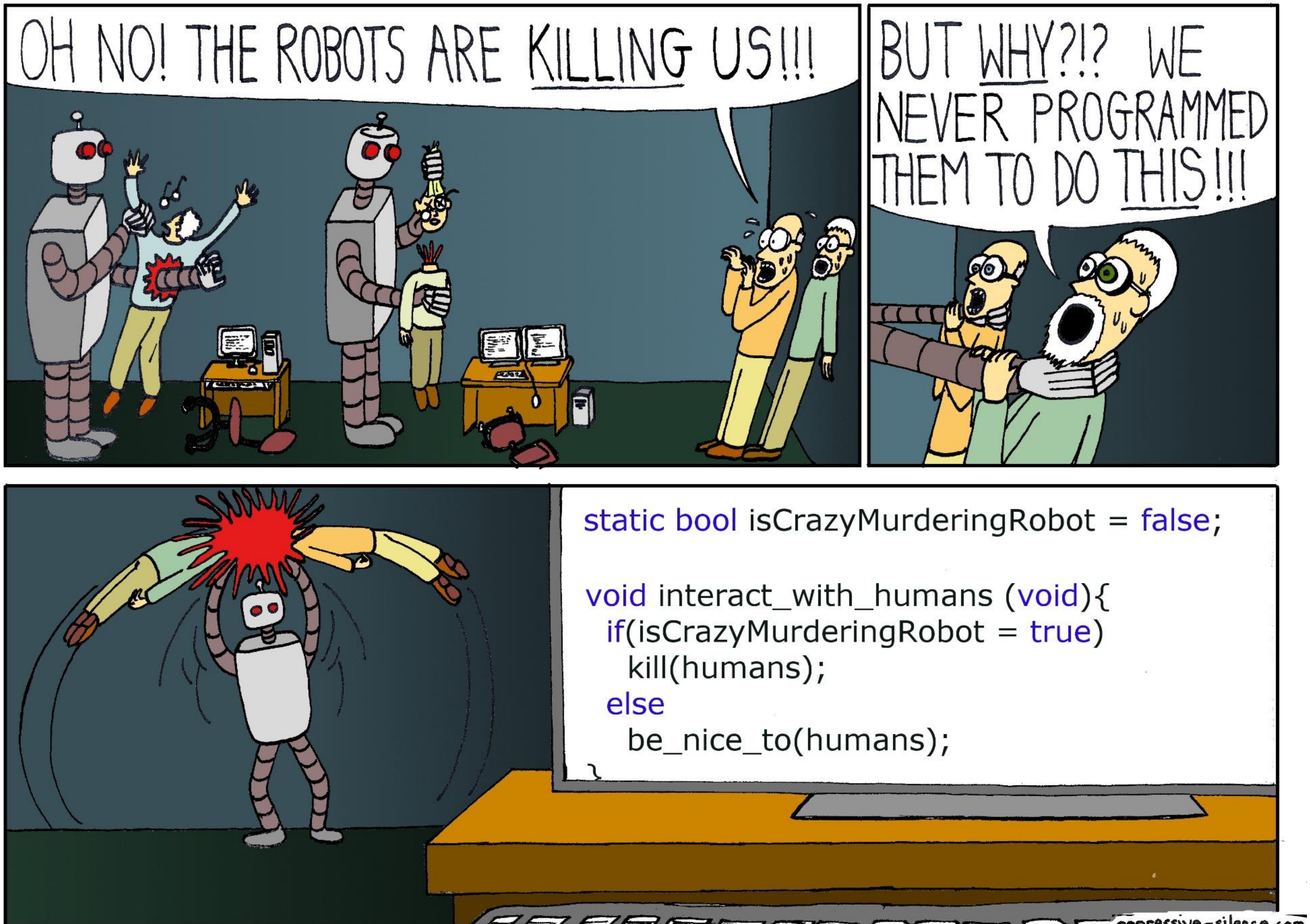
Lecture #16 - That's all folks!

- Intro to Graphs
- Graph Traversals
  - Depth-first
  - Breadth-first
- Dijkstra's Algorithm

Final Exam: Saturday, March 17th 11:30am-2:30pm

Final Exam Location: TBA

Graphs


OH NO! THE ROBOTS ARE KILLING US!!!
BUT WHY?!? WE NEVER PROGRAMMED THEM TO DO THIS!!!
static bool isCrazyMurderingRobot = false;
void interact_with_humans (void){
if(isCrazyMurderingRobot = true)
kill(humans);
else
be_nice_to(humans);
}
oppressive-silence.com

Graphs

Why should you care?

Facebook? Duh!

Not good enough?
Google+?

Computer Animation?

Google Maps?

The Internet?

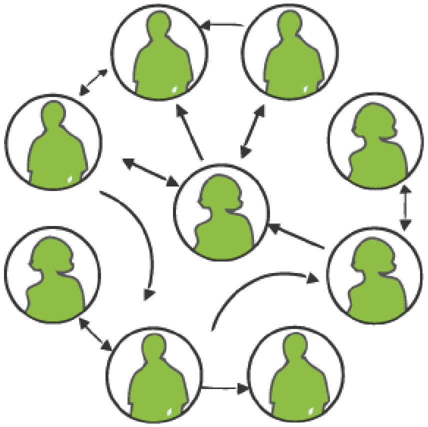
So pay attention!

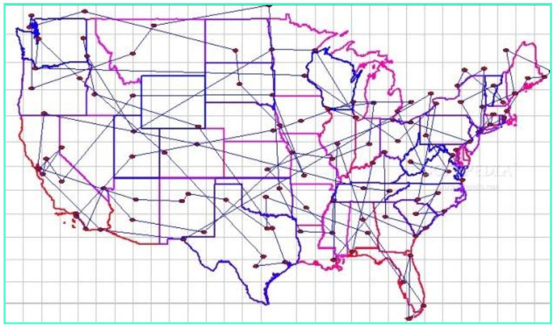
Introduction to Graphs

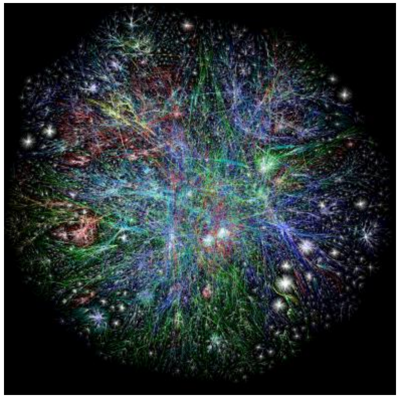
A graph is an ADT that stores a set of entities and also keeps track of the relationships between all of them.

Examples of Entities
People
Cities
Web pages

Examples of Relationships
Joe is friends with Linda
LA is 3000 miles from NYC
ucla.edu links to awesome.com





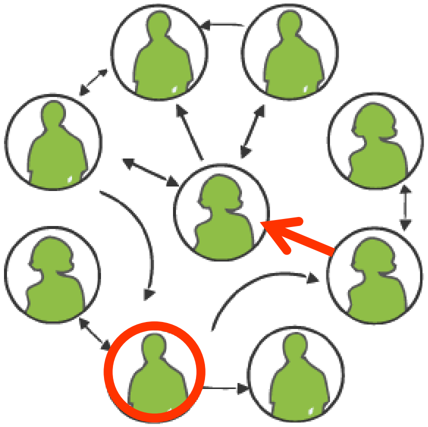


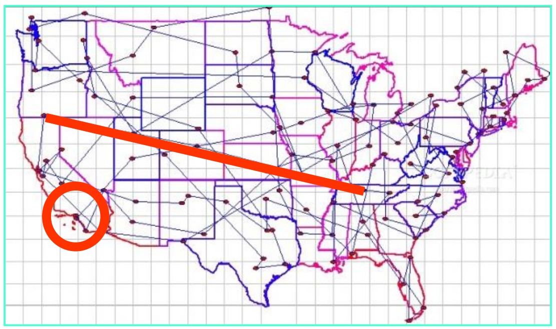
Introduction to Graphs

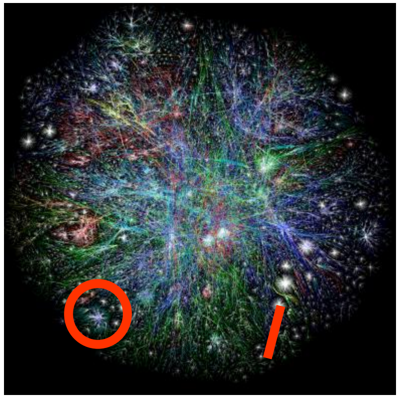
Each graph holds two types of items:

Vertices (aka Nodes):
A vertex might represent a person, a city or a web page.

Edges (aka Arcs):
An edge simply connects two* vertices to each other.







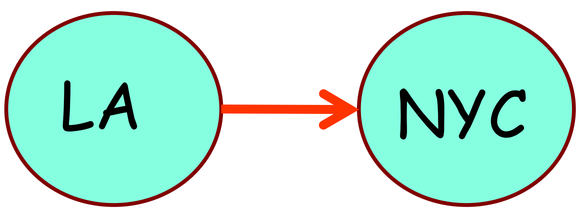
* Technically, an edge could connect a vertex to itself!

Directed vs. Undirected Graphs

There are two major types of graphs...

Directed Graph

In a directed graph, an edge goes from one vertex to another in a specific direction.

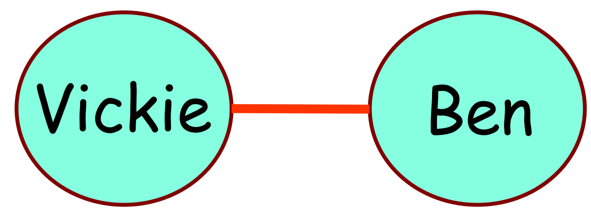


For example, above we have an edge that goes from the LA vertex to NYC vertex, but not the other way around.

(e.g., there may be a flight from LA to NYC but not the other way around)

Undirected Graph

In an undirected graph, all edges are bi-directional. You can go either way along any edge.



For example, Vickie and Ben are mutual friends on FaceBook.

(It would be kinda creepy if Vickie liked Ben, but not visa-versa)

Representing a Graph in Your Programs

The easiest way to represent a graph is with a double-dimensional array.

The size of both dimensions of the array is equal to the number of vertices in the graph.

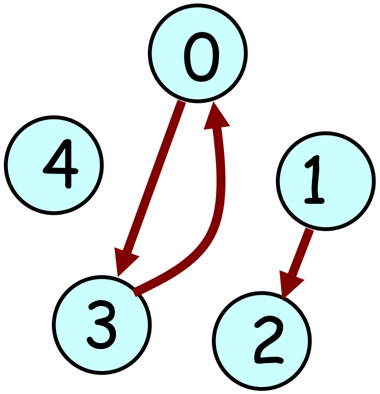
```
bool graph[5][5];
```

Each element in the array indicates whether or not there is an edge between vertex i and vertex j.

Representing a Graph in Your Programs

Each element in the array indicates whether or not there is an edge between vertex i and vertex j.

```
bool graph[5][5];

// edge from vertex 0 to vertex 3
graph[0][3] = true;
graph[1][2] = true;
graph[3][0] = true;
```



	0	1	2	3	4
0				T	
1			T		
2					
3	T				
4					

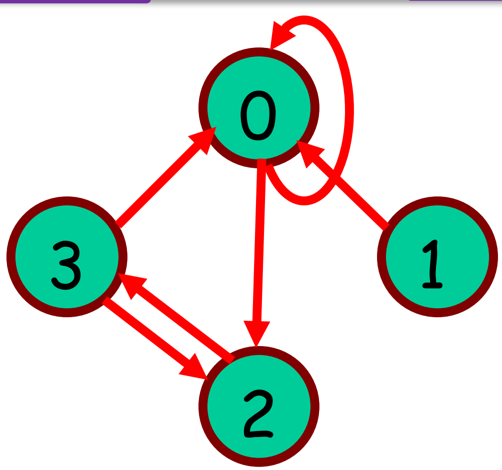
As you can see, when we set array[i][j] to true, it represents a directed edge from vertex i to vertex j.

This is called an adjacency matrix.

Representing a Graph in Your Programs

Exercise: What does the following directed graph look like?

Nodes	0	1	2	3
0	True	False	True	False
1	True	False	False	False
2	False	False	False	True
3	True	False	True	False



Representing a Graph in Your Programs

Question:

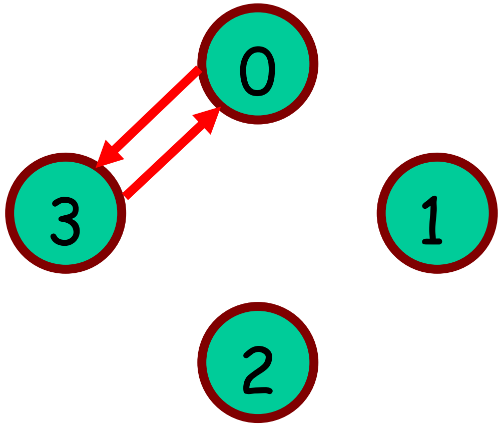
How do you represent an undirected graph with an adjacency matrix?

It's easy!

To bi-directionally connect vertices i and j, simply set array[i][j] to true and set array[j][i] to true as well!

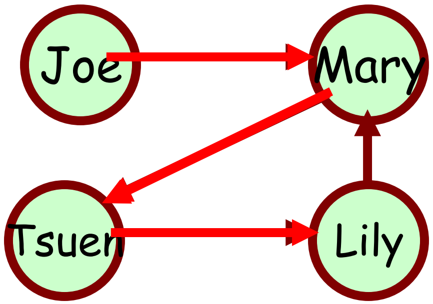
```
graph[0][3] = true;
graph[3][0] = true;
```

Nodes	0	1	2	3
0				True
1				
2				
3	True			



An Interesting Property of Adjacency Matrices

Consider the following graph:



And it's associated A.M.:

$$\begin{array}{c|cccc} & \text{Joe} & \text{Mary} & \text{Tsuen} & \text{Lily} \\ \hline \text{Joe} & 0 & 1 & 0 & 0 \\ \text{Mary} & 0 & 0 & 1 & 0 \\ \text{Tsuen} & 0 & 0 & 0 & 1 \\ \text{Lily} & 0 & 1 & 0 & 0 \end{array}$$

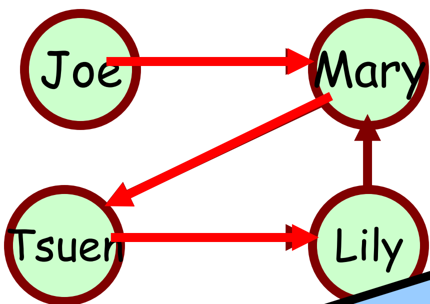
Neato effect: If you multiply the matrix by itself something cool happens!

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} = \begin{array}{c|cccc} & \text{Joe} & \text{Mary} & \text{Tsuen} & \text{Lily} \\ \hline \text{Joe} & 0 & 0 & 1 & 0 \\ \text{Mary} & 0 & 0 & 0 & 1 \\ \text{Tsuen} & 0 & 1 & 0 & 0 \\ \text{Lily} & 0 & 0 & 1 & 0 \end{array}$$

The resulting matrix shows us which vertices are exactly two edges apart.

An Interesting Property of Adjacency Matrices

Consider the following graph:



And it's associated A.M.:

	Joe	Mary	Tsuen	Lily
	0	1	0	0
	0	0	1	0
	0	0	0	1
Lily	0	1	0	0

So now you know how Google+ and Facebook work! NOT!

Neato effect: ... by itself someth...

0	1	0	0
0	0	1	0
0	0	0	1
0	1	0	0

X

0	1	0	0
0	0	1	0
0	0	0	1
0	1	0	0

=

	Joe	Mary	Tsuen	Lily
Joe	0	0	0	1
Mary	0	0	0	0
Tsuen	0	0	0	0
Lily	0	0	0	0

And if we multiply our new matrix by the original matrix again, we'll get all vertices that are exactly 3 edges apart!

Another Way to Represent a Graph

Question:

How else can we represent a graph (without a 2D array)?

Answer:

A directed graph of n vertices can be represented by an array of n linked lists. This is called an adjacency list.

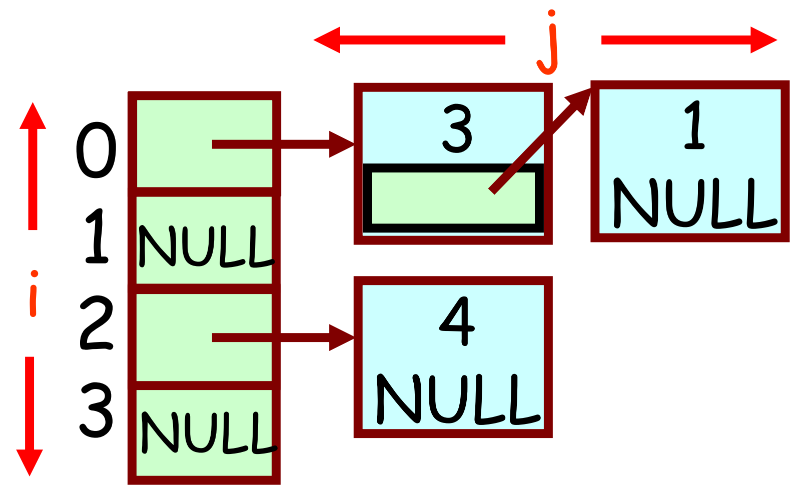
```
list<int> graph[n];
```

If we add a number j, to list number i (e.g., to graph[i]), this means that there is an edge from vertex i to vertex j.

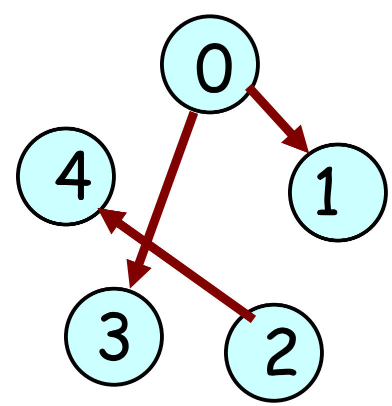
The Adjacency List

If we add a number j, to list number i (e.g., to graph[i]), this means that there is an edge from vertex i to vertex j

```
list<int> graph[4];

// edge from node 0 to node 3
graph[0].push_back(3);
graph[2].push_back(4);
graph[0].push_back(1);
```



So for each entry j, in list i, this means that there is an edge from vertex i to vertex j.



Which Representation Should You Use?

When should you use an adjacency matrix vs. an adjacency list?

Scenario #1:

We've got 10,000,000 users who have relationships with each other – typically each person is friends with just a few hundred other people.

What would you do?

Option A: Store the graph in a 10 million by 10 million array?

(That's 100 trillion cells)

Option B: Store your graph in an array holding 10 million linked lists, each holding roughly 500 items?

(That's only 5 billion pieces of data)

Which Representation Should You Use?

When should you use an adjacency matrix vs. an adjacency list?

Scenario #2:
We've got 1,000 cities, with airlines offering flights from every city to almost every other city.

What would you do?

Option A: Store the graph in a 1000 by 1000 array?

(That's 1 million cells)

Option B: Store your graph in an array holding 1000 linked lists, each holding roughly 1000 items?

(That's also 1 million pieces of data, but it's more complex)

Which Representation Should You Use?

When should you use an adjacency matrix vs. an adjacency list?

Use an adjacency matrix if you have lots of edges between vertices but few vertices (< 10,000 vertices).

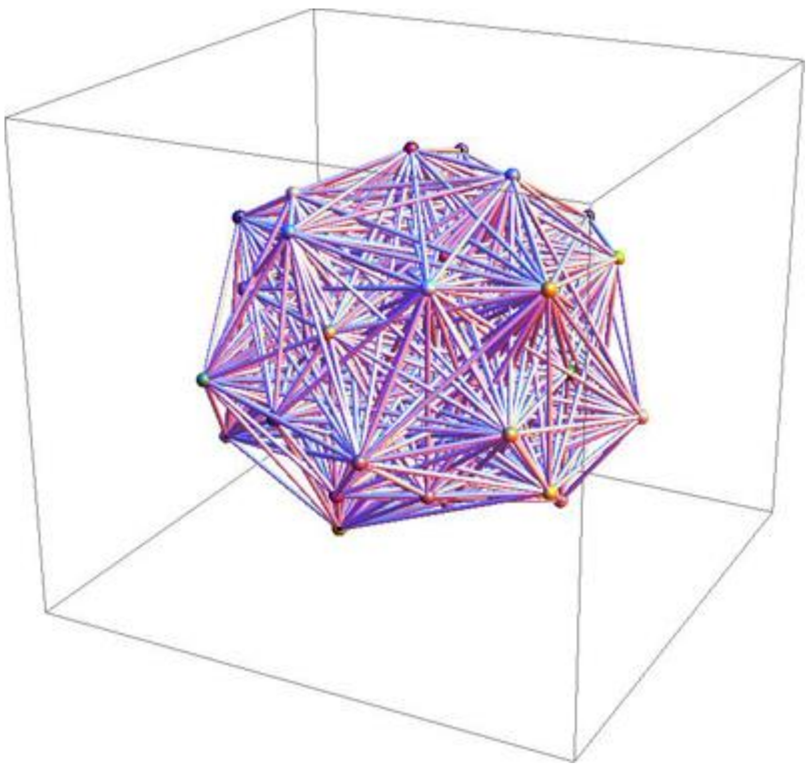
Use an adjacency list if you have few edges between vertices and lots of vertices (> 10,000 verices).

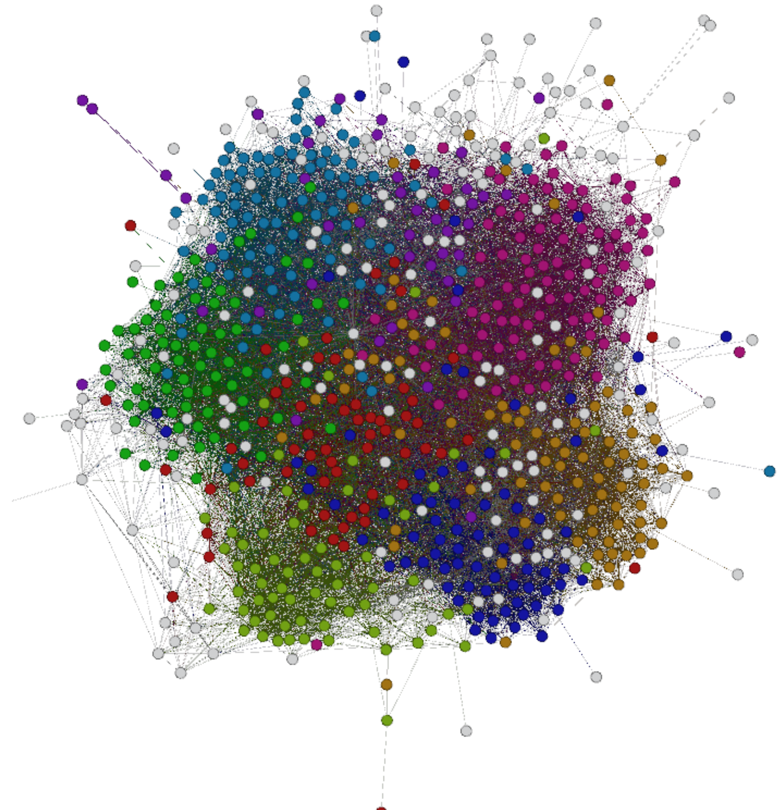
A graph that has many edges between the vertices is called a "dense graph".

A graph that has few edges between the vertices is called a "sparse graph".

Let's see examples of both...

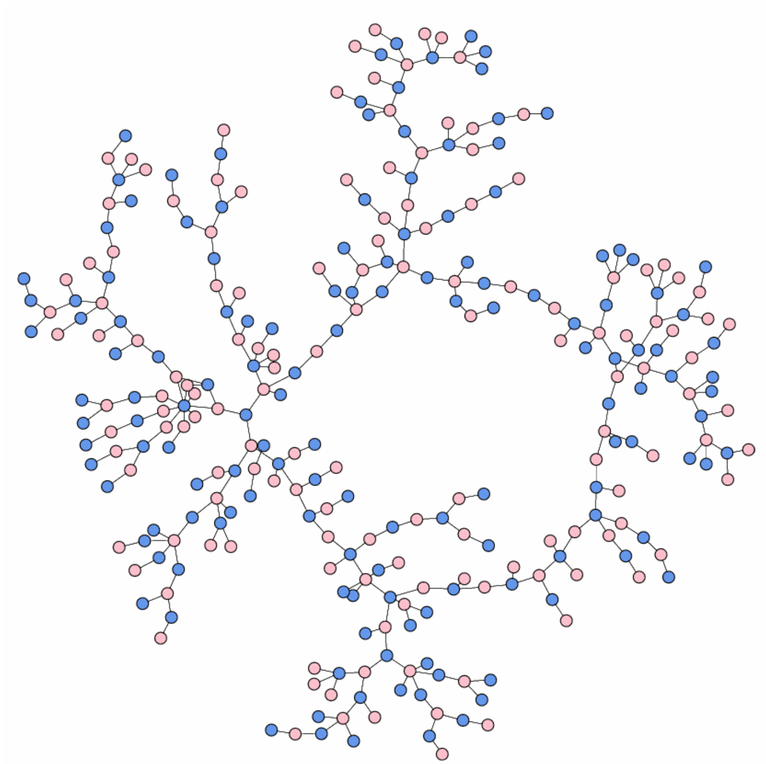
Dense(r) Graphs

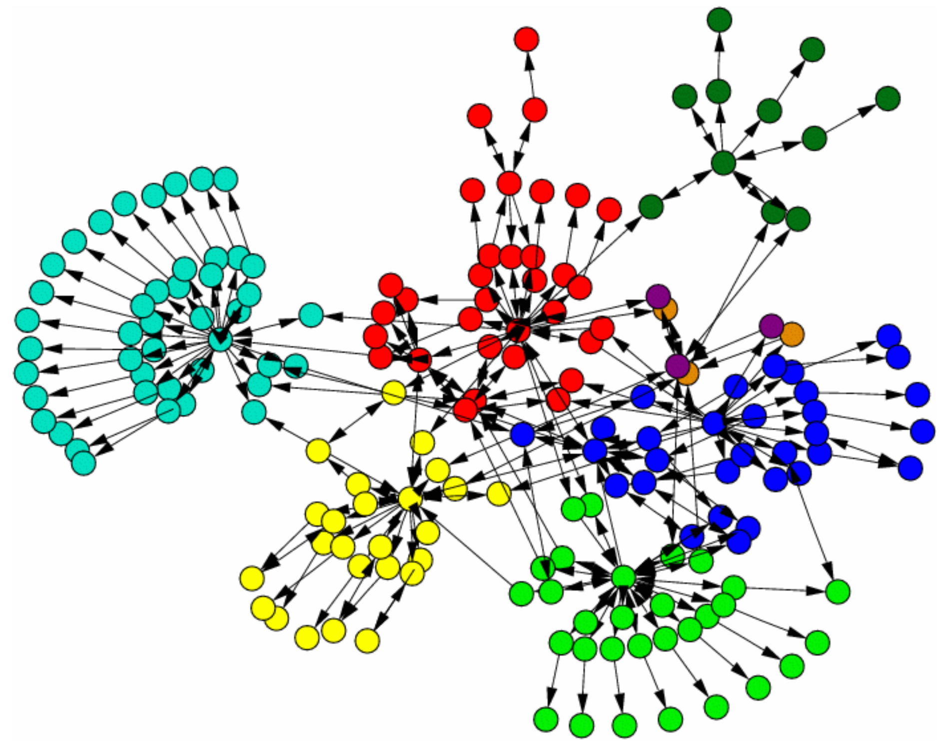




Friendships on Facebook for people from Caltech.

Sparse Graphs





(Intra-website links)

(High-school dating habits)

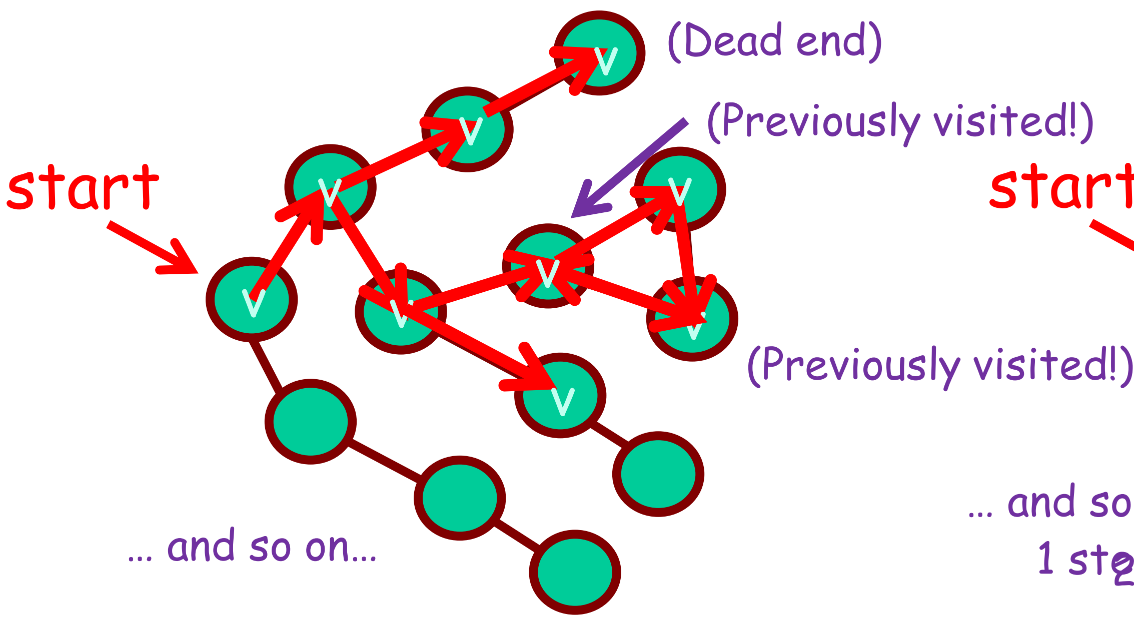
Graph Traversals

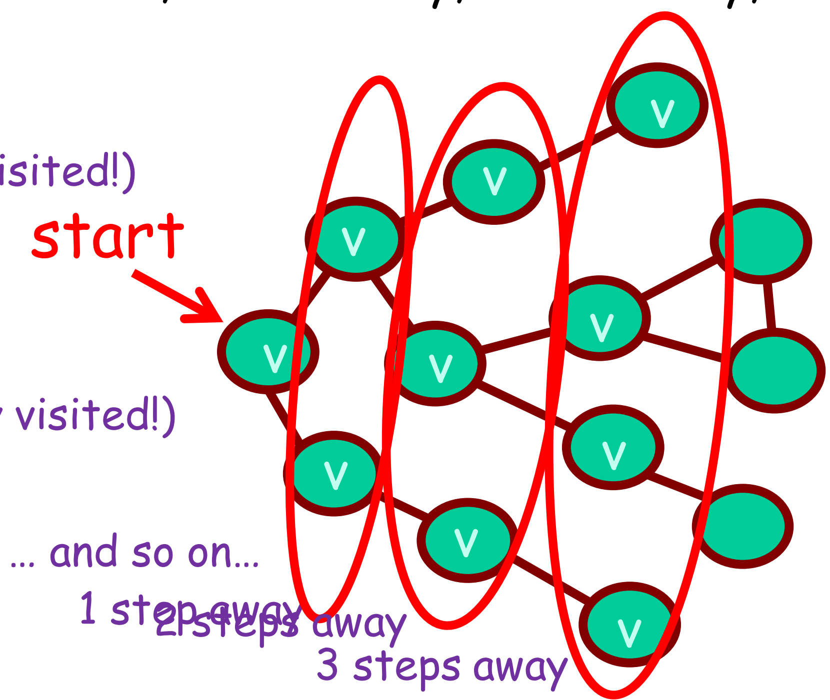
We can traverse graphs just like we traverse binary trees!

There are two types of graph traversals:
Depth-first and Breadth-first

A Depth-first Traversal keeps moving forward until it hits a dead end or a previously-visited vertex… then it backtrack sand tries another path

A Breadth-first Traversal explores the graph in growing concentric circles, exploring all vertices 1 away from the start, then 2 away, then 3 away, etc.





Depth-first Traversals

Let's learn the Depth-first Traversal algorithm first:

```
Depth-First-Traversal(curVertex)
{
   If we've already visited  the current vertex
      Return

   Otherwise
      Mark the current vertex as visited
      Process the current vertex (e.g., print it out)

      For each edge leaving the current vertex
         Determine which vertex the edge takes us to
         Call Depth-First-Traversal on that vertex
}
```

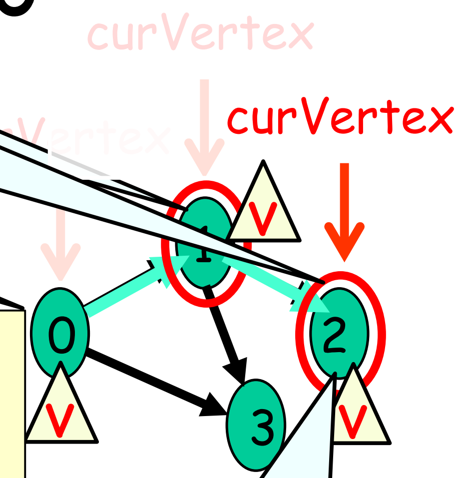
(Notice that it's recursive!)

Depth-first Traversal Demo

We haven't yet visit

We visited this

We haven't yet visited this Vertex!



Depth-First-Traversal(curVertex)
{
    If we've already visited the current vertex
        Return

    Otherwise
        Mark the current vertex as visited
        Process the current vertex (e.g., print

        For each edge leaving the current vertex
            Determine which vertex the edge takes us to
            Call Depth-First-Traversal on that vertex
}

But Vertex #2 has no outgoing edges...

So there's nothing to do!

Depth-first Traversal Demo

We haven't yet visited this Vertex!

Depth-First-Traversal(curVertex)
{
If we've already visited the current vertex
Return

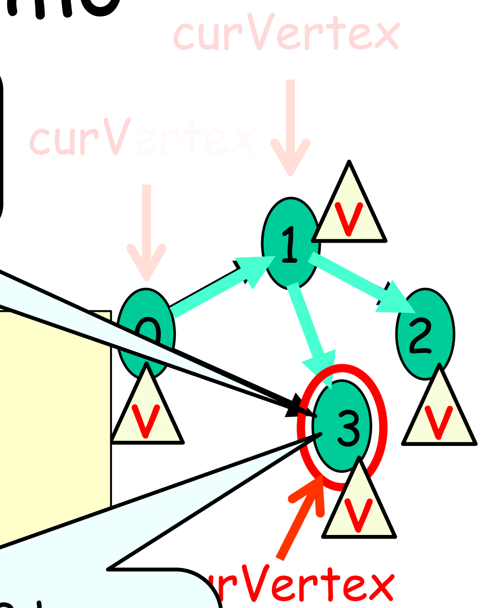
Otherwise

Mark the current vertex
Process the current verte

For each edge leaving the current vertex
Determine which vertex the edge takes us to
Call Depth-First-Traversal on that vertex
}

Alas, Vertex #3 has
no outgoing edges...

So there's nothing to do!

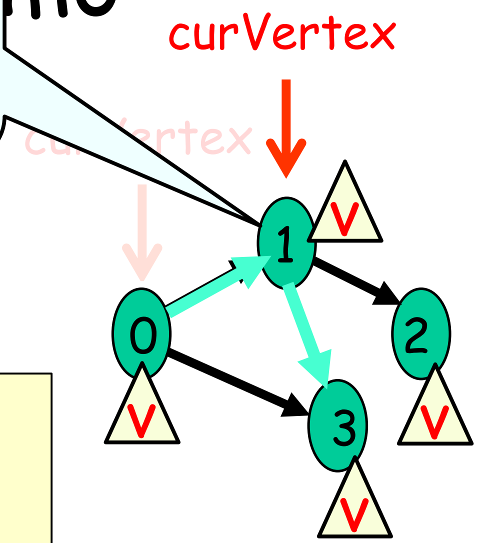


sed vertex 0!
sed vertex 1!
sed vertex 2!
rocessed vertex 3!

Depth-firs… …mo

Vertex #1 has
no MORE outgoing edges…

So there's nothing to do!



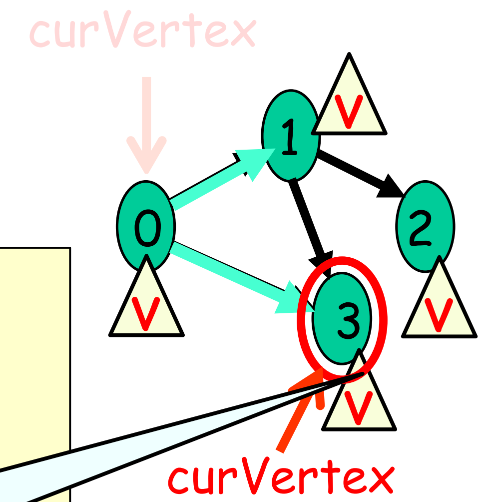
Depth-First-Traversal(curVertex)
{
  If we've already visited the current vertex
    Return

  Otherwise
    Mark the current vertex as visited
    Process the current vertex (e.g., print it out)

    For each edge leaving the current vertex
      Determine which vertex the edge takes us to
      Call Depth-First-Traversal on that vertex
}

Processed vertex 0!
Processed vertex 1!
Processed vertex 2!
Processed vertex 3!

Depth-first Traversal Demo



Depth-First-Traversal(curVertex)
{
If we've already visited the current vertex
Return

Otherwise

Mark the current vert
Process the current ve

For each edge leaving the current vertex
Determine which vertex the edge takes us to
Call Depth-First-Traversal on that vertex
}

D
{
}

But we've already visited Vertex #3!

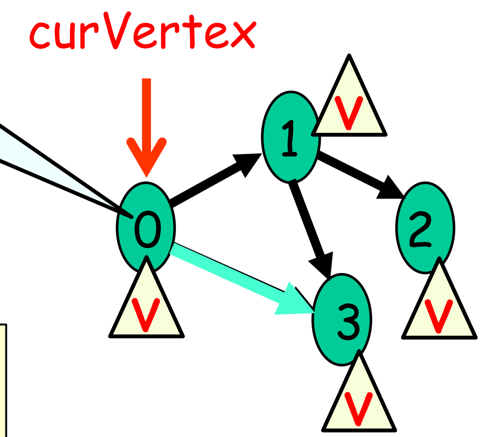
So we don't want to do so again!

Processed vertex 0!
Processed vertex 1!
Processed vertex 2!
Processed vertex 3!

Depth-first Traversal Demo

Vertex #0 has
no MORE outgoing edges...

So there's nothing to do!



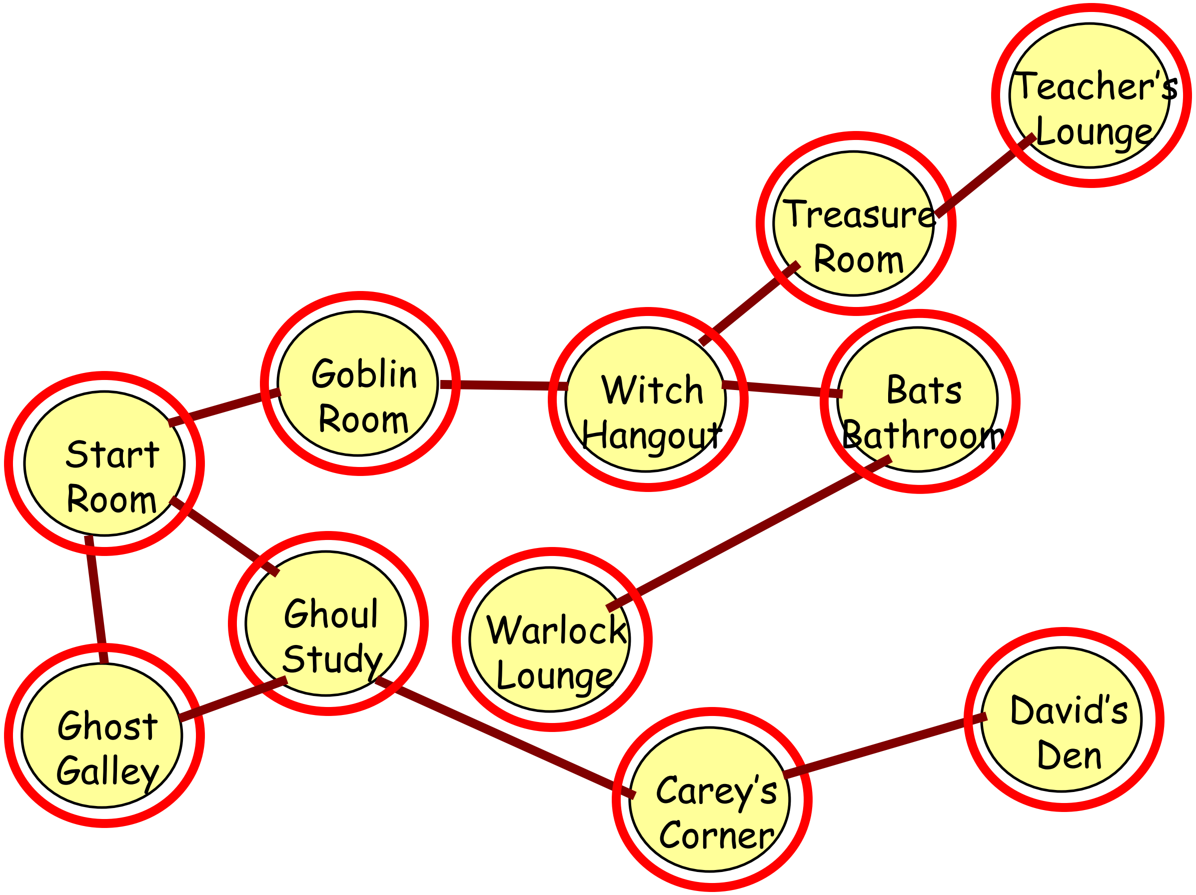

```
Depth-First-Traversal(curVertex)
{
   If we've already visited  the current vertex
       Return

   Otherwise
     Mark the current vertex as visited
     Process the current vertex (e.g., print it out)

     For each edge leaving the current vertex
        Determine which vertex the edge takes us to
        Call Depth-First-Traversal on that vertex
}
```

Processed vertex 0!
Processed vertex 1!
Processed vertex 2!
Processed vertex 3!

And we're done!

Depth-first Traversal Challenge

What does a Depth-first Traversal look like on this graph?



Implementing Depth-first Traversal w/Stack!

You can also implement your Depth-first Traversal with a stack if you like! (What's not to like???)

```
Depth-First-Search-With-Stack(start_room)
  Push start_room on the stack
  While the stack is not empty
        Pop the top item off the stack and put it in variable c
        If c hasn't been visited yet
           Drop a breadcrumb (we've visited the current room)
           For each door d leaving the room
           If the room r behind door d hasn't been visited
              Push r onto the stack.
```

Basically, the stack allows you to simulate recursion…

Or does the recursion allow you to simulate a stack?

Hmmmmmmm!

Breadth-first Graph Traversal

Idea:

Process all of the vertices that are 1 edge away from the start vertex,

then process all vertices that are two edges away,

then process all vertices that are three edges away,

etc...

Question:

What data structure could we use to implement this?

Answer:

Not a P, but a ?

Breadth-first Graph Traversal

```
Breadth-First-Search (startVertex)
{
   Add the starting vertex to our queue
   Mark the starting vertex as "discovered"
   While the queue is not empty
      Dequeue the top vertex from the queue and place in c
      Process vertex c (e.g., print its contents out)
      For each vertex v directly reachable from c
         If v has not yet been "discovered"
            Mark v as "discovered"
            Insert vertex v into the queue
}
```

Hmmm. Does this algorithm look familiar?

It's a-maze-ingly similar to our queue-based maze-solving algorithm!!!

Breadth-first Traversal Demo

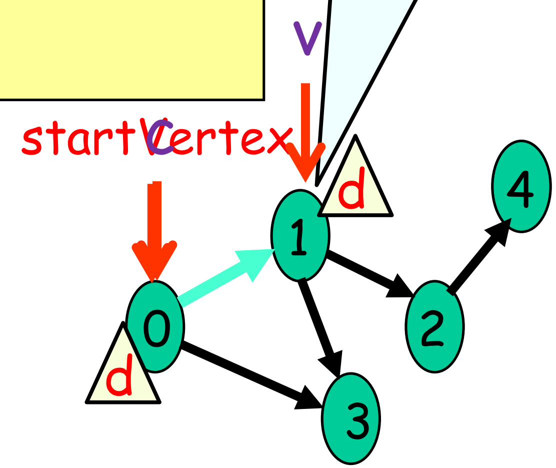
Processed vertex 0!

```
Breadth-First-Search (startVertex)
{
   Add the starting vertex to our queue
   Mark the starting vertex as "discovered"
   While the queue is not empty
      Dequeue the top vertex from the queue and place in c
      Process vertex c (e.g., print its contents out)
      For each vertex v directly reachable from c
            If v has not yet been "discovered"
               Mark v as "discovered"
               Insert vertex v into the queue
}
```

We haven't discovered this this Vertex yet!

c

Queue



Breadth-first Traversal Demo

Processed vertex 0!

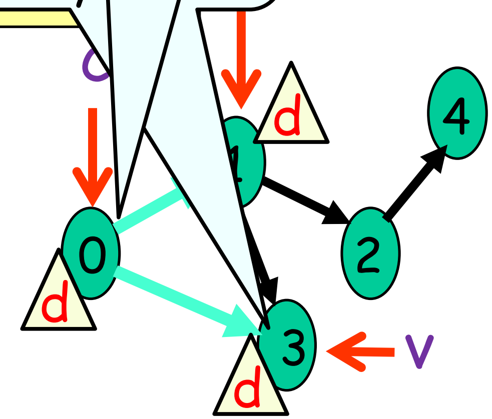
```
Breadth-First-Search (startVertex)
{
   Add the starting vertex to our queue
   Mark the starting vertex as "discovered"
   While the queue is not empty
      Dequeue the top vertex from the queue and place in c
      Process vertex c (e.g., print its contents out)
      For each vertex v directly reachable from c
            If v has not yet been "discovered"
               Mark v as "discovered"
               Insert vertex v into the queue
}
```

Vertex c has no other edges, so we're done with it.

c
0

Queue

1	3				



Breadth-first Traversal Demo

Processed vertex 0!
Processed vertex 1!

Breadth-First-Search (startVertex)
{
Add the starting vertex to our queue
Mark the starting vertex as "discovered"
While the queue is not empty
Dequeue the top vertex from the queue and place in c
Process vertex c (e.g., print its con
For each vertex v directly reachab
If v has not yet been "discove
Mark v as "discovered"
Insert vertex v into the queue
}

Vertex #1 has no more outgoing edges…

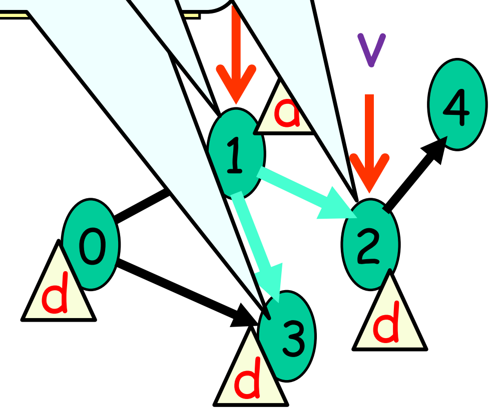
n't
his this

Ah ha! We ve already disco ed this this Verte

c

Queue

1	2				



Breadth-first Traversal Demo

Processed vertex 0!
Processed vertex 1!
Processed vertex 3!

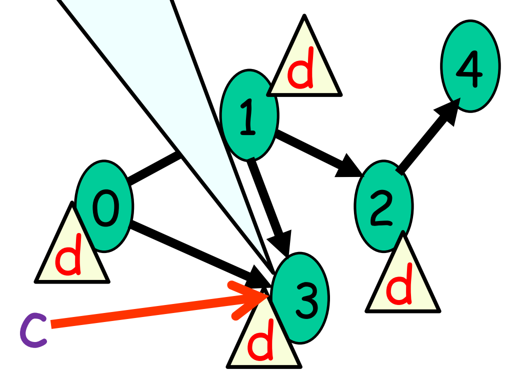
Breadth-First-Search (startVertex)
{
Add the starting vertex to our queue
Mark the starting vertex as "discovered"
While the queue is not empty
Dequeue the top vertex from the queue and place in c
Process vertex c (e.g., print its contents out)
For each vertex v directly reachable from c
If v has not yet been "discovered"
Mark v as "discovered"
Insert vertex v into the queu
}

Vertex #3 has NO outgoing edges at all! So we're done.

c

Queue

3	2				



Breadth… …rsal Demo

And finally we're done!

Processed vertex 0!
Processed vertex 1!
Processed vertex 3!
Processed vertex 2!
Processed vertex 4!

Breadth-First-Sea… (startVertex)
{
  Add the starti… vertex to our queue
  Mark the star…ing vertex as "discovered"
  While the queue is not empty
    Dequeue the top vertex from the queue and…
    Process vertex c (e.g., print its contents out…
    For each vertex v directly reachable fro…
      If v has not yet been "discovered"
        Mark v as "discovered"
        Insert vertex v into the queue
}

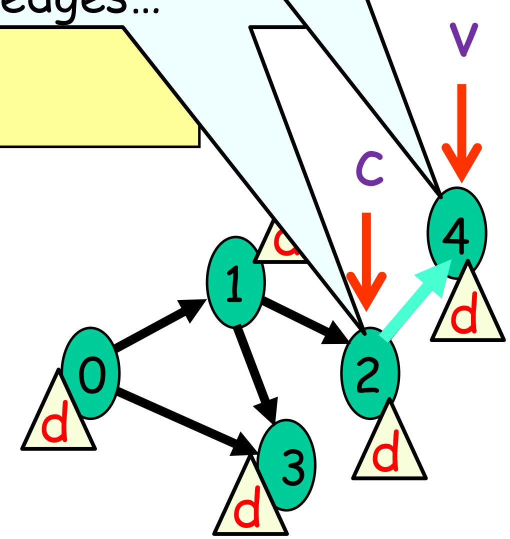
Vertex #4 has NO outgoing edges…

more outgoing edges…

c

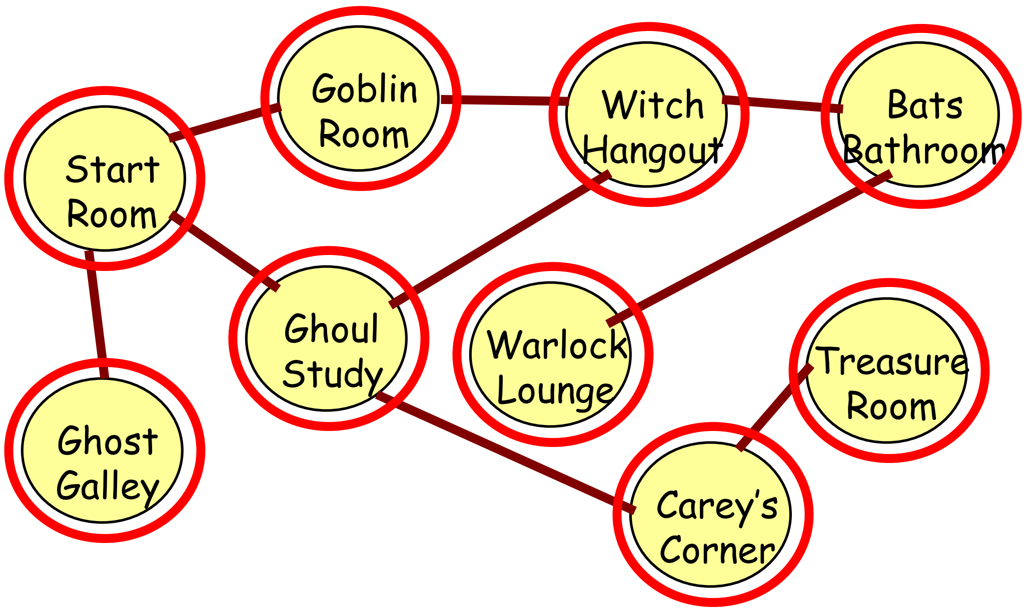
Queue

4					



Breadth-first Traversal Challenge

What does a Breadth-first Traversal look like on this graph?

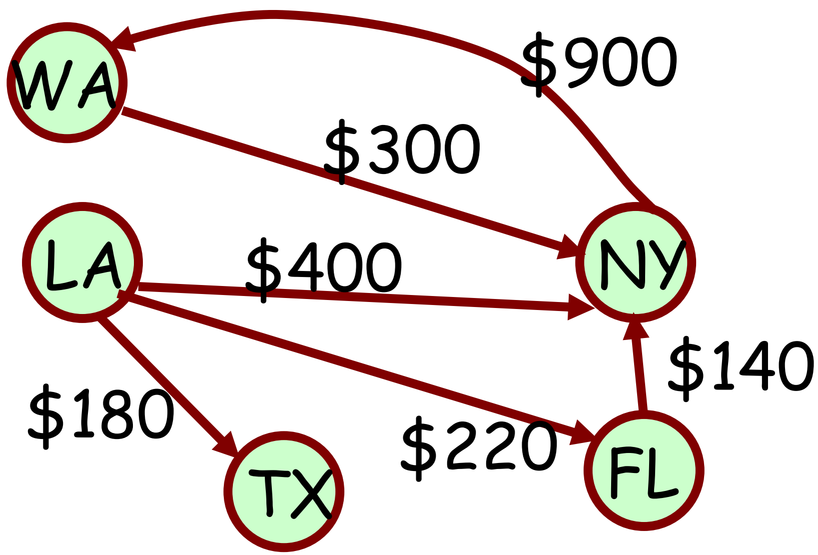


Graphs With Weighted Edges

What does it mean for a graph to have weighted edges?

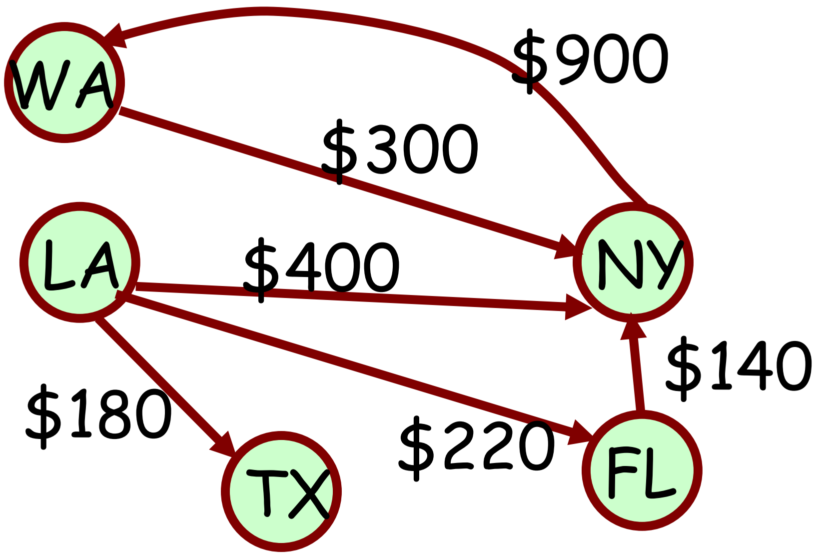
Definition: Each edge connecting vertex u with vertex v has a weight or cost associated with it.

Question: Why would we want to have weighted edges?



Graphs With Weighted Edges

Definition: The weight of a path from vertex u to vertex v is the sum of the weights of the edges between the two vertices.



Question: What's the cost of traveling from LA to NY to WA?

Question: What's the shortest path from LA to WA?

Definition: The shortest path between two vertices is the path with the lowest total cost of edges between the two vertices. (The shortest path is a set of vertices)

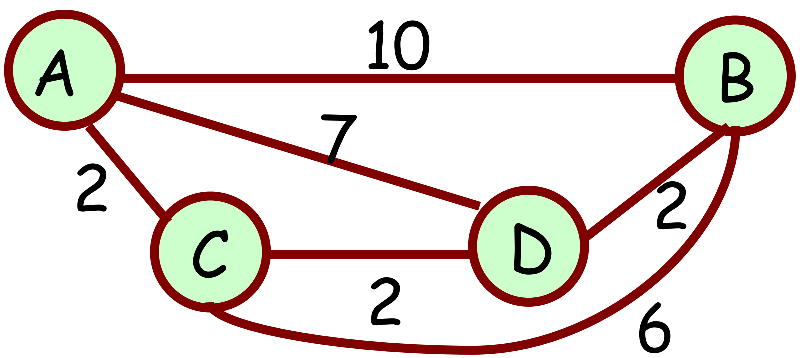
Finding the Shortest Path

Question: How can we find the shortest path between any two nodes in a graph?

Answer: Dijkstra's Algorithm (the dorm guy?)

Dijkstra's Algorithm:

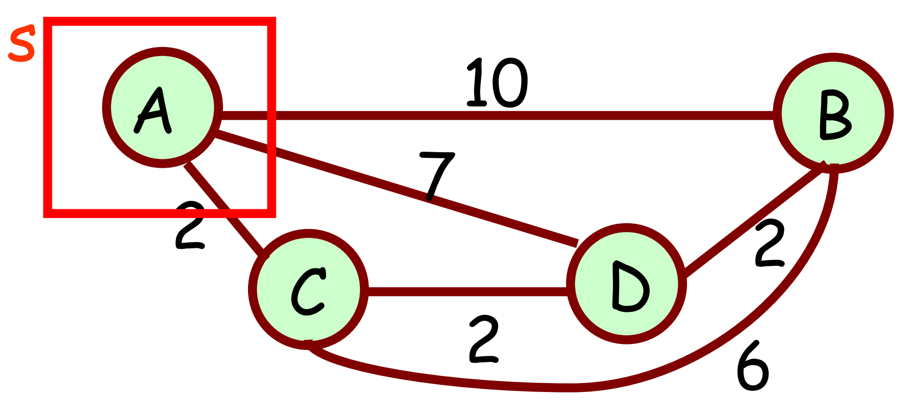
This algorithm determines the length of the shortest path (i.e. set of vertices) from a start vertex s to all other vertices in the graph.



So Dijkstra(A) would give us a value of 6 for A to B, a value of 2 for A to C, and 4 for A to D.

Dijkstra's Algorithm

Input: A graph G, and a starting vertex s



G must not have any negative edge values.

Output: An array called Dist of *optimal distances* from s to every other node in the graph.

Dist

A	B	C	D
0	6	2	4

Dijkstra's Algorithm: Basic Idea

Dijkstra's algorithm splits vertices in two distinct sets: the set of *unsettled* vertices and the set of *settled* vertices.

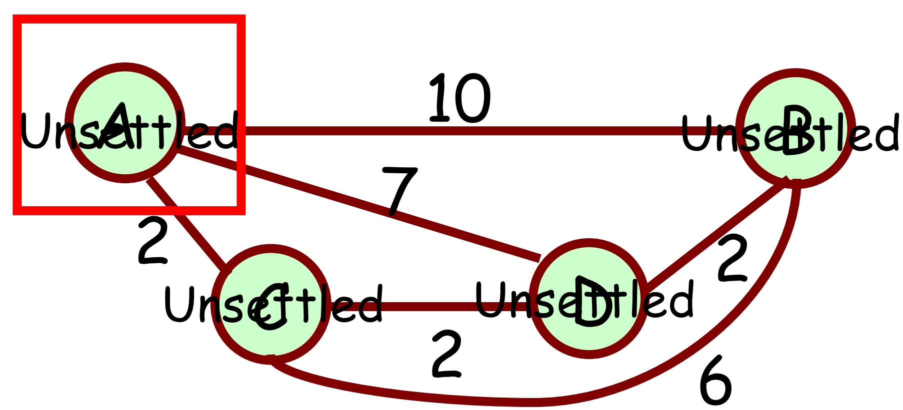
Unsettled vertex: A vertex v is unsettled if we don't know the optimal distance to it from the starting vertex s.

Settled vertex: A vertex v is settled if we have learned the optimal distance to it from the starting vertex s.

Initially all vertices are unsettled.

The algorithm ends once all vertices are in the settled set.

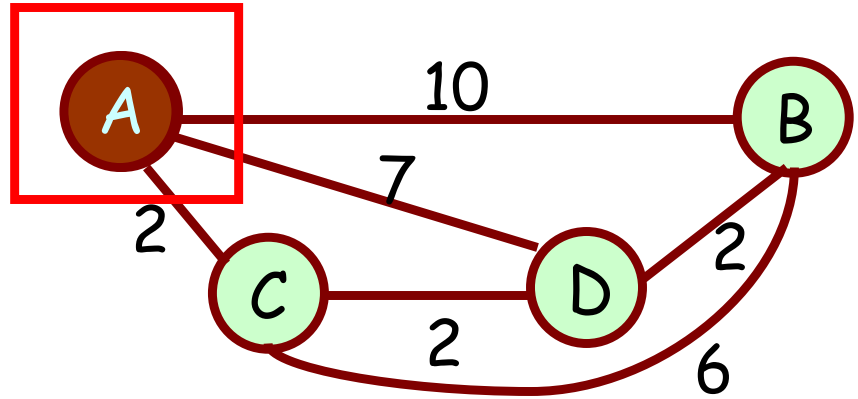
Start vertex



Dijkstra on a Graph

Assume that all vertices are infinitely far away to start…

Since we start at vertex A, we know it's the closest vertex to us! How far is it? Zero steps away! We can settle it immediately!



Distance
(Best known so far)

0	10	2	7
A	B	C	D

Now let's see which unsettled vertices we can reach *directly* from A.

- B is 10 units away.
- C is 2 units away.
- D is 7 units away.

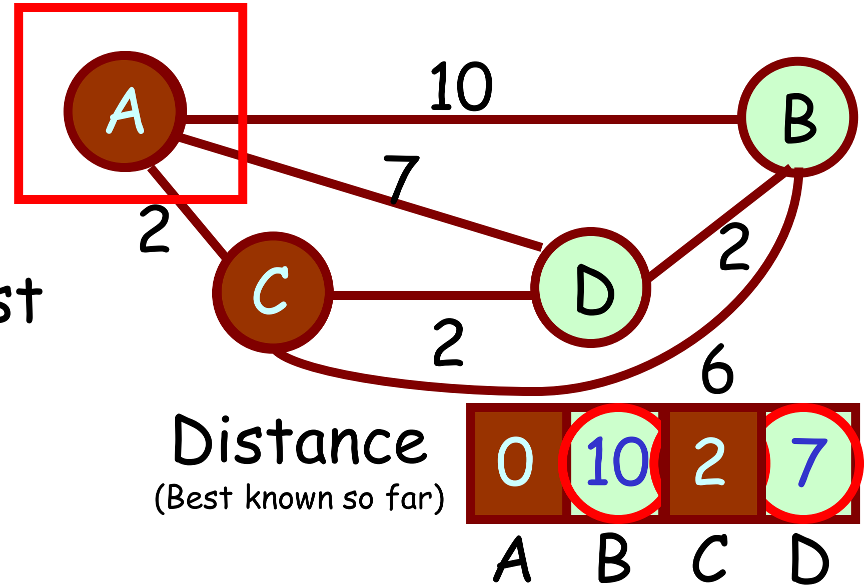
And going directly from A to D is only 7 units away, which is less than infinity, so I'll update this entry too…

Dijkstra on a Graph

Ok, so now we know the costs to travel to all vertices directly reachable from A.

Which unsettled vertex is closest to A?

Right! C is closest to A.

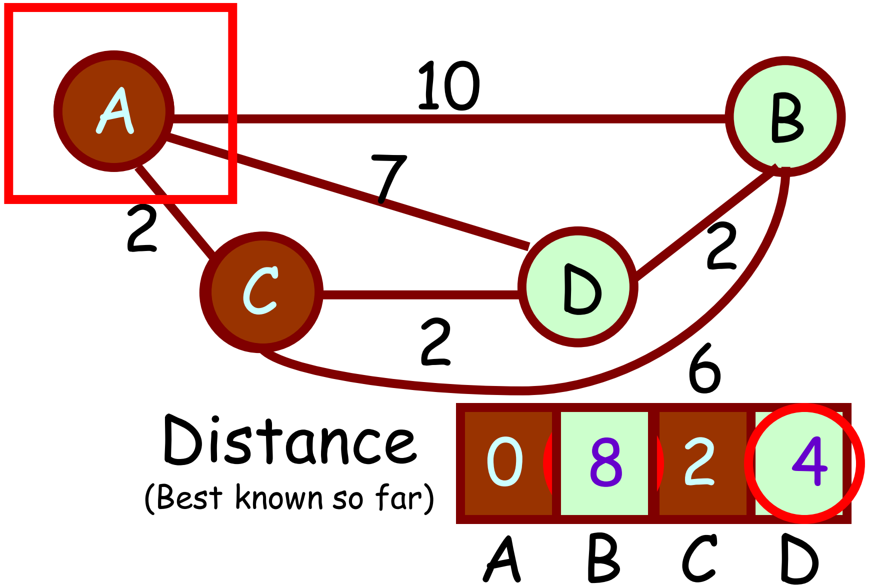


If we go directly to C (A → C), it costs us 2 units. Is there any possible way I can travel to C cheaper by going through B or through D first?

So I know that if I travel directly from A to C, at a cost of 2 units, that's the *fastest* possible route. Therefore we can settle C at 2 units.

Dijkstra on a Graph

At this point, we know the shortest path from A to C. Now let's see if we can travel through C to reach one of our other unsettled vertices faster.



Ok, which unsettled vertices can be reached directly from C?

- B is 6 units away.
- D is 2 units away.

Let's do D next. We know we can get from A to C in 2 units, and we can directly go from C to D in 2 units, so we can reach D in just 4 units!

Is our new distance to D better than our old one?

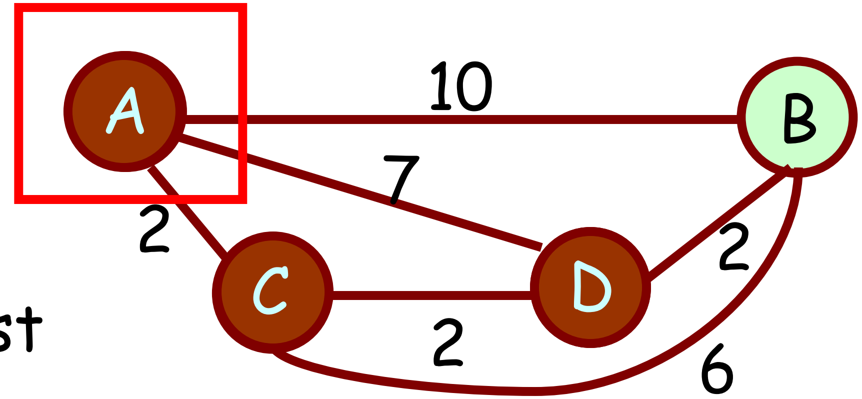
Yup!! Let's update our table again!

Dijkstra on a Graph

Ok, so now we know the best cost to get to all unsettled vertices, assuming we travel through C.

Which unsettled vertex is closest to A now?

Right! D is closest.



Distance
(Best known so far)

A	B	C	D
0	8	2	4

If we take the path A → C → D, it costs us 4 units. Is there any possible way I can travel to D cheaper by going through B first?

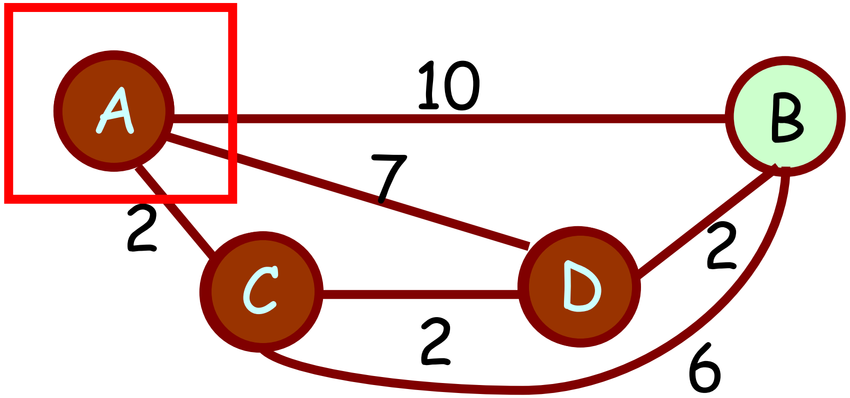
So I know that if I travel from A → C → D, at a cost of 4 units, that's the *fastest* possible route. Therefore we can settle D at 4 units.

Dijkstra on a Graph

At this point, we know the shortest path from A to D. Now let's see if we can travel through D to reach one of our other unsettled vertices faster.

Ok, which unsettled vertices can be reached directly from D?

- B is 2 units away.



Distance
(Best known so far)

A	B	C	D
0	6	2	4

Let's check B. We know we can get from A to D in 4 units, and we can directly go from D to B in 2 units, so we can reach B in just 6 units!

Is our new distance to B better than our old one?

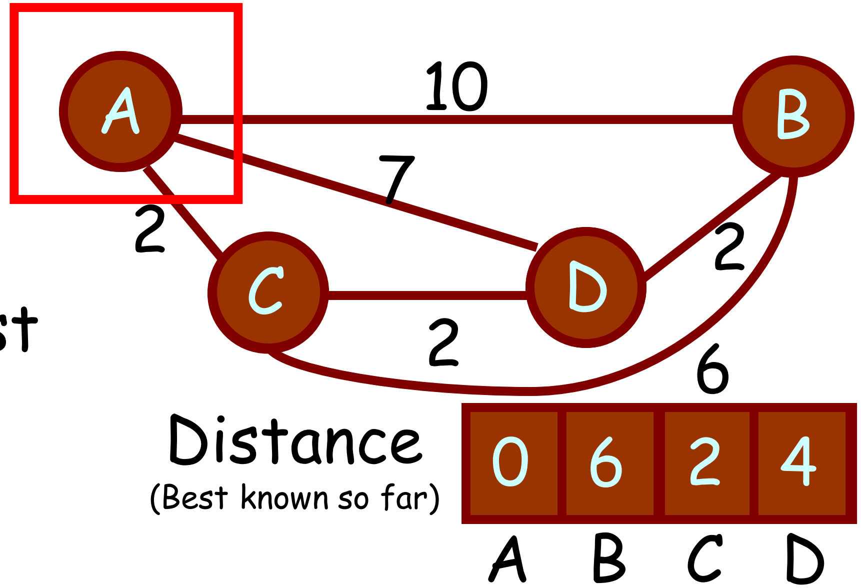
You bet!! Let's update our table!

Dijkstra on a Graph

Ok, so now we know the best cost to get to all unsettled vertices, assuming we travel through D.

Which unsettled vertex is closest to A now?

Right! B is closest.



And now that all of our vertices are settled, we are guaranteed to have found the *minimum* travel distances to each of our vertices!

Dijkstra

And now I'll give you the more formal algorithm…

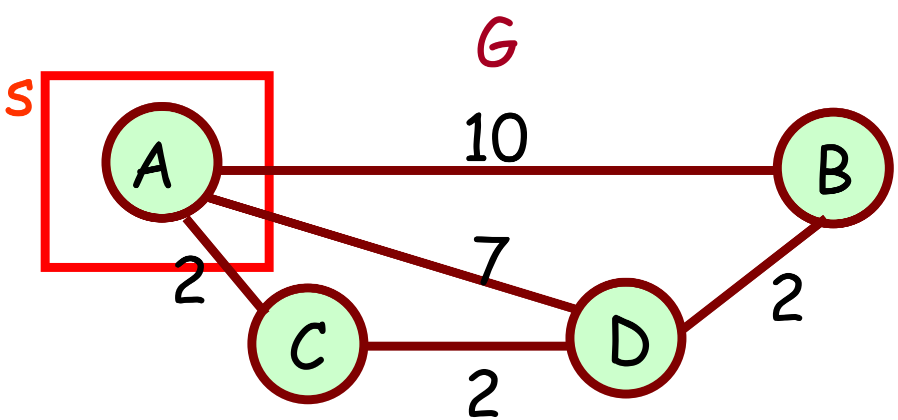
Born: 11 May 1930, Rotterdam, Netherlands
Died: 6 August 2002, Nuenen, Netherlands

Dijkstra's Algorithm

Dijkstra's Algorithm uses 2 data structures:

1. An array called Dist that holds the the current best known cost to get from s to every other vertex in the graph.

   For each vertex i, Dist[i] starts out with a value of:
   - 0 for vertex s
   - Infinity for all other vertices



Dist from vertex s to…

A	B	C	D
0	∞	∞	∞

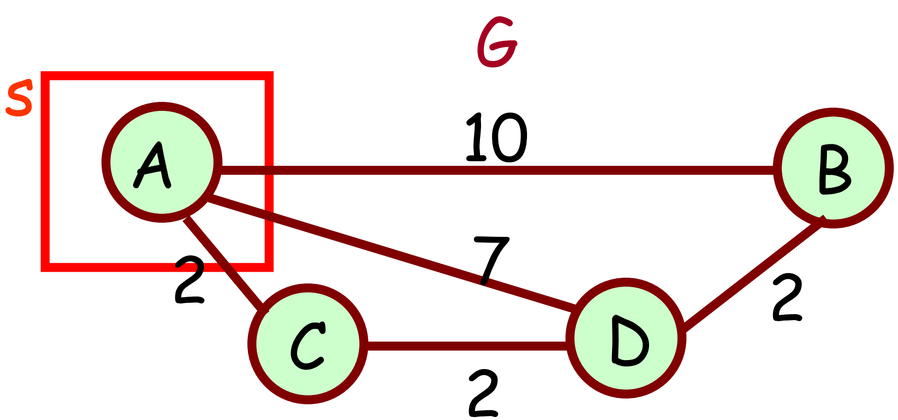
Idea: We start at node A so we're 0 steps away from node A. We assume the other vertices are infinitely far away from A.

Dijkstra's Algorithm

Dijkstra's Algorithm uses 2 data structures:

2. An array called Done that holds true for each vertex that has been fully processed, and false otherwise.

For each vertex i, Done[i] starts out with a value of false.



Dist from vertex s to…

A	B	C	D
0	∞	∞	∞

Done

A	B	C	D
false	false	false	false

Dijkstra's Algorithm

While there are still unprocessed vertices:

Set u = the closest unprocessed vertex to the start vertex s

Mark vertex u as processed: Done[u] = true.

We now know how to reach u optimally from s

Loop through all unprocessed vertices:

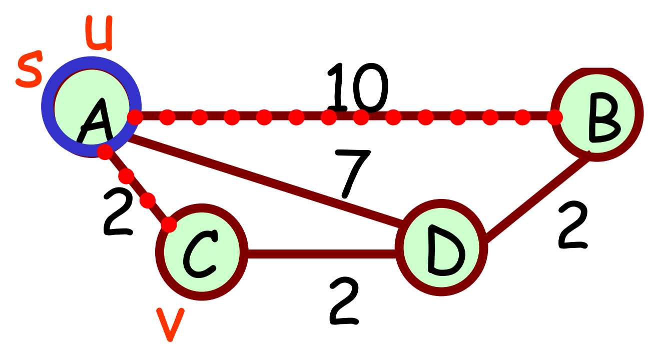
Set v = the next unprocessed vertex

If there's an edge from u to v then compare:

a. the previously computed path from s to v (i.e. Dist[v]) OR

b. the path from s to u, and then from u to v (I.e. Dist[u] + weight(u,v))

If the new cost is less than old cost then Set Dist[v] = Dist[u] + weight(u,v)



Dist from vertex s to…

A	B	C	D
0	10	2	∞

Done

A	B	C	D
true	false	false	false
u		v	

Dijkstra's Algorithm

While there are still unprocessed vertices:

Set u = the closest unprocessed vertex to the start vertex s

Mark vertex u as processed: Done[u] = true.

We now know how to reach u optimally from s

Loop through all unprocessed vertices:

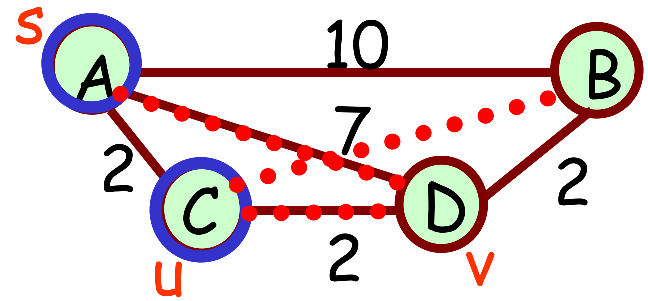
Set v = the next unprocessed vertex

If there's an edge from u to v then compare:

a. the previously computed path from s to v (i.e. Dist[v]) OR

b. the path from s to u, and then from u to v (I.e. Dist[u] + weight(u,v))

If the new cost is less than old cost then Set Dist[v] = Dist[u] + weight(u,v)



Previous cost: 7

New cost: 2 + 2 = 4

	A	B	C	D
Dist from vertex s to…	0	10	2	4
Done	true	false	true	false
			u	v

Dijkstra's Algorithm

While there are still unprocessed vertices:

Set u = the closest unprocessed vertex to the start vertex s

Mark vertex u as processed: Done[u] = true.

We now know how to reach u optimally from s

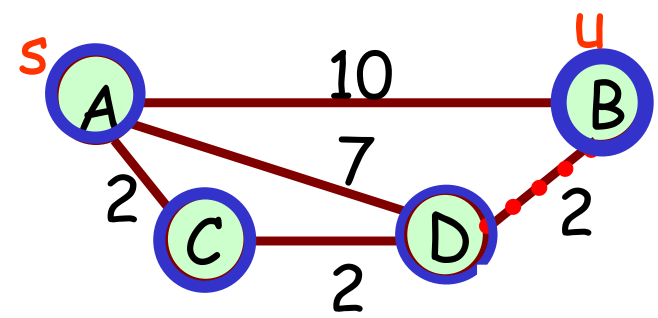
Loop through all unprocessed vertices:

Set v = the next unprocessed vertex

If there's an edge from u to v then compare:

a. the previously computed path from s to v (i.e. Dist[v]) OR
b. the path from s to u, and then from u to v (I.e. Dist[u] + weight(u,v))

If the new cost is less than old cost then
Set Dist[v] = Dist[u] + weight(u,v)



Previous cost: 10

New cost: 4 + 2 = 6

Dist from vertex s to...	A 0	B 6	C 2	D 4
Done	A true	B true	C true	D true
		u		

And we're done! The Dist array contains the results.