Algorithms in Systems Engineering
ISE 172

Lecture 17

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Sections 7.1-7.3
- References
  - CLRS Chapter 21 and 22
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Connectivity Relations

- So far, we have only considered sets of items that are related to each other through some kind of ordering (if at all).
- In other words, two items x and y are only related by their relative positions in the ordered list.
- We will now generalize this idea by considering additional *connectivity relationships* between items.
- To do so, we will specify that there is a direct link between certain pairs of items.
- This will allow us to ask questions such as the following.
  - Is x connected "directly" to y?
  - Is x connected to y "indirectly," i.e., through a sequence of direct connections?
  - What is the set of of all items connected to x, directly or indirectly?
  - What is the shortest number of connections needed to get from x to y?

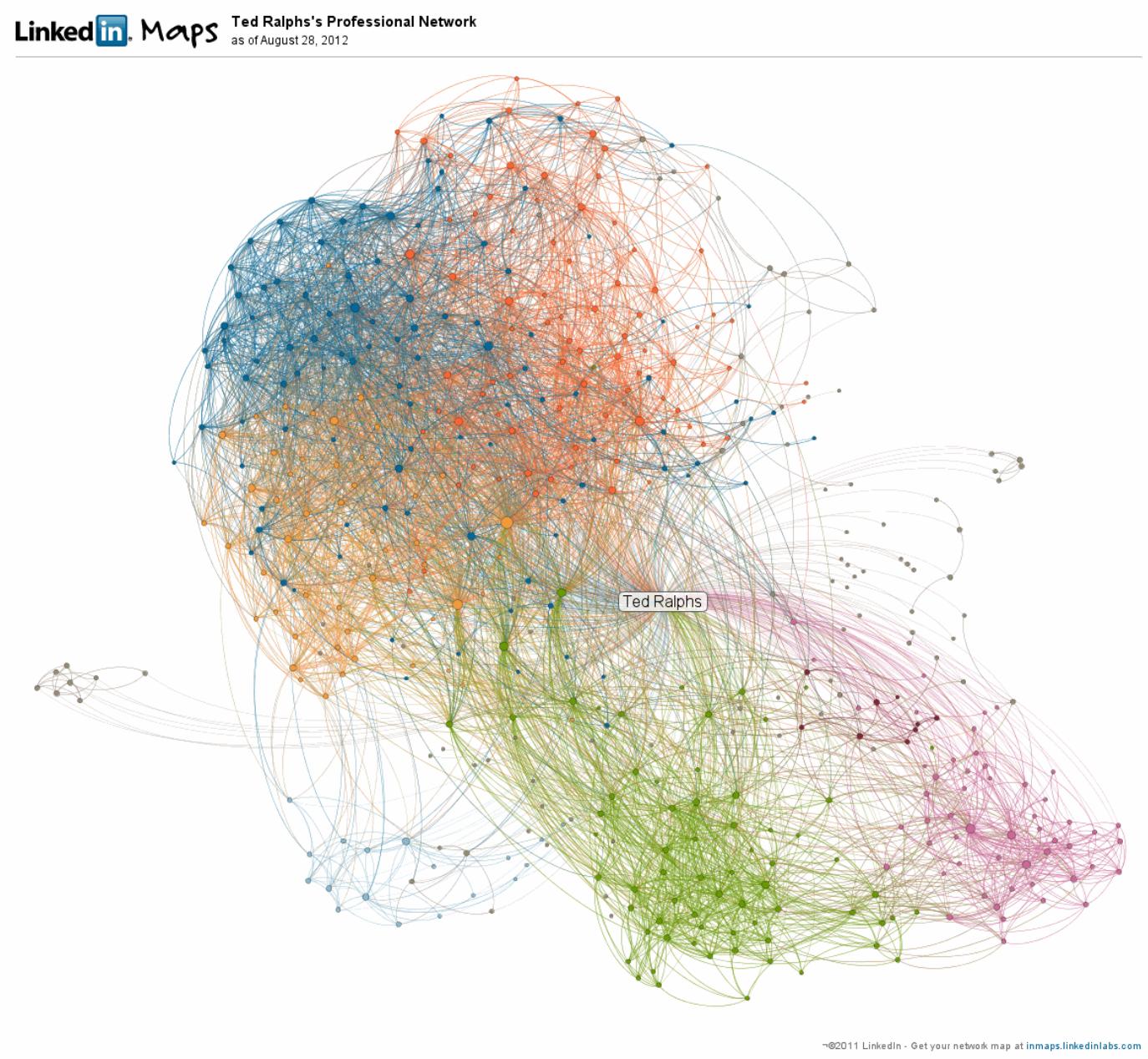
Graphs

- A *graph* is an abstract object used to model such connectivity relations.
- A graph consists of a list of items, along with a set of connections between the items.
- The study of such graphs and their properties, called *graph theory*, is hundreds of years old.
- Graphs can be visualized easily by creating a physical manifestation.
- There are several variations on this theme.
  - The connections in the graph may or may not have an *orientation* or a *direction*.
  - We may not allow more than one connection between a pair of items.
  - We may not allow an item to be connected to itself.
- For now, we consider graphs that are
  - *undirected*, i.e., the connections do not have an orientation, and
  - *simple*, i.e., we allow only one connection between each pair of items and no connections from an item to itself.

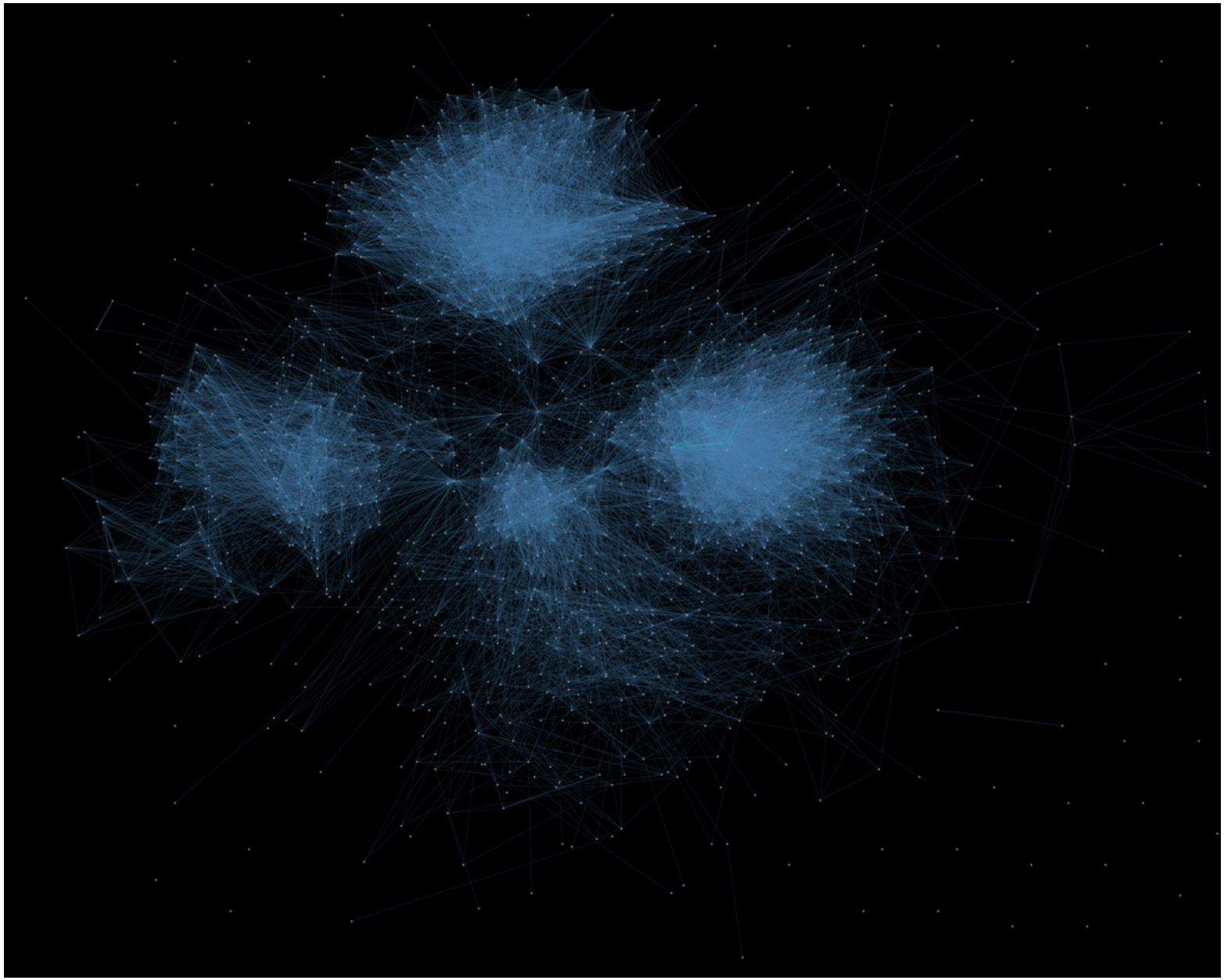
Applications of Graphs

- Maps
- Social Networks
- World Wide Web
- Circuits
- Scheduling
- Communication Networks
- Matching and Assignment

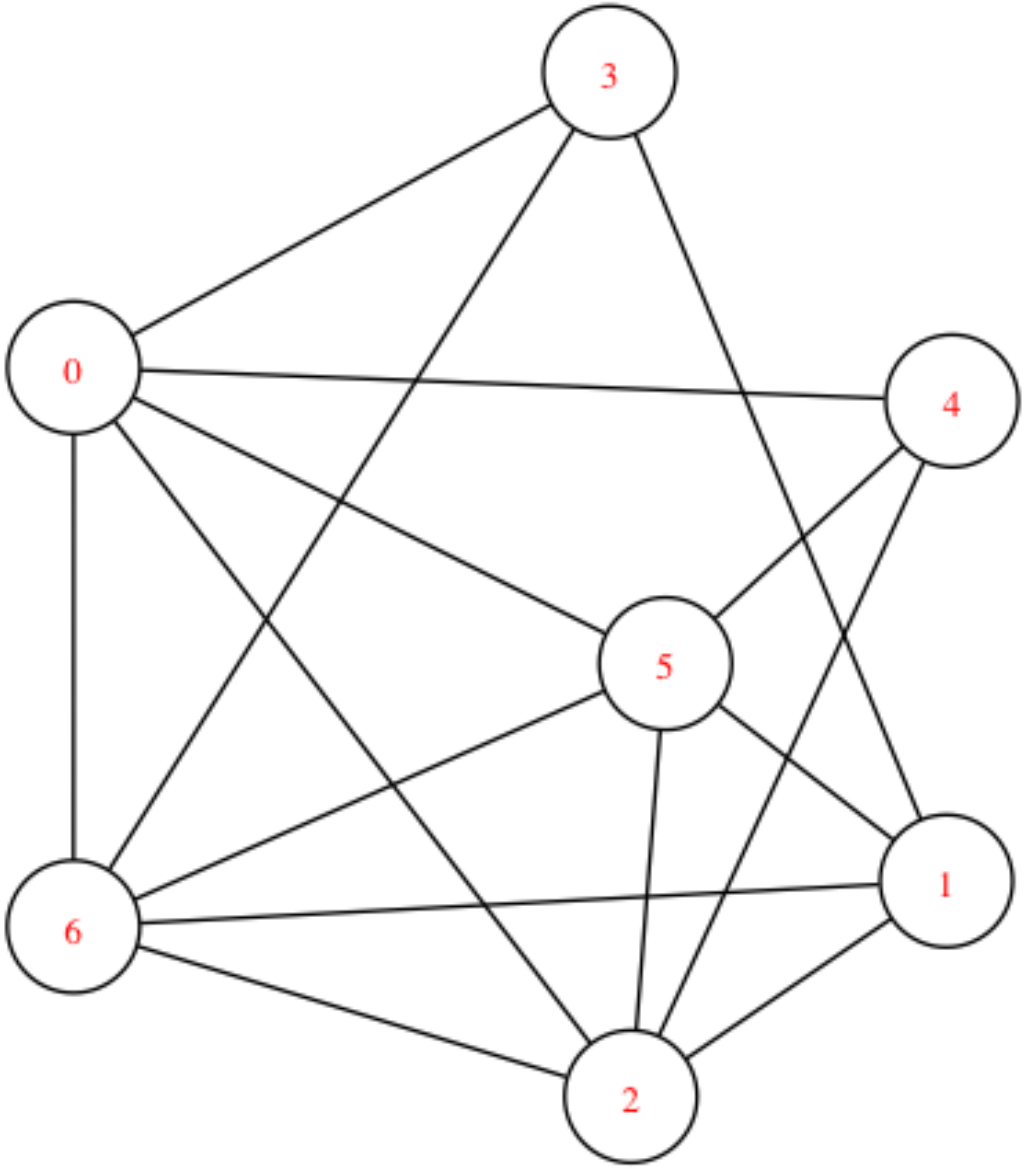
Example Graph (Social Network)



A Facebook Graph



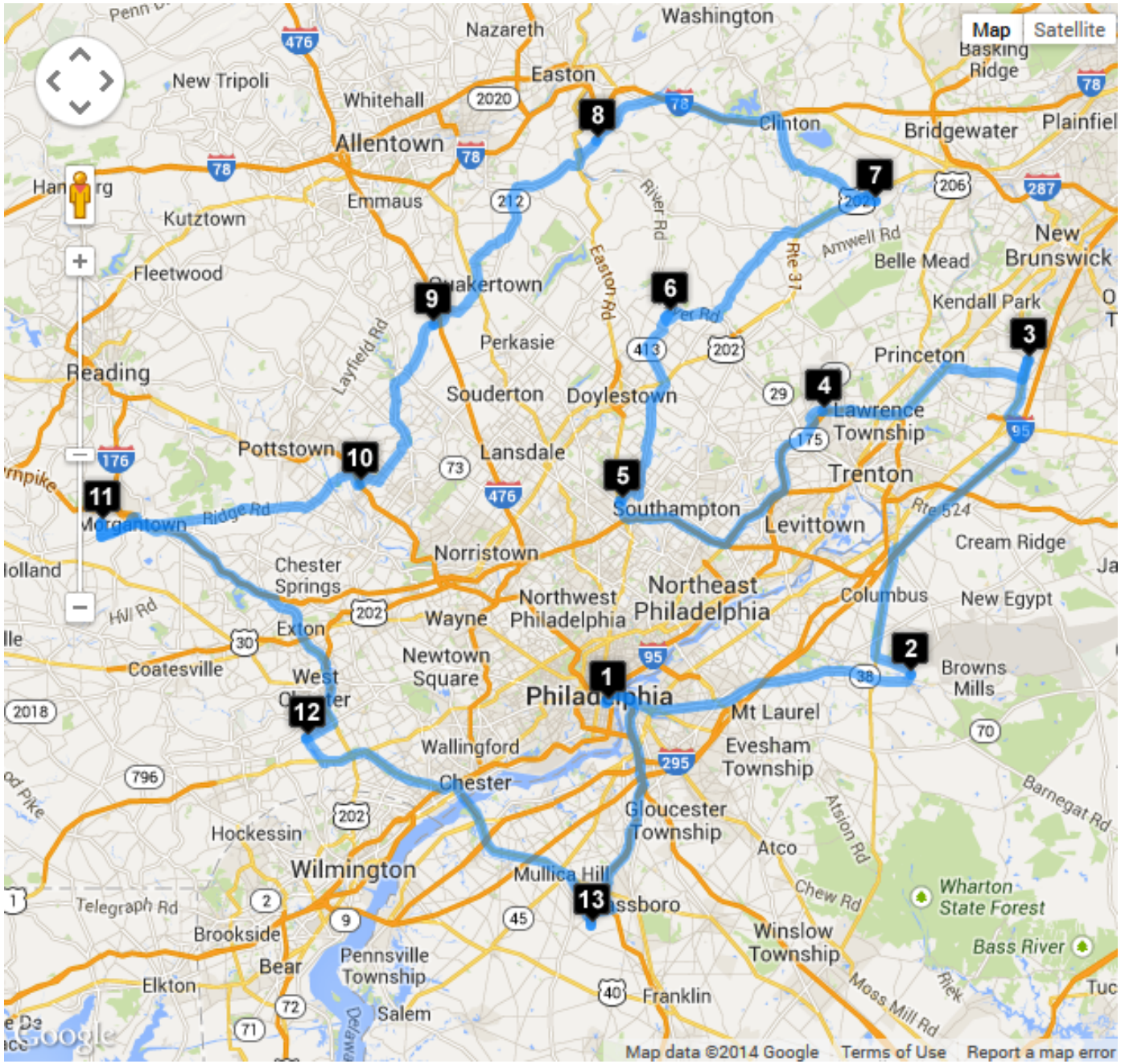
Example of an Abstract Graph



Graph Terminology and Notation

- In an undirected graph, the "items" are usually called *vertices* (sometimes also called *nodes*).
- The set of vertices is denoted V and the vertices are indexed from 0 to $n-1$, where $n = |V|$.
- The connections between the vertices are unordered pairs called *edges*.
- The set of edges is denoted E and $m = |E| \leq n(n-1)/2$.
- An undirected graph $G = (V, E)$ is then composed of a set of vertices V and a set of edges $E \subseteq V \times V$.
- If $e = \{i, j\} \in E$, then
  - i and j are called the *endpoints* of e,
  - e is said to be *incident* to i and j, and
  - i and j are said to be *adjacent* vertices and are also called *neighbors*.

A Tour of a Graph Made from Map Data



More Terminology

- Let $G = (V, E)$ be an undirected graph.
- A *subgraph* of G is a graph composed of an edge set $E' \subseteq E$ along with all incident vertices.
- A subset V' of V, along with all incident edges is called an *induced subgraph*.
- A *path* in G is a sequence of vertices such that each vertex is adjacent to the vertex preceding it in the sequence.
- A path is *simple* if no vertex occurs more than once in the sequence.
- A *cycle* is a path that is simple except that the first and last vertices are the same.
- A *tour* is a cycle that includes all the vertices.

Operations on Graphs

- What are the basic operations we might want to perform on a graph?

Graph API

```
class Graph:
    def __init__(self, type):
        self.type = type

    def get_node_list(self)
    def get_edge_list(self)
    def add_node(self, v)
    def del_node(self, v)
    def add_edge(self, v, w)
    def del_edge(self, v, w)
    def __str__(self
```

Node Class

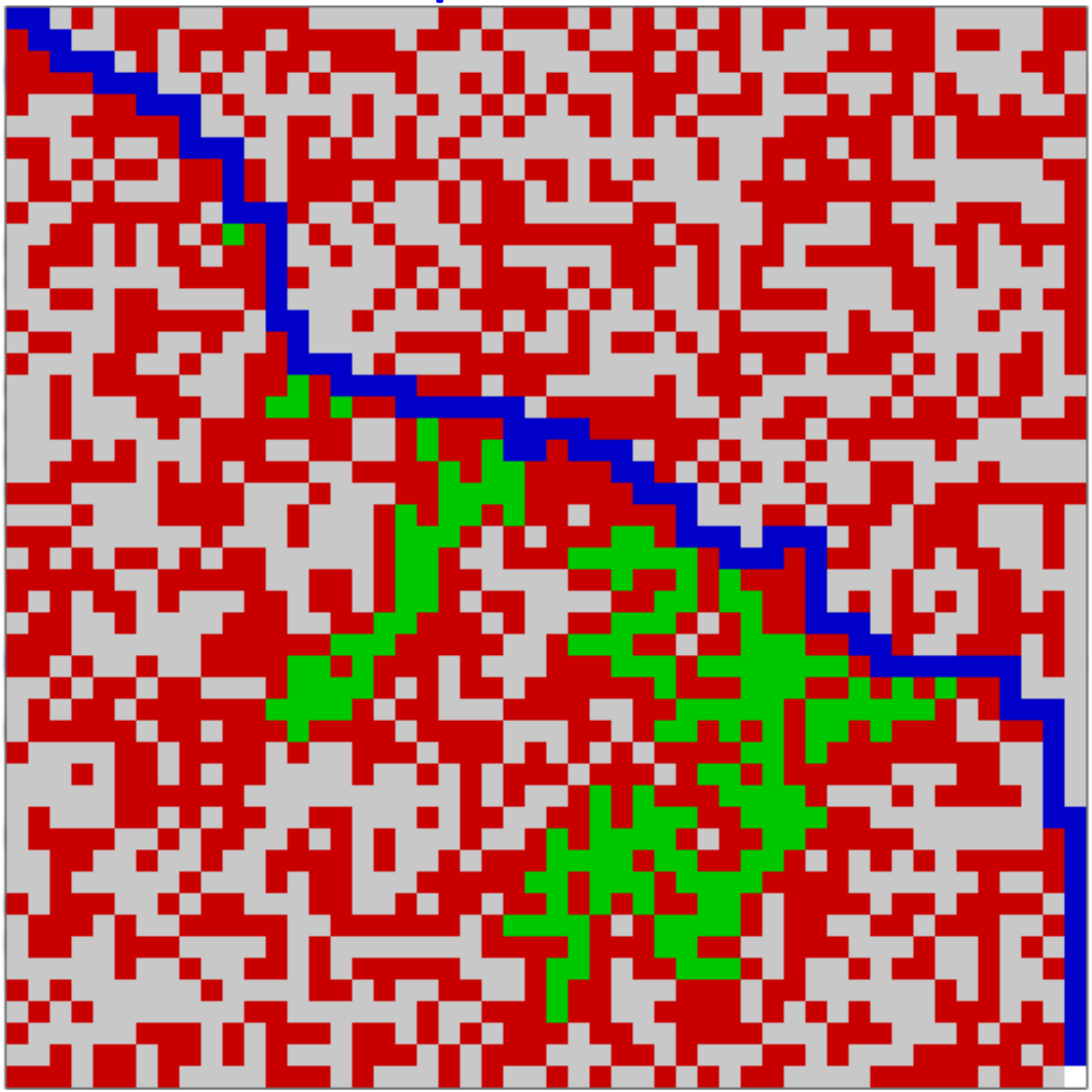
```
class Node:
    def __init__(self, name):
        self.name = i

    def get_neighbors(self):
    __str__(self)
```

- This is a generalization of the `Node` class from linked lists in which we have not just a single "next" node, but a collection of them.
- In other words, a linked list is a special kind of graph.
- What other places have we already seen graphs?

The Maze Lab

- In the maze lab, we were actually working with a graph.
- The "nodes" were the empty cells (the ones without walls).
- We had an "edge" anywhere there was two adjacent empty cells.
- We tried to find a path from the "entrance node" to the "exit node".
- This is very similar to the kinds of problem we will want to solve on more general graphs.
- Note that we stored the graph in the maze lab implicitly by storing the location of the nodes.
- The existence of edges was checked by applying rules about which nodes are connected according their location on the grid.

A Graph from a Maze



Trees as Graphs

- In graph terminology, a *tree* is a connected graph with no cycles and a *forest* is a graph consisting of a collection of trees.
- Properties of trees
  - Every tree has exactly $n - 1$ edges.
  - In a tree, there is a unique path from any given vertex to any other vertex.
- A tree that has a specified *root vertex* is called a rooted tree.
  - In a rooted tree, there is a unique path from the root to every other vertex.
  - We can therefore uniquely define the parent of a vertex v as the vertex that immediately precedes it on the path from the root to v.
  - Hence, we are justified in thinking of trees in the way that we had previously, as a set of hierarchical relationships between the vertices.

Data Structures for Undirected Graphs

- To support these basic graph operations, we need a data structure to store the graph.
- As with many previous data structures, there are generally two different ways to compactly represent a graph (with many variations).
  - Ajacency matrix: An implementation based on arrays.
  - Adjacency lists: An implementation based on linked lists.
- We have to analyze the tradeoffs between these two representations, as we have before.

Adjacency Matrix Implementation

- Consider an undirected graph $G = (V, E)$.
- The *adjacency matrix* A of G is an $n \times n$ symmetric $0 - 1$ matrix constructed as follows:

$$A_{ij} = A_{ji} = \begin{cases} 1 & \text{if } \{i, j\} \in E, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

- How do we implement the `Graph` class using an adjacency matrix?

Efficiency of Adjacency Matrices

- A fundamental operation we will to be able to perform efficiently is to find out which nodes are neighbors of a particular given node.
- How easily can we do this with an adjacency matrix?
- What is the running time of this basic operation?
- Can we do better?

Adjacency Lists Implementation

- The *adjacency list* for node i is a linked list of all other nodes adjacent to i in the graph.
- What we are essentially doing is compressing one row of the adjacency matrix by storing just the locations of the nonzero entries.
- Since most rows are extremely sparse, this is very advantageous.
- Note that adjacency lists do not have to be in any particular order.
- An adjacency list representation of a graph consists of an adjacency list for each node in the graph.
- How do we implement the `Graph` class using an adjacency lists?

Comparing the Implementations

- How does the adjacency list implementation compare to the adjacency matrix implementation?
  - Efficiency of basic operations
  - Memory requirements

A Client Function for Printing a Graph

- Here's an example of a standard way in which the graph interface class is used.
- Here, we print out a graph by enumerating all the edges incident to each vertex.

```
def print(G):
    for n in G.get_node_list():
        print n, ":",
        for i in n.get_neighbors():
            print i
        print
```