pat Justin Jane Rory Julie Morgan June Jean Quinn Jay Lindsay Riley Amy Ellis Ali Jordan Blake

Connected Components: directed graphs.

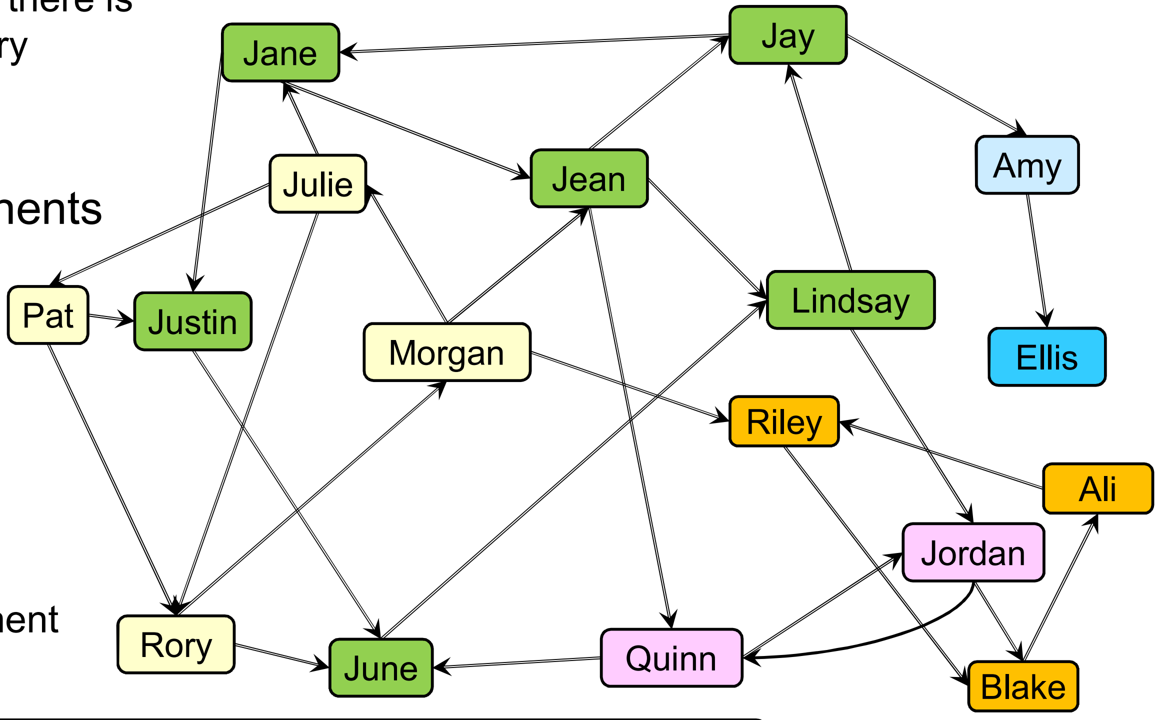
- Why doesn't the algorithm work on directed graphs?



Pat Justin Jane Rory Julie Morgan June Jean Quinn Jay Lindsay Riley Amy Ellis Ali Jordan Blake

Connected Components in Directed Graphs

- A **strongly connected component** of a directed graph:
  - a maximal set of nodes where there is a path from every node to every other node.
- Strongly connected components may not be disconnected from each other!
  - There may be a path from a node in a component to a node outside, **but not back,**
  - There may be a path from a node outside a component to a node inside, **but not back.**



How do we find Strongly Connected Components?

Kosuraja's Algorithm: Strongly Connected Components

```
Kosuraja(graph):
    for each node in graph:
         node.component ← -1                            // initialize nodes to not be in a component
    componentNum ← 0
    nodeList ← empty list;
    visited ← empty set

    for each node in graph:
        if node is not visited then
            ForwardVisit(node, nodeList, visited)    // traverse graph from node forward along edges,
                                                     // adding nodes to nodeList in post-order

    for each node in nodeList in reverse order:
        if node.component = -1 then
            BackwardVisit(node, componentNum)  // traverse graph from node backward along edges
            componentNum++                     // marking nodes with the component number
```

Kosuraja's Algorithm: Strongly Connected Components

```
// Search forward from node, putting node on nodeList after visiting everything it can get to.

ForwardVisit(node, nodeList, visited)
   if node is not in visited then
      add node to visited.
      for each neighbour in node.outNeighbours:
         ForwardVisit(neighbour, nodeList, visited)
      add node to nodeList.

// Search backwards from node, marking all the nodes than can get to it as the same component

BackwardVisit(node, componentNum)
   if node.component = -1 then
      node.component ← componentNum
      for each backNeighbour in node.inNeighbours:
         BackwardVisit(backNeighbour, componentNum).
```

Kosuraja's Algorithm

$$\begin{tikzcd}
 & A \arrow[r] \arrow[dr] & B \arrow[d] & S \arrow[r] & R \arrow[d] & & \\
E & D \arrow[u] \arrow[l] & C \arrow[l] \arrow[r] & P \arrow[u] & Q \arrow[l] \arrow[r] & T \arrow[r] & U
\end{tikzcd}$$

NodeList: