University of Illinois at Urbana-Champaign
Dept. of Electrical and Computer Engineering

ECE 120: Introduction to Computing

Good Design

Two Goals Guide Our Choices in Software Design

1. **simpler (or feasible) approach**
   - avoid unnecessary complexity
   - use clear and obvious techniques when possible
   - a simple design that does work is better than a complex design that may work
2. **easy to understand and test**
   - as easy as possible to read (structure, indentation, comments!)
   - organize functionality to enable both separate and system-wide testing

Example: Survey Forms

What is the metric of goodness, anyway?

Imagine that you want
- to survey opinions of a group of **N** in a room
- using 20 questions with answers 1 to 5.

You then want to
- average the answers to each question
- over the **N** participants.

Method 1: One Form per Person

You could take the usual approach:
- make a survey form with your 20 questions
- and have each person fill in one form.

How do you compute the averages?

In particular, do you iterate
first over questions or forms?

Which Algorithm is Better?

```
for each question Q
   sum = 0
   for each form F
      sum = sum + F(Q)
   print sum divided by N
```

(question first)

```
arraySum = all zeroes
for each form F
   for each question Q
      arraySum[Q] = arraySum[Q] + F(Q)
for each question Q
   print arraySum[Q] divided by N
```

(form first)

Pros and Cons of the Question-First Algorithm

```
for each question Q
   sum = 0
   for each form F
      sum = sum + F(Q)
   print sum divided by N
```

Only have to remember one sum at a time.

Get out form F and look up question Q a total of 20N times.

Pros and Cons of the Form-First Algorithm

```
arraySum = all zeroes
for each form F
   for each question Q
      arraySum[Q] = arraySum[Q] + F(Q)
for each question Q
   print arraySum[Q] divided by N
```

Need to keep track of 20 different sums!

One form at a time, reading answers (once each) in order.

Method 2: One Form per Question

But why use the traditional approach?

We can get the best of both algorithms by changing our data gathering method!

In particular:

- Make a form for each of the 20 questions.
- Have each person answer once on each form.

New Algorithm is Better than Either of the Others!

for each form/question **Q**
  sum = 0
  for each person **P** (all on one form)
    sum = sum + P(Q)
  print sum divided by **N**

Only have to remember one sum at a time.

One form at a time, reading answers (once each) in order.

Which Method is Better?

Method 1: One Form per Person

Method 2: One Form per Question

So what do you think?

What's better about **Method 1**?
- easier to organize
- easier to avoid "cheating"
- easier to measure participation (per person)

Many Ways to Measure Goodness

All that from a simple choice:

do we iterate first over forms,
or over questions?

(We have to iterate over both. We just get to pick which one is first, and which is second.)

As you can see, there are **many ways to measure "goodness"** for algorithm design.

It's That Time Again

Time to help me, I mean.

I need coffee.

But first, I need food.

I have a map.

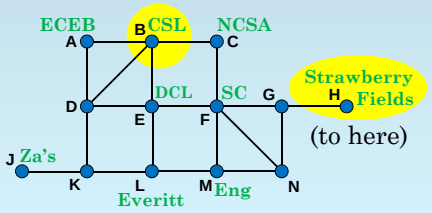
Help me to find my way
- from my office in CSL
- to Strawberry Fields.

Directions Needed from CSL to Strawberry Fields

I want to go from my office in CSL…

… to Strawberry Fields.

(from here)

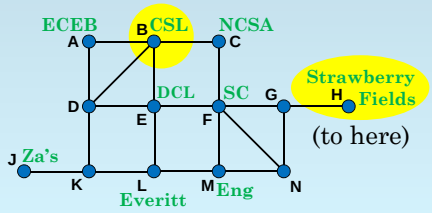


Directions Needed from CSL to Strawberry Fields

Can you give me a path?

B → C or E → F → G → H? Great!

(from here)



Actually, I May Need More Help

So.

Well.

Can you tag along with me?

Around campus, I mean?

We can give you a title.

"Assistant Walking Director"

Or something like that.

Actually… Just Teach My Computer!

Oh, wait!

You just learned how to program!

Teach my computer how to help me.

I will enter the map.

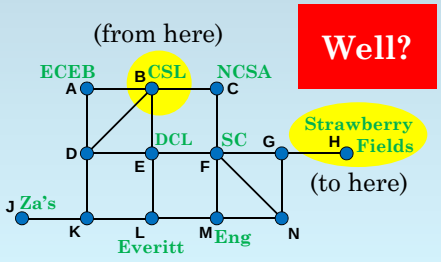
I will say where I am.

I will say where I want to go.

You tell it how to find a path.

Please Teach My Computer to Do What You Did

Here's the map. I want to be able to **find a path from any node to any other**.



Here's One Approach to Finding the Shortest Path

Don't panic!

I have **An Idea™**!

Here it is:

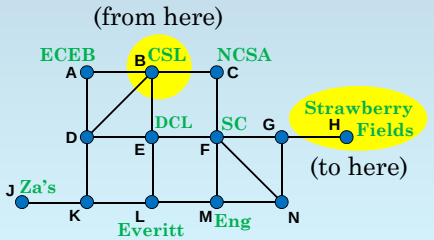
1. list all paths,
2. measure all of the paths, and
3. pick the shortest one.

Sound good?

Let's Try My Idea: First, List the Paths

Starting from **B**…

B → A → B → A → B → A → B → A → B → A



Professors Shouldn't Be Allowed to Wander in Circles

Don't panic!

I have ~~**An Idea™**~~ **a better idea**!

Here it is:

1. list ~~all paths~~, **all simple paths,**
2. measure all of the paths, and
3. pick the shortest one.

In mathematics, a "simple path" is one that includes any node at most one time. (So we can't go back to a place we've been already.)

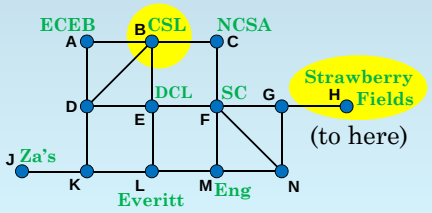
Let's Try Again: First, List the Simple Paths

Starting from **B**…

That's one path!

B → C → F → E → L → M→ N → G → H

(from here)



This Problem May Be a Little Exhausting

As it turns out …

For a general graph with **P** nodes,
- the number of simple paths
- between any pair of nodes
- is exponential in **P**.

The solution?

For **HW12**, please list **all paths for all pairs** of nodes in my map.

(Just kidding.)

One More Try! Let's Use a Queue

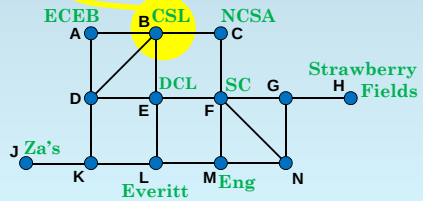
Let's make
- **a queue** of nodes
- and keep track of the **best previous location** for each node.

We'll **process the nodes** in the queue
- one by one
- **by adding any unvisited neighbors** to the queue.

Use a Queue to Find Shortest Paths

	explored								
queue	B								
previous	-								

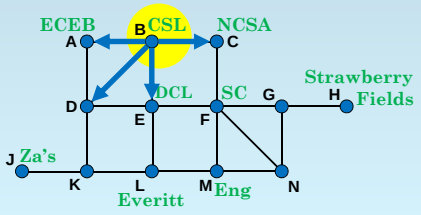
Add the starting node to the queue.



Process Node B by Adding all Neighbors

explored	√									
queue	B	A	D	E	C					
previous	-	B	B	B	B					

Process the next queue node by adding neighbors.



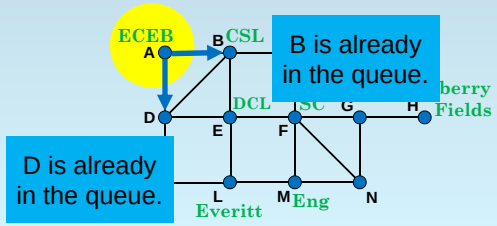
Process Node A by Adding all Neighbors

explored	√	√								
queue	B	A	D	E	C					
previous	-	B	B	B	B					

Process the next node.

B is already in the queue.

D is already in the queue.



Process Node D by Adding all Neighbors

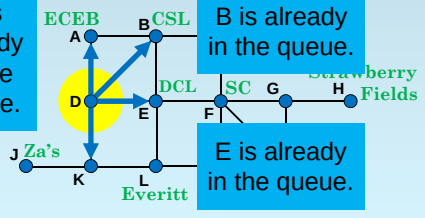
explored	√	√	√							
queue	B	A	D	E	C	K				
previous	-	B	B	B	B	D				

Process the next node.

A is already in the queue.

B is already in the queue.

E is already in the queue.



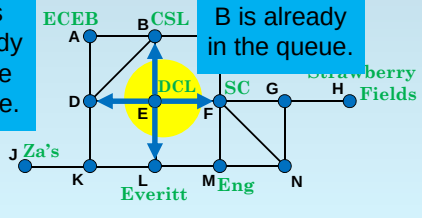
Process Node E by Adding all Neighbors

explored	√	√	√	√						
queue	B	A	D	E	C	K	L	F		
previous	-	B	B	B	B	D	E	E		

Process the next node.

D is already in the queue.

B is already in the queue.

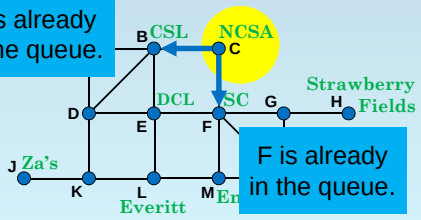


Process Node C by Adding all Neighbors

explored	√	√	√	√	√				
queue	B	A	D	E	C	K	L	F	
previous	-	B	B	B	B	D	E	E	

Process the next node.

B is already in the queue.



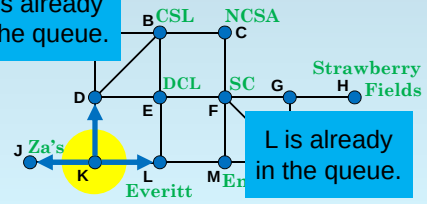
F is already in the queue.

Process Node K by Adding all Neighbors

explored	√	√	√	√	√	√				
queue	B	A	D	E	C	K	L	F	J	
previous	-	B	B	B	B	D	E	E	K	

Process the next node.

D is already in the queue.



L is already in the queue.

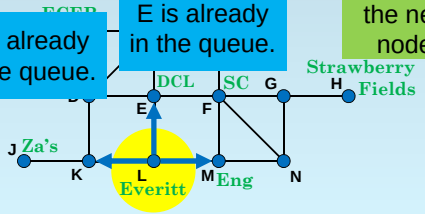
Process Node L by Adding all Neighbors

explored	√	√	√	√	√	√	√						
queue	B	A	D	E	C	K	L	F	J	M			
previous	-	B	B	B	B	D	E	E	K	L			

Process the next node.

E is already in the queue.

K is already in the queue.

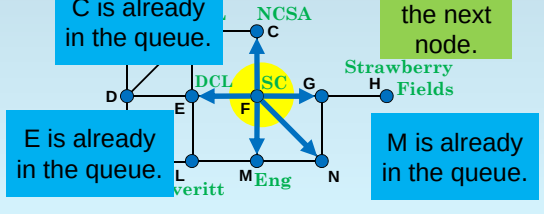


Process Node F by Adding all Neighbors

explored	√	√	√	√	√	√	√	√					
queue	B	A	D	E	C	K	L	F	J	M	N	G	
previous	-	B	B	B	B	D	E	E	K	L	F	F	

Process the next node.

C is already in the queue.

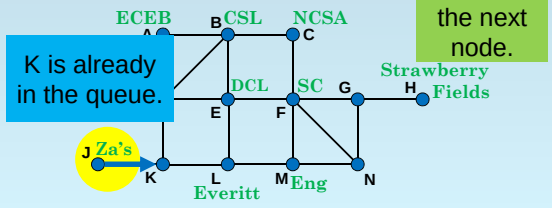


E is already in the queue.

M is already in the queue.

Process Node J by Adding all Neighbors

explored	√	√	√	√	√	√	√	√	√				
queue	B	A	D	E	C	K	L	F	J	M	N	G	
previous	-	B	B	B	B	D	E	E	K	L	F	F	

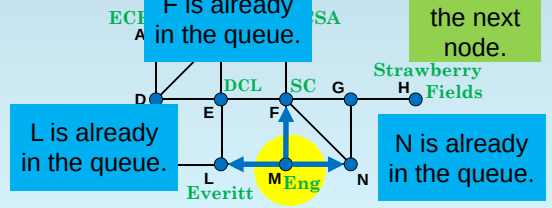


K is already in the queue.

Process the next node.

Process Node M by Adding all Neighbors

explored	√	√	√	√	√	√	√	√	√	√			
queue	B	A	D	E	C	K	L	F	J	M	N	G	
previous	-	B	B	B	B	D	E	E	K	L	F	F	



F is already in the queue.

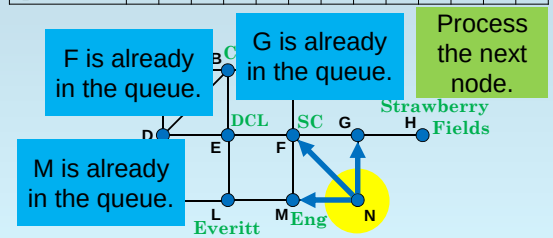
Process the next node.

L is already in the queue.

N is already in the queue.

Process Node N by Adding all Neighbors

explored	√	√	√	√	√	√	√	√	√	√	√		
queue	B	A	D	E	C	K	L	F	J	M	N	G	
previous	-	B	B	B	B	D	E	E	K	L	F	F	



F is already in the queue.

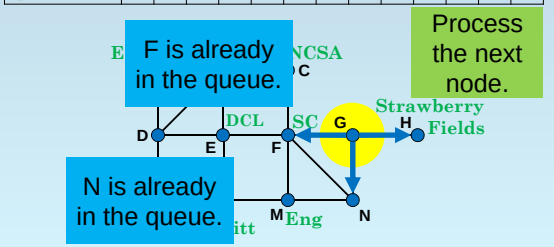
G is already in the queue.

Process the next node.

M is already in the queue.

Process Node G by Adding all Neighbors

explored	√	√	√	√	√	√	√	√	√	√	√	√	
queue	B	A	D	E	C	K	L	F	J	M	N	G	H
previous	-	B	B	B	B	D	E	E	K	L	F	F	G



F is already in the queue.

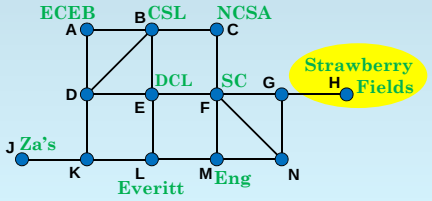
Process the next node.

N is already in the queue.

We Found Strawberry Fields!

queue	B	A	D	E	C	K	L	F	J	M	N	G	H
previous	-	B	B	B	B	D	E	E	K	L	F	F	G

We reached our goal! Now for the path…



Use the Previous Nodes to Find the Path Backwards

queue	B	A	D	E	C	K	L	F	J	M	N	G	H
previous	-	B	B	B	B	D	E	E	K	L	F	F	G

How did we get to H? **From G.**

How did we get to G? **From F.**

How did we get to F? **From E.**

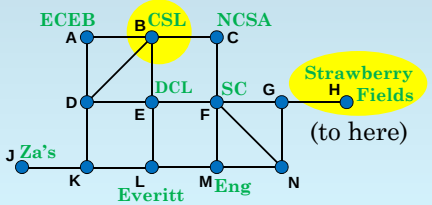
How did we get to E? **From B.**

Now We Have a Way to Find Shortest Paths!

So we found …

B → E → F → G → H? Great!

(from here)



(to here)

Breadth-First Search Finds Short Paths Quickly

distance	0	1				2			3				4
queue	B	A	D	E	C	K	L	F	J	M	N	G	H
previous	-	B	B	B	B	D	E	E	K	L	F	F	G

The approach that we used is called **breadth-first search (BFS)**.

It explores nodes **in order of distance** (see the line on top of our queue).

So you can use BFS with a commercial map database and still find a path just as quickly.

Is BFS Artificial Intelligence?

BFS was invented by E. F. Moore (remember Moore machines?).

After you have seen BFS, it seems pretty simple.

But people used to think of it as "artificial intelligence."

The Point: Algorithms Can Be Subtle

Finding the simplest algorithm to solve a problem can be challenging.

Experience helps.

Classes help.

All CompEs must take **CS225 (Data Structures)** and **CS374 (Algorithms)**, so you have time to learn.

Don't worry about finding ideal solutions in our class.

Build Good Habits Now

What can you do now?

As mentioned earlier,

- Always start with a mental model of your code (for now, as a flow chart on paper).
- Write lots of comments.
- Structure your code clearly:
  - spaces for binary instructions, and
  - indentation and alignment for C/assembly.

Some Other Tips as You Get Started

Avoid repetition

- Reuse code instead of cutting and pasting.
- Every time you copy code, you copy any bugs that the code contains.
- You will be surprised by how often
  - you have to track down a bug that you thought you had already eliminated.
  - Cutting and pasting produces such errors.
  - Different programmers on your team can also produce such errors.

Some Other Tips as You Get Started

Design your code to simplify testing

- You have finite time.
- A program that is hard to test will be tested less often and less thoroughly.

For example,

- the GUI (graphical user interface) to the LC-3 simulator
- simply issues commands to the command-line simulator.
- Why? The command-line simulator can be tested by a computer!