CSE 332 Data Abstractions: Graphs and Graph Traversals

Kate Deibel
Summer 2012
Some course stuff and a humorous story

GRADES, MIDTERMS, AND IT SEEMED LIKE A GOOD IDEA
The Midterm

It was too long—I admit that

If it helps, this was the first exam I have ever written

Even still, I apologize
You All Did Great

I am more than pleased with your performances on the midterm

The points you missed were clearly due to time constraints and stresses

You showed me you know the material

Good job!
How Grades are Calculated

Many (if not most) CSE major courses use curving to determine final grades

- Homework and exam grades are used as indicators and are adjusted as necessary

- Example:
  A student who does excellent on homework and projects (and goes beyond) will get a grade bumped up even if his/her exam scores are poorer
My Experiences as a Teacher

Timed exams are problematic

- Some of the best students I have known did not do great on exams

The more examples of student work that one sees, the more learning becomes evident

- Even partial effort/incomplete work tells a lot
- Unfortunately, this means losing points

The above leads to missing points

- All students (even myself back in the day) care about points
My Repeated Mistake

As a teacher, I should talk more about how points get transformed into a final grade

I learned this lesson my first year as a TA...

... and indirectly caused the undergraduate CSE servers to crash
It Seemed Like a Good Idea at the Time

At the annual CS education conference (SIGCSE), there is a special panel about teaching mistakes and learning from them.

My Promises

I know you will miss points

If you do the work in the class and put in the effort, you will earn more than a passing grade

As long as you show evidence of learning, you will earn a good grade regardless
**What This Means For You**

Keep up the good work

Do not obsess over points

The final will be less intense
That was fun but you are here for learning...

BACK TO CSE 332 AND GRAPH [THEORY]
Where We Are

We have learned about the essential ADTs and data structures:
- Regular and Circular Arrays (dynamic sizing)
- Linked Lists
- Stacks, Queues, Priority Queues
- Heaps
- Unbalanced and Balanced Search Trees

We have also learned important algorithms
- Tree traversals
- Floyd's Method
- Sorting algorithms
Where We Are Going

Less generalized data structures and ADTs

More on algorithms and related problems that require constructing data structures to make the solutions efficient

Topics will include:

- Graphs
- Parallelism
Graphs

A graph is a formalism for representing relationships among items

- Very general definition
- Very general concept

A graph is a pair: \( G = (V, E) \)
- A set of vertices, also known as nodes: \( V = \{v_1, v_2, \ldots, v_n\} \)
- A set of edges \( E = \{e_1, e_2, \ldots, e_m\} \)
  - Each edge \( e_i \) is a pair of vertices \( (v_j, v_k) \)
  - An edge "connects" the vertices

Graphs can be **directed** or **undirected**
A Graph ADT?

We can think of graphs as an ADT

- Operations would include isEdge(v_j,v_k)
- But it is unclear what the "standard operations" would be for such an ADT

Instead we tend to develop algorithms over graphs and then use data structures that are efficient for those algorithms

Many important problems can be solved by:

1. Formulating them in terms of graphs
2. Applying a standard graph algorithm
Some Graphs

For each example, what are the vertices and what are the edges?

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites

Core algorithms that work across such domains is why we are CSE
Scratching the Surface

Graphs are a powerful representation and have been studied deeply.

Graph theory is a major branch of research in combinatorics and discrete mathematics.

Every branch of computer science involves graph theory to some extent.
To make formulating graphs easy and standard, we have a lot of *standard terminology* for graphs.
**Undirected Graphs**

In **undirected graphs**, edges have no specific direction

- Edges are always "two-way"

Thus, \((u, v) \in E\) implies \((v, u) \in E\).

- Only one of these edges needs to be in the set
- The other is implicit, so normalize how you check for it

**Degree** of a vertex: number of edges containing that vertex

- Put another way: the number of adjacent vertices
**Directed Graphs**

In directed graphs (or digraphs), edges have direction. Thus, \((u, v) \in E\) does not imply \((v, u) \in E\).

Let \((u, v) \in E\) mean \(u \rightarrow v\)

- Call \(u\) the **source** and \(v\) the **destination**
- **In-Degree** of a vertex: number of in-bound edges (edges where the vertex is the destination)
- **Out-Degree** of a vertex: number of out-bound edges (edges where the vertex is the source)

2 edges here
Self-Edges, Connectedness

A self-edge a.k.a. a loop edge is of the form \((u, u)\)
- The use/algorithm usually dictates if a graph has:
  - No self edges
  - Some self edges
  - All self edges

A node can have a(n) degree / in-degree / out-degree of zero

A graph does not have to be connected
- Even if every node has non-zero degree
- More discussion of this to come
More Notation

For a graph $G = (V, E)$:
- $|V|$ is the number of vertices
- $|E|$ is the number of edges
  - Minimum?
  - Maximum for undirected?
  - Maximum for directed?

If $(u, v) \in E$, then $v$ is a neighbor of $u$ (i.e., $v$ is adjacent to $u$)
- Order matters for directed edges: $u$ is not adjacent to $v$ unless $(v, u) \in E$
More Notation

For a graph $G = (V, E)$:

- $|V|$ is the number of vertices
- $|E|$ is the number of edges
  - Minimum: 0
  - Maximum for undirected: $|V||V+1|/2 \in O(|V|^2)$
  - Maximum for directed: $|V|^2 \in O(|V|^2)$

If $(u, v) \in E$, then $v$ is a neighbor of $u$ (i.e., $v$ is adjacent to $u$)

- Order matters for directed edges: $u$ is not adjacent to $v$ unless $(v, u) \in E$
Examples Again

Which would use directed edges?
Which would have self-edges?
Which could have 0-degree nodes?

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites
Weighted Graphs

In a weighted graph, each edge has a weight or cost

- Typically numeric (ints, decimals, doubles, etc.)
- Orthogonal to whether graph is directed
- Some graphs allow negative weights; many do not

![Diagram of weighted graph](attachment:weighted_graph_diagram.png)
Examples Again

What, if anything, might weights represent for each of these?
Do negative weights make sense?

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites
Paths and Cycles

We say "a path exists from $v_0$ to $v_n$" if there is a list of vertices $[v_0, v_1, ..., v_n]$ such that $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$.

A cycle is a path that begins and ends at the same node ($v_0 == v_n$)

Example path (that also happens to be a cycle): [Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle]
Path Length and Cost

Path length: Number of edges in a path
Path cost: Sum of the weights of each edge

Example where

P = [Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle]

length(P) = 5
cost(P) = 11.5

Length is sometimes called "unweighted cost"
Simple Paths and Cycles

A simple path repeats no vertices (except the first might be the last):
[Seattle, Salt Lake City, San Francisco, Dallas]
[Seattle, Salt Lake City, San Francisco, Dallas, Seattle]

A cycle is a path that ends where it begins:
[Seattle, Salt Lake City, Seattle, Dallas, Seattle]

A simple cycle is a cycle and a simple path:
[Seattle, Salt Lake City, San Francisco, Dallas, Seattle]
Example:

- Is there a path from A to D? No
- Does the graph contain any cycles? No
Undirected Graph Connectivity

An undirected graph is **connected** if for all pairs of vertices $u \neq v$, there exists a **path** from $u$ to $v$.

An undirected graph is **complete**, or **fully connected**, if for all pairs of vertices $u \neq v$ there exists an **edge** from $u$ to $v$.
Directed Graph Connectivity

A directed graph is **strongly connected** if there is a path from every vertex to every other vertex.

A directed graph is **weakly connected** if there is a path from every vertex to every other vertex *ignoring direction of edges*.

A directed graph is **complete or fully connected**, if for all pairs of vertices \( u \neq v \), there exists an edge from \( u \) to \( v \).
Examples Again

For undirected graphs: connected?
For directed graphs: strongly connected? weakly connected?

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites
Trees as Graphs

When talking about graphs, we say a tree is a graph that is:

- undirected
- acyclic
- connected

All trees are graphs, but NOT all graphs are trees

How does this relate to the trees we know and "love"?
Rooted Trees

We are more accustomed to rooted trees where:

- We identify a unique root
- We think of edges as directed: parent to children

Picking a root gives a unique rooted tree

- The tree is simply drawn differently and with undirected edges
Rooted Trees

We are more accustomed to rooted trees where:

- We identify a unique root
- We think of edges as directed: parent to children

Picking a root gives a unique rooted tree

- The tree is simply drawn differently and with undirected edges
Directed Acyclic Graphs (DAGs)

A **DAG** is a directed graph with no directed cycles

- Every rooted directed tree is a DAG
- But not every DAG is a rooted directed tree

- Every DAG is a directed graph
- But not every directed graph is a DAG

![DAG Diagram](image-url)
Examples Again

Which of our directed-graph examples do you expect to be a DAG?

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites
Density / Sparsity

Recall:
In an undirected graph, \(0 \leq |E| < |V|^2\)

Recall:
In a directed graph, \(0 \leq |E| \leq |V|^2\)

So for any graph, \(|E|\) is \(O(|V|^2)\)

Another fact:
If an undirected graph is connected, then \(|E| \geq |V|-1\) (pigeonhole principle)
Density / Sparsity

$|E|$ is often much smaller than its maximum size.

We do not always approximate as $|E|$ as $O(|V|^2)$
- This is a correct bound, but often not tight

If $|E|$ is $\Theta(|V|^2)$ (the bound is tight), we say the graph is dense
- More sloppily, dense means "lots of edges"

If $|E|$ is $O(|V|)$ we say the graph is sparse
- More sloppily, sparse means "most possible edges missing"
Insert humorous statement here

GRAPH DATA STRUCTURES
What’s the Data Structure?

Graphs are often useful for lots of data and questions

- Example: "What’s the lowest-cost path from x to y"

But we need a data structure that represents graphs

Which data structure is "best" can depend on:

- properties of the graph (e.g., dense versus sparse)
- the common queries about the graph ("is (u,v) an edge?" vs "what are the neighbors of node u?")

We will discuss two standard graph representations

- Adjacency Matrix and Adjacency List
- Different trade-offs, particularly time versus space
**Adjacency Matrix**

Assign each node a number from 0 to $|V| - 1$.

A $|V| \times |V|$ matrix of Booleans (or 0 vs. 1).

- Then $M[u][v] == true$ means there is an edge from $u$ to $v$.
Adjacency Matrix Properties

Running time to:

- Get a vertex’s out-edges:
- Get a vertex’s in-edges:
- Decide if some edge exists:
- Insert an edge:
- Delete an edge:

Space requirements:

Best for sparse or dense graphs?
## Adjacency Matrix Properties

### Running time to:
- Get a vertex’s out-edges: \(O(|V|)\)
- Get a vertex’s in-edges: \(O(|V|)\)
- Decide if some edge exists: \(O(1)\)
- Insert an edge: \(O(1)\)
- Delete an edge: \(O(1)\)

### Space requirements:
\(O(|V|^2)\)

Best for sparse or dense graphs? \textit{dense}
How will the adjacency matrix vary for an undirected graph?

- Will be symmetric about diagonal axis
- Matrix: Could we save space by using only about half the array?

But how would you "get all neighbors"?
Adjacency Matrix Properties

How can we adapt the representation for weighted graphs?

- Instead of Boolean, store a number in each cell
- Need some value to represent ‘not an edge’
  - 0, -1, or some other value based on how you are using the graph
- Might need to be a separate field if no restrictions on weights
Adjacency List

Assign each node a number from 0 to |V|-1

- An array of length |V| in which each entry stores a list of all adjacent vertices (e.g., linked list)
Adjacency List Properties

Running time to:

- Get a vertex’s out-edges:
- Get a vertex’s in-edges:
- Decide if some edge exists:
- Insert an edge:
- Delete an edge:

Space requirements:

Best for sparse or dense graphs?
Adjacency List Properties

Running time to:

- Get a vertex’s out-edges: \(O(d)\) where \(d\) is out-degree of vertex
- Get a vertex’s in-edges: \(O(|E|)\) (could keep a second adjacency list for this!)
- Decide if some edge exists: \(O(d)\) where \(d\) is out-degree of source
- Insert an edge: \(O(1)\) (unless you need to check if it’s already there)
- Delete an edge: \(O(d)\) where \(d\) is out-degree of source

Space requirements: \(O(|V|+|E|)\)

Best for sparse or dense graphs? \textit{sparse}
Undirected Graphs

Adjacency lists also work well for undirected graphs with one caveat

- Put each edge in two lists to support efficient "get all neighbors"
Which is better?

Graphs are often sparse
- Streets form grids
- Airlines rarely fly to all cities

Adjacency lists should generally be your default choice
- Slower performance compensated by greater space savings
Might be easier to list what isn't a graph application...

APPLICATIONS OF GRAPHS: TRAVERSALS
What’s the *shortest way* to get from Seattle to Pullman?
What’s the *fastest way* to get from Seattle to Pullman?
If Wenatchee’s phone exchange goes down, can Seattle still talk to Pullman?
If Tacomas’s phone exchange goes down, can Olympia still talk to Spokane?
If we’re at 3rd and Pine, how can we get to 1st and University using Metro? How about 4th and Seneca?
Graph Traversals

For an arbitrary graph and a starting node v, find all nodes reachable from v (i.e., there exists a path)

- Possibly "do something" for each node (print to output, set some field, return from iterator, etc.)

Related Problems:
- Is an undirected graph connected?
- Is a digraph weakly/strongly connected?
  - For strongly, need a cycle back to starting node
Graph Traversals

Basic Algorithm for Traversals:
- Select a starting node
- Make a set of nodes adjacent to current node
- Visit each node in the set but "mark" each nodes after visiting them so you don't revisit them (and eventually stop)
- Repeat above but skip "marked nodes"
traverseGraph(Node start) {
    Set pending = emptySet();
    pending.add(start)
    mark start as visited
    while(pending is not empty) {
        next = pending.remove()
        for each node u adjacent to next
            if(u is not marked) {
                mark u
                pending.add(u)
            }
    }
}
Running Time and Options

Assuming add and remove are $O(1)$, entire traversal is $O(|E|)$ if using an adjacency list.

The order we traverse depends entirely on how add and remove work/are implemented.

- **DFS**: a stack "depth-first graph search"
- **BFS**: a queue "breadth-first graph search"

DFS and BFS are "big ideas" in computer science.

- **Depth**: recursively explore one part before going back to the other parts not yet explored.
- **Breadth**: Explore areas closer to start node first.
Recursive DFS, Example with Tree

A tree is a graph and DFS and BFS are particularly easy to "see" in one order.

Order processed: A, B, D, E, C, F, G, H

- This is a "pre-order traversal" for trees
- The marking is unneeded here but because we support arbitrary graphs, we need a means to process each node exactly once.

```java
DFS(Node start) {
    mark and process start
    for each node u adjacent to start
        if u is not marked
            DFS(u)
}
```
DFS with Stack, Example with Tree

Order processed: A, C, F, H, G, B, E, D

- A different order but still a perfectly fine traversal of the graph

DFS2(Node start) {
  initialize stack s to hold start
  mark start as visited
  while(s is not empty) {
    next = s.pop() // and "process"
    for each node u adjacent to next
    if(u is not marked)
      mark u and push onto s
  }
}
A "level-order" traversal
DFS/BFS Comparison

BFS always finds the shortest path (or "optimal solution") from the starting node to a target node

- Storage for BFS can be extremely large
- A $k$-nary tree of height $h$ could result in a queue size of $k^h$

DFS can use less space in finding a path

- If longest path in the graph is $p$ and highest out-degree is $d$ then DFS stack never has more than $d \cdot p$ elements
Implications

For large graphs, DFS is hugely more memory efficient, if we can limit the maximum path length to some fixed $d$.

If we knew the distance from the start to the goal in advance, we could simply not add any children to stack after level $d$.

But what if we don’t know $d$ in advance?
Iterative Deepening (IDFS)

Algorithms

- Try DFS up to recursion of K levels deep.
- If fails, increment K and start the entire search over.

Performance:

- Like BFS, IDFS finds shortest paths.
- Like DFS, IDFS uses less space.
- Some work is repeated but minor compared to space savings.
Saving the Path

Our graph traversals can answer the standard \textit{reachability} question:

"Is there a path from node $x$ to node $y$?"

But what if we want to actually output the path?

Easy:

- Store the previous node along the path:
  When processing $u$ causes us to add $v$ to the search, set $v.path$ field to be $u$
- When you reach the goal, follow path fields back to where you started (and then reverse the answer)
- What's an easy way to do the reversal? A Stack!!
Example using BFS

What is a path from Seattle to Austin?

- Remember marked nodes are not re-enqueued
- Note shortest paths may not be unique

![Graph diagram]

- Seattle
- San Francisco
- Salt Lake City
- Dallas
- Chicago
- Austin

Distance:
- Seattle to San Francisco: 1
- San Francisco to Salt Lake City: 1
- Salt Lake City to Dallas: 2
- Dallas to Chicago: 1
- Seattle to Austin: 3
Topological Sort

Problem: Given a DAG $G = (V, E)$, output all the vertices in order such that if no vertex appears before any other vertex that has an edge to it.

Example input:

```
CSE 142, CSE 143, CSE 311, CSE 331, CSE 332, CSE 312, CSE 341, CSE 351, CSE 333, CSE 440, CSE 352
```

Example output:

```
142, 126, 143, 311, 331, 332, 312, 341, 351, 333, 440, 352
```

Disclaimer: Do not use for official advising purposes! (Implies that CSE 332 is a pre-req for CSE 312 – not true)
Questions and Comments

Terminology:
A DAG represents a partial order and a topological sort produces a total order that is consistent with it.

Why do we perform topological sorts only on DAGs?
- Because a cycle means there is no correct answer.

Is there always a unique answer?
- No, there can be one or more answers depending on the provided graph.

What DAGs have exactly 1 answer?
- Lists.
**Uses Topological Sort**

Figuring out how to finish your degree

Computing the order in which to recalculate cells in a spreadsheet

Determining the order to compile files with dependencies

In general, use a dependency graph to find an allowed order of execution
Topological Sort: First Approach

1. Label each vertex with its in-degree
   - Think "write in a field in the vertex"
   - You could also do this with a data structure on the side

2. While there are vertices not yet outputted:
   a) Choose a vertex $v$ labeled with in-degree of 0
   b) Output $v$ and "remove it" from the graph
   c) For each vertex $u$ adjacent to $v$, decrement in-degree of $u$
      - (i.e., $u$ such that $(v,u)$ is in $E$)
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440

Removed?

In-deg:
**Example**

![Graph Diagram]

**Output:**

Node: 126 142 143 311 312 331 332 333 341 351 352 440

Removed?

In-deg: 0 0 2 1 2 1 1 2 1 1 1 1
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed? x
In-deg: 0 0 2 1 2 1 1 2 1 1 1 1 1 1

Output: 126
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440

Removed? x x

In-deg: 0 0 2 1 2 1 1 2 1 1 1 1

Output: 126 142
Example

Node:  126 142 143 311 312 331 332 333 341 351 352 440
Removed?  x  x  x
In-deg:  0  0  2  1  2  1  1  1  2  1  1  1  1
        1  0  0  0  0  0  0  0

Output:  
126
142
143
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed? x x x x x
In-deg: 0 0 2 1 2 1 1 2 1 1 1 1 1
       1 0 1 0 0 0 0 0 0 0 0 0 0

Output: 126 142 143 311
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed? x x x x x x
In-deg: 0 0 2 1 2 1 1 2 1 1 1 1

Output: 126 142 143 311 331
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440

Removed? x x x x x x x x

In-deg: 0 0 2 1 2 1 1 1 2 1 1 1 1

Output: 126 142 143 311 331 332
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed? x x x x x x x x x
In-deg: 0 0 2 1 2 1 1 2 1 1 1 1 1 1
1 0 1 0 0 1 0 0 0 0
0 0

Output:
126
142
143
311
331
332
312
Example

Node:  126 142 143 311 312 331 332 333 341 351 352 440

Removed? x  x  x  x  x  x  x  x  x  x

In-deg:  0  0  2  1  2  1  1  2  1  1  1  1

Output:  
126
142
143
311
331
332
312
341

MATH 126
Example

Node:  126 142 143 311 312 331 332 333 341 351
Removed?: x  x  x  x  x  x  x  x  x  x  x  x
In-deg:  0  0  2  1  2  1  1  2  1  1  1  1
         1  0  1  0  0  1  0  0  0  0  0
         0  0  0

Output:
  126
  142
  143
  311
  331
  332
  312
  341
  351

CSE 142 → CSE 143 → CSE 311 → CSE 331 → CSE 440
CSE 142 → CSE 143 → CSE 341 → CSE 332 → ...
CSE 142 → CSE 143 → CSE 351 → CSE 333 → CSE 352

MATH 126
Example

Output:
126
142
143
311
331
332
312
341
351
352
333

Node: 126 142 143 311 312 331 332 333 341 351 352 440
Removed? x x x x x x x x x x x
In-deg: 0 0 2 1 2 1 1 1 2 1 1 1 1 1 1
Example

Node: 126 142 143 311 312 331 332 333 341 351 352 353 440

Removed? x x x x x x x x x x x x

In-deg: 0 0 2 1 2 1 1 2 1 1 1 1

Output: 126 352
        142
        143
        311
        331
        332
        312
        341
        351
        352
        333

July 23, 2012 CSE 332 Data Abstractions, Summer 2012
Example

Node:  126  142  143  311  312  331  332  333  341  351  352  440

Removed?  x  x  x  x  x  x  x  x  x  x  x  x

In-deg:  0  0  2  1  2  1  1  2  1  1  1  1
               1  0  1  0  0  1  0  0  0  0
               0  0  0  0

Output:  126  352
         142  440
         143
         311
         331
         332
         312
         341
         351
         352
         333

July 23, 2012  CSE 332 Data Abstractions, Summer 2012
What is the worst-case running time?

- Initialization $O(|V| + |E|)$ (assuming adjacency list)
- Sum of all find-new-vertex $O(|V|^2)$ (because each $O(|V|)$)
- Sum of all decrements $O(|E|)$ (assuming adjacency list)
- So total is $O(|V|^2 + |E|)$ – not good for a sparse graph!

```java
labelEachVertexWithItsInDegree();
for(i=0; i < numVertices; i++) {
    v = findNewVertexOfDegreeZero();
    put v next in output
    for each w adjacent to v
        w.indegree--;
}
```
Avoid searching for a zero-degree node every time!

- Keep the “pending” zero-degree nodes in a list, stack, queue, bag, or something that gives O(1) add/remove
- Order we process them affects the output but not correctness or efficiency

Using a queue:

- Label each vertex with its in-degree,
- Enqueue all 0-degree nodes
- While queue is not empty
  - \(v = \text{dequeue}()\)
  - Output \(v\) and remove it from the graph
  - For each vertex \(u\) adjacent to \(v\), decrement the in-degree of \(u\) and if new degree is 0, enqueue it
Running Time?

labelAllWithIndegreesAndEnqueueZeros();
for(i=0; i < numVertices; i++) {
    v = dequeue();
    put v next in output
    for each w adjacent to v {
        w.indegree--;
        if(w.indegree==0)
            enqueue(w);
    }
}

- Initialization: $O(|V| + |E|)$ (assuming adjacency list)
- Sum of all enqueues and dequeues: $O(|V|)$
- Sum of all decrements: $O(|E|)$ (assuming adjacency list)
- So total is $O(|E| + |V|)$ – much better for sparse graph!
More Graph Algorithms

Finding a shortest path is one thing

- What happens when we consider weighted edges (as in distances)?

Next time we will discuss shortest path algorithms and contributions of a curmudgeonly computer scientist