

# Graph Algorithms

“Graph Algorithms”

[http://people.sc.fsu.edu/~jburkardt/presentations/asa\\_2011\\_graphs.pdf](http://people.sc.fsu.edu/~jburkardt/presentations/asa_2011_graphs.pdf)

.....

ISC4221C-01:

Algorithms for Science Applications II

.....

John Burkardt

Department of Scientific Computing  
Florida State University

Spring Semester 2011



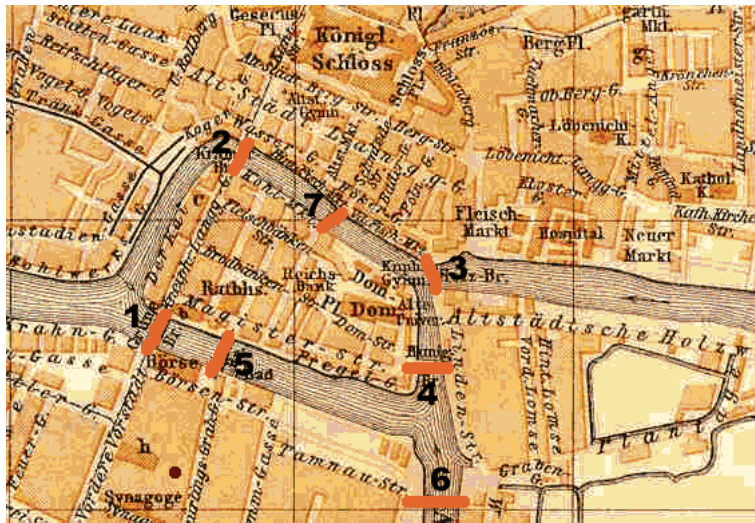
# Graph Algorithms

- **Overview**
- Representing a Graph
- Connections
- The Connection Algorithm in MATLAB
- Components
- Adjacency
- Depth-First Search
- Weighted Graphs
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- Permutations
- The Traveling Salesman
- Projects



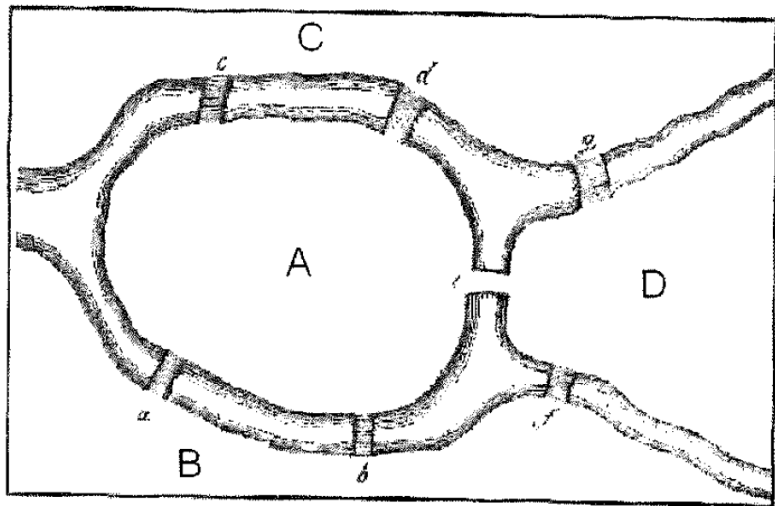
# OVERVIEW: Who Invented Graphs?

## Euler's Königsberg Bridge Problem (1735)



# OVERVIEW: Who Invented Graphs?

Euler's Drawing Selects the Important Information



# OVERVIEW: What is a Graph?

We have a set of **points** (or *nodes* or *vertices* or *cities* ).

A pair of points is called an **edge** (or *link* or *arc* or *road*), and might represent some relationship between points.

A **graph is a list of edges**.

The edge list for the graph of the bridge could be written this way:

{ {A,C}, {A,C}, {A,B}, {A,B}, {A,D}, {C,D}, {B,D} }.

Do we allow repeated edges?

Is {A,C} the same as {C,A}?

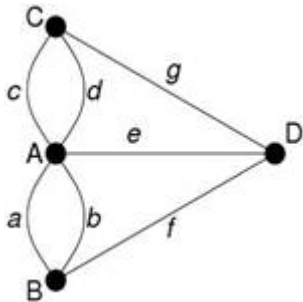
Can we have {A,A}?

Do we care about the *length* of the bridge?



# OVERVIEW: How are Graphs Displayed?

The Abstract Form of the Bridge Problem



# OVERVIEW: Why are Graphs Useful?

For the bridge problem, drawing the graph allows us to drop all the unimportant information.

For small problems, the picture of a graph can be a very effective device for displaying information.

For larger problems, a computer can extract hidden information from a graph, and there are standard ways of representing a graph as data.

The idea of a graph arises in many fields; the same mathematical algorithms can be applied to problems that originally would seem to have nothing in common.



# OVERVIEW: How are Graphs Used

Graphs can be used to organize and analyze a simplified geometry. For example, how are the US States connected?

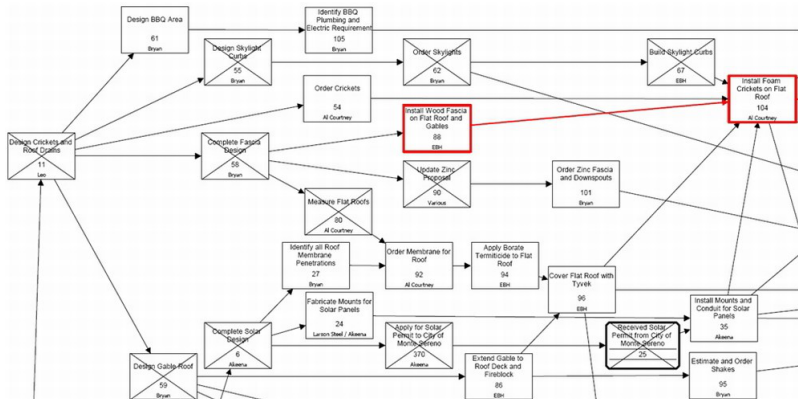








# OVERVIEW: How are Graphs Used



# OVERVIEW: What is Computable on a Graph?

During this series of lectures on graphs, we will study algorithms to answer a few common graph problems, including:

- 1 what points can be reached from a given point?
- 2 how many different routes are there between two points?
- 3 what is the shortest route between two points?
- 4 what is the shortest round trip visiting all points?
- 5 what is the shortest set of "roads" possible that leave all points connected?



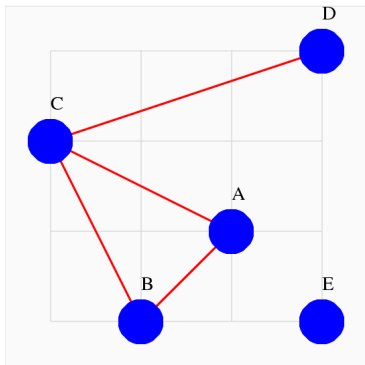
# Graph Algorithms

- Overview
- **Representing a Graph**
- Connections
- The Connection Algorithm in MATLAB
- Components
- Adjacency
- Depth-First Search
- Weighted Graphs
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- Permutations
- The Traveling Salesman
- Projects



# REPRESENT: The SIMPLE Example Graph

How could we store the information for this graph?



# REPRESENT: The Edge List

Mathematically, perhaps the simplest representation would be as an **edge list**.

Each of the **M** edges is described by its pairs of nodes.

For this example, we would have:

$$\mathbf{EdgeList} = \{\{A, B\}, \{A, C\}, \{B, C\}, \{C, D\}\}$$

Does this list adequately represent our graph?

What's missing?

What ways could we use to improve the representation?

Are there graphs for which this representation is not useful?



# REPRESENT: Storing an Edge List

We prefer to draw graphs in which the nodes have labels such as 'A', 'B', 'C' or 'Pittsburgh' and 'Cincinnati'.

On a computer, it's easier to work with an edge list in which the nodes are identified by a numeric index. The edge list for our simple graph could be stored in MATLAB as:

```
EdgeList = [  
    1,  2;  
    1,  3;  
    2,  3;  
    3,  4 ];
```

If the original labels were important, we can arrange to store them in a second array and access them when needed.





# REPRESENT: The Adjacency Matrix

We can also represent a graph by the **adjacency matrix**. If we have **N** nodes, the adjacency matrix **Adj** will be an **N** by **N** matrix, and the **(I,J)** element is:

$$Adj_{i,j} = \begin{cases} 1, & \text{if nodes I and J are connected;} \\ 0, & \text{otherwise.} \end{cases}$$

**Quiz:** since we are working with “simple” graphs, what do we assume about the value of  $Adj_{i,i}$ ?

*Hard question:* a different ordering of the nodes will give a different adjacency matrix for the same graph. Is it possible to determine when two adjacency matrices describe the same graph?



# REPRESENT: The Adjacency Matrix

The adjacency matrix for the SIMPLE graph is:

$$\mathbf{Adj} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The edge list for this graph only contained 4 items of information.  
Can you see that **Adj** also really only has 4 items?



# REPRESENT: The Adjacency Structure

A *sparse graph* involves many nodes  $\mathbf{N}$ , and a few edges per node. The adjacency matrix is enormous, and mostly zero.

A better way of storing the information for a sparse graph is called the **adjacency structure**. It requires the ability to make a list of  $\mathbf{N}$  sublists, where the size of the sublists varies.

The sublist for each node gives all its adjacent nodes.

The SIMPLE graph could be represented by this list:

Node	Sublist(Node)
1	-> 2, 3;
2	-> 1, 3;
3	-> 1, 2, 4;
4	-> 3;
5	-> emptyset;



## REPRESENT: The Adjacency Structure

To implement an adjacency structure, we use MATLAB's **cell array** data type, which is like a matrix or vector, except that the entries are not numbers but lists, strings, or other quantities.

```
A = cell(5,1);      <-- Set up a list of 5 sublists;
```

```
A{1} = [ 2, 3 ];   <-- neighbors of node 1;
```

```
A{2} = [ 1, 3 ];   <-- neighbors of node 2;
```

```
A{3} = [ 1, 2, 4 ]; <-- neighbors of node 3;
```

```
A{4} = [ 3 ];      <-- neighbors of node 4;
```

```
A{5} = [];         <-- neighbors of node 5;
```

```
length(A{1})      <-- returns 2; Count node 1 neighbors.
```

```
A{1}(2)           <-- returns 3; Neighbor 2 of node 1.
```



# REPRESENT: The Incidence Matrix

The **incidence matrix** is a way of representing a graph that includes information about both the node and the edges.

Now we assume we have **N** nodes and **M** edges, and that both the nodes and the edges have been numbered.

The **I**-th row of the incidence matrix describes the **I**-th edge. It indicates which two nodes are joined by that edge.

The **J**-th column of the incidence matrix describes the **J**-th node. It indicates which edges touch that node.

For our SIMPLE graph, we have:

$$\mathbf{Inc} = \begin{bmatrix} & A & B & C & D & E \\ AB : & 1 & 1 & 0 & 0 & 0 \\ AC : & 1 & 0 & 1 & 0 & 0 \\ BC : & 0 & 1 & 1 & 0 & 0 \\ CD : & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$



# REPRESENT: An Adjacency Function

For some problems, it might be natural to use an **adjacency function**. In other words, instead of having an explicit table of 0's and 1's, we provide a function **adjacent**( $i,j$ ) which is given the indices or other information about the two nodes we are interested in, and returns the adjacency status of 0 or 1.

This might seem a strange situation, except that there are cases where the adjacency relationship can be determined naturally in this way.

Suppose we have a table of four letter words, and we are interested in the puzzle in which we try to change one word into another by altering one letter at a time, with the requirement that after each single letter change, the result is still a legal word.



# REPRESENT: An Adjacency Function

We might try, for instance, to convert **head** to **tail**:

head

hea**l**

**t**ea**l**

te**l**l

ta**a**ll

ta**i**l

If we have a list of four letter words, then we could search for legal moves by asking questions such as **adjacent('head','heal')** which would return 1 while **adjacent('head','foot')** returns 0. In this case, the input items are computed to be adjacent if they differ in exactly one spot.

You should see that for this case, an adjacency function is an extremely compact way of determining the adjacency information. A more realistic example of this kind of adjacency computation might involve biological sequence data.



# REPRESENT: The GRF format

When we think about a graph, we really think of it as a drawing.

Often a graph can be drawn in a way that keeps edges from crossing, and that looks clean and logical.

When the presentation of the graph is important, the **GRF** format can be used.

Each line of the file looks like this:

```
i x y n1 n2 ... nk
```

where

- $i$  is the index of the node (1 to **N**);
- $x y$  are the coordinates of the point where the node is drawn;
- $n1, n2, \dots, nk$  are the nodes that connect to this node.





# REPRESENT: The GRF format

A GRF file for the SIMPLE graph could look like this:

```
1 2.0 1.0 2 3
2 1.0 0.0 1 3
3 0.0 2.0 1 2 4
4 3.0 3.0 3
5 3.0 0.0
```



## REPRESENT: GRF\_DISPLAY can display the GRF format

The MATLAB program **grf\_display** can be used to display a graph that has been stored in the **GRF** format.

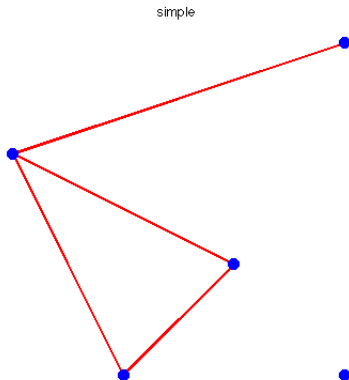
If the file is called *simple.grf*, then you could display it with the command

```
grf_display simple
```



# REPRESENT: GRF\_DISPLAY shows the Simple Graph

Here is how `grf_display` would display the simple graph:



# REPRESENT: The Degree of a Node

The **degree** of a node is the number of edges that begin there.

For any representation, we can determine the degree of a node:

- *edge list*, look at every edge, and count the number of times the node occurs;
- *adjacency matrix*, find either the row or column corresponding to the node, and count the number of 1's;
- *adjacency structure*, count the number of entries in the sublist for the node;
- *incidence matrix*, find the column corresponding to the node, and count the 1's;
- *GRF file*, find the line of the file corresponding to the node and count the neighbors;



## REPRESENT: Summary

The **edge list** is simple to implement and doesn't waste space; however, the data is not organized; to find neighbors of a node, you must look at every item.

The **adjacency matrix** is simple to implement, and makes it easy to find the neighbors of a node; it does waste a lot of space.

The **adjacency structure** is a little harder to set up, but makes it very easy to list the neighbors of any node, and doesn't waste space. However, if the adjacency structure is defined as a MATLAB cell array, you can't use **load** and **save** to write it to a file or read it back.

(The adjacency structure can be set up as a list and a pointer vector, in which case **load** and **save** can be used...)



# Graph Algorithms

- Overview
- Representing a Graph
- **Connections**
- The Connection Algorithm in MATLAB
- Components
- Adjacency
- Depth-First Search
- Weighted Graphs
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- Permutations
- The Traveling Salesman
- Projects



# CONNECTIONS: The Global View

When we describe a graph, for instance by an edge list, we are stating which pairs of nodes are directly connected by an edge. This is all we need to say in order to define the graph.

Edges are local connections; they don't go very far. But when we draw the graph, we can see **global** information, that is, “the big picture”. One of the most important global properties that a graph can have involves **connectivity** or **connectedness**.

This involves questions such as:

- *Can we travel from node **A** to node **Z**?*
- *Can we travel from any node to any other node?*
- *What is the quickest way to get from **G** to **W**?*

To answer these questions, we seek algorithms to turn local information into global knowledge.



## CONNECTIONS: Kinds of Graphs

A **loop** is an edge that joins a node to itself.

If there may be more than one edge between two nodes, we have a **multi-graph** (Example: the Euler bridge graph).

If each edge has a length, or some other quantity associated with it, we have a **weighted graph**.

Some of our later example graphs will include edge lengths; that will allow us to find shortest driving routes, and so on.

If edges have “direction”, that is,  $\{A,C\}$  is not the same thing as  $\{C,A\}$ , then we say we have a **directed graph**, or “digraph”.

In many cases, the model you are looking at requires the edges to have direction. A graph of the traffic pattern of downtown Tallahassee would require many one-way streets. If we have time, we will look at some interesting digraph problems.





## CONNECTIONS: Simple Graphs, Adjacent Nodes

A graph that is not directed, has no multiple edges, has no loops, and no edge weights is **a simple graph**.

In a simple graph, there either is or isn't exactly one edge between any two distinct nodes, and that's all we can say.

In a simple graph, edges don't have "direction", so we are free to describe an edge as a set,  $\{A,B\}$  or  $\{B,A\}$ . Either form means the same thing.

If there is an edge between two nodes, we say the nodes are **adjacent** (that is, directly touching).



## CONNECTIONS: Walks and Paths

If we think about edges as connecting pairs of nodes, then even if two nodes are not adjacent, it might be possible to reach one node from the other by taking one edge, then another and another.

If the edges make it possible, we can describe the process of moving from one node to the other in two ways:

- as a *node list*, simply listing the nodes in the order we visited them, including the start and finish;
- as an *edge list*, listing the edges we used, in order.

A list of edges that describes a journey has a special property, of course. Each consecutive pair of edges must have a node in common, since one edge got us to that node, and one edge took us away from it.



## CONNECTIONS: Walks and Paths

We define a **walk** from **A** to **B** as node list with the properties that the first element is **A**, the last element is **B**, and every consecutive pair of nodes are adjacent.

An example of a walk from **A** to **D** on our simple graph might be **A-C-B-C-D**.

**QUIZ:** Show there are an infinitely many walks from **A** to **D**!

We define a **path** as a walk in which no edge is used more than once. Thus, our walk example is not a path.

**QUIZ:** Exactly how many paths are there from **A** to **D**?

For our work, paths will be much more important and interesting than walks!



# CONNECTIONS: Connected Nodes and Graphs

If there is a path from node **A** to node **B**, then we say the nodes are **connected**.

If no node is connected to some node **A**, we say **A** is **isolated**.

If every pair of points in a graph are connected, then we say we have a **connected graph**; otherwise, it is **disconnected**.

If a graph is not connected, the nodes and edges can be organized into “subgraphs”. Every node in the first subgraph can reach the other nodes in that subgraph, but can't reach nodes in any other subgraph. The subgraphs are like islands.

These subgraphs are the **connected components** of the graph.



## CONNECTIONS: Computation?

Let us assume we are working with a graph  $\mathcal{G}$  which is simple (no multiple edges, no edge directions, no special edge weights).

Suppose we have the picture of the graph, and that there are nodes labeled **A** and **B**, and that you have a red pencil and an eraser.

How would you try to answer the following question:

*Can you find and draw a path from **A** and **B**?*

If the graph is really complicated, is there a “sure-fire” way that would be guaranteed to always work?



## CONNECTIONS: Computation?

Now suppose that the graph has been stored in the computer, and we are given the indices of two nodes, **A** and **B**.

- 1 One task is simply to report whether there is a path.
- 2 A second task would be to actually report what that path is!

*Is there a computational algorithm that can find a path from **A** and **B** if there is one?*

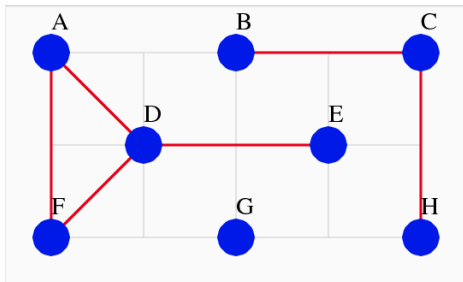
This might be related to the “sure-fire” way we talked about when we were solving the problem using a picture of the graph and a red pencil...



# CONNECTIONS: The DISCONNECTED Example Graph

Can we **compute** a path from **A** to **E**?

Can we **compute** the number of “connected components”, that is, pieces of this graph?



# CONNECTIONS: The Connection Algorithm

The **Connection Algorithm** seeks a path from a node **A** to another node, say, **B**. All we have to work with is the list of edges.

We will divide the nodes into three sets: *used*, *new*, *untouched*.

We begin with:

- $used = \emptyset$
- $new = \{ A \}$
- $untouched = \{ \text{all nodes but } A \}$





# CONNECTIONS: The Connection Algorithm

We now carry out the following steps:

- 1 If *new* is empty, we failed. Exit.
- 2 Move one item from *new* into **used**, and call it *S*.
- 3 For each edge which uses *S*:
  - Call the other node in the edge *T*;
  - If *T* is equal to **B**, we succeeded. Exit.
  - Otherwise, if *T* was *untouched*, move it into *new*.
- 4 Go back to step 1.

**Do this now:**

Use the Connection Algorithm to seek a path from **A** to **E**.  
How about a path from **H** to **A**?



# Graph Algorithms

- Overview
- Representing a Graph
- Connections
- **The Connection Algorithm in MATLAB**
- Components
- Adjacency
- Depth-First Search
- Weighted Graphs
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- Permutations
- The Traveling Salesman
- Projects



# MATLAB: Implementing the Connection Algorithm

We can see things, such as a path, that are hard to turn into an algorithm.

It turns out that we even if we can write up an algorithm, it can be harder still to turn that algorithm into an **implementation**, that is, a complete program in some computer language that carries out the algorithm.

Since the Connection Algorithm seems reasonably simple, let's take some time to think about the choices we have when writing it up in MATLAB.



## MATLAB: Problem Data

Let's call the number of nodes **N**; for the DISCONNECTED example, **N=8**;

Represent the graph by an edge list called **edge**:

```
1 4
1 6
2 3
3 8
4 5
4 6
```

The **edge** array has **NEDGES** rows and 2 columns. We can get the value of **NEDGES** by the command

```
nedges = size ( edge, 2 )
```

where “2” means we're asking for the second dimension.

That's all the data we have to start with.



Our algorithm requires us to define and use some sets.

In MATLAB, we could think of the sets *used*, *new* and *untouched* simply as vectors of length  $\mathbf{N}$ . The  $\mathbf{T}$ -th value in each vector is 1 if  $\mathbf{T}$  is in that set, and 0 otherwise.

A node should only be in one set. So if  $\mathbf{T}$  is in *new*, we must make sure that

```
used(t) = 0;  
new(t) = 1;  
untouched(t) = 0;
```



# MATLAB: Modifying Sets

For the Connection Algorithm, we start with all the nodes in the *untouched*, set, so our MATLAB code begins with:

```
used = zeros(1,n);  
new = zeros(1,n);  
untouched = ones(1,n);
```

Now we pick **any** entry of *untouched* as our starting point and move it into *new*. Let's assume that's node **F**, which is index 6:

```
untouched(6) = 0;    <-- Node 6 moves out of untouched  
new(6) = 1;         <-- and into our work area;
```



# MATLAB: Step 1: Initialize

So here's what our sets look like as we begin the program:

Label:		A	B	C	D	E	F	G	H
Index:		1	2	3	4	5	6	7	8
-----+									
UNTOUCHED:		1	1	1	1	1	0	1	1
NEW:		0	0	0	0	0	1	0	0
USED:		0	0	0	0	0	0	0	0



## MATLAB: Step 2: Select From New

Step 2 moves one item from *new* to *used* and calls it **S**.

```
s = 0;
for i = 1 : n
    if ( new(i) ~= 0 )
        s = i;
        break
    end
end
new(s) = 0;
used(s) = 1;
```

The **find** command can also locate **S**:

```
i = find ( new ~= 0 )
s = i(1);
```





## MATLAB: Step 3: Seek Untouched Neighbors

```
for j = 1 : nedges
    if ( edge(1,j) == s )
        t = edge(2,j);
    elseif ( edge(2,j) == s )
        t = edge(1,j);
    else
        t = 0;
    end
    if ( t ~= 0 )
        if ( untouched(t) )
            untouched(t) = 0;
            new(t) = 1;
        end
    end
end
end
```



## MATLAB: After Step 3

After step 3, we've put more stuff into **new**:

Label:		A	B	C	D	E	F	G	H
Index:		1	2	3	4	5	6	7	8
-----+-----									
UNTOUCHED:		0	1	1	0	1	0	1	1
NEW:		1	0	0	1	0	0	0	0
USED:		0	0	0	0	0	1	0	0

and node **F** has retired into *used*, and we are ready to go back to step 1 and seek untouched neighbors of node **A** or **D**.



# Graph Algorithms

- Overview
- Representing a Graph
- Connections
- The Connection Algorithm in MATLAB
- **Components**
- Adjacency
- Depth-First Search
- Weighted Graphs
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- Permutations
- The Traveling Salesman
- Projects



# COMPONENTS: Success From Failure

Our Connection Algorithm can fail if there is no path between the nodes we specified. The algorithm even realizes that it has failed, when it runs out of nodes in the *new* set.

If our algorithm fails, we know that the graph must have at least two components.

**Quiz:** If the algorithm *doesn't* fail, is there just one component?

In the spirit of “If at first you don’t succeed, fail, fail again!”, we can actually use the failures of the connection algorithm to tell us exactly how many connected components there are, and we can even list the nodes that belong to each one.

**How can we do this?**



## COMPONENTS: Finding One Connected Component

Notice that our Connection Algorithm is gradually discovering every node that can be reached from the starting node. It actually usually stops a little early, that is, as soon as it finds the special target node we called **B**.

Suppose we remove the line that says:  
*“if you reach **B**, you have succeeded.”*

Then the algorithm will run and run until every node that could be reached from **A** has been checked and tossed into the *used* group.

That set is exactly one connected component of the graph!



# COMPONENTS: Finding Other Connected Components

If the graph was connected, then we're actually done.

How can we tell whether we are done?

Assuming we aren't done, how can we find another component?



# COMPONENTS: The Connected Component Algorithm

Initialize *used* and *new* to  $\emptyset$ .

Initialize *untouched* to all the nodes.

Set the component count to  $\mathbf{c} = 0$ ;

- 1 If *untouched* is empty, successful exit.
- 2 Move one node from *untouched* to *new*.
- 3  $\mathbf{c} = \mathbf{c} + 1$ ;
- 4 Call the modified Connection Algorithm;
- 5 Go back to step 1.

At the end,  $\mathbf{c}$  is the number of components we found.

The modified Connection Algorithm can even keep track of which component each node belongs to.



# COMPONENTS: The Modified Connection Algorithm

Assume  $c$  and the sets *used*, *new*, *untouched* have been assigned.

- 1 If *new* is empty, exit.
- 2 Remove one item from *new*, and call it  $S$ .
- 3 For each edge which uses  $S$ :
  - Call the other node in the edge  $T$ ;
  - If  $T$  was *untouched*, move it into *new*.
- 4 **component(S)=c;**
- 5 Move  $S$  into *used*;
- 6 Go back to step 1.





# Graph Algorithms

- Overview
- Representing a Graph
- Connections
- The Connection Algorithm in MATLAB
- Components
- **Adjacency**
- Depth-First Search
- Weighted Graphs
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- Permutations
- The Traveling Salesman
- Projects



# ADJACENCY: The Adjacency Matrix

We said that two points **A** and **B** are adjacent if there is an edge that directly connects them. For a simple graph, the adjacency information contains a complete description of the graph. For each pair of nodes, we need a “yes” or “no” answer.

Recall our definition of the **the adjacency matrix**. If we have **N** nodes, the adjacency matrix **Adj** will be an **N** by **N** matrix, and the **(I,J)** element is:

$$Adj_{i,j} = \begin{cases} 1, & \text{if nodes I and J are connected;} \\ 0, & \text{otherwise.} \end{cases}$$

**QUIZ:** What property does this matrix have?  
(Hint: what is the transpose of this matrix?)

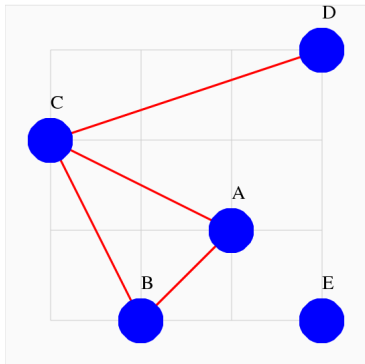
**QUIZ:** If we changed this property, what kind of graph have we created?



# ADJACENCY: Computing Adjacency from a Picture

What is the adjacency matrix for the SIMPLE graph?

How can we tell that node **E** is completely disconnected?



# ADJACENCY: Using the Matrix

The adjacency matrix is:

$$Adj = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Recall that the same graph can be represented by this edge list

1 2  
1 3  
2 3  
3 4

Do you see there are really only **4** pieces of information in **Adj**?



## ADJACENCY: Computing Adjacency from a Picture

Since the adjacency matrix **Adj** is a matrix, it's natural to wonder whether you can do numerical operations with it.

What would it mean to multiply this matrix times a vector? Let's take the column vector  $v = [0, 1, 0, 0, 0]' = [0, 1, 0, 0, 0]^T$  for instance. Computing  $Adj * v$  we get  $[1, 0, 1, 0, 0]'$ . That's just column 2 of **Adj**.

We can interpret the result as follows. Setting  $v_0 = [0, 1, 0, 0, 0]'$  means we are starting at node 2 (B). Multiplying by the matrix **Adj** returns us  $Adj * v_0 = v_1 = [1, 0, 1, 0, 0]'$ ; that is, it says we can move to node 1 or 3 in one step.



# ADJACENCY: A “MATLAB Moment”

When I want to talk about the column vector

$$v = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

I wrote, instead,  $v = [0,1,0,0,0]'$ , for convenience. There are different conventions in MATLAB and mathematics for indicating a transpose. Here are three ways of writing that vector:

$v = [0,1,0,0,0]'$ ;    <-- MATLAB transpose operator;  
 $v = [0;1;0;0;0]$ ;    <-- semicolons end a row.  
 $v = [0,1,0,0,0]^T$     <-- Mathematical format;



# ADJACENCY: Computing Adjacency from a Picture

Another multiplication gets us  $Adj * v1 = v2 = [1, 2, 1, 1, 0]$ .

Let's try to explain the second entry, which is 2. Because this is in the second position, it's talking about node 2.

After one step, we were at node 1 or 3. Taking a step from node 1 can get us to 2 or 3. Or, from 3, we can get to 1, 2 or 4:

	1	2	3	4	5
From node 1, can move to:	-	1	1	-	-
From node 3, can move to:	1	1	-	1	-
Number of ways to reach:	1	2	1	1	0

So this is how many different ways to start at node 2, take 2 steps, and end up at node 2, namely B-A-B and B-C-B.



# ADJACENCY: Counting Paths From A Starting Point

Let the vector  $\mathbf{v}$  be zero except for a 1 in the entry corresponding to some node  $\mathbf{A}$  in the graph  $\mathcal{G}$ , and let  $\mathbf{Adj}$  be the adjacency matrix for  $\mathcal{G}$ .

Then the vector:

- $\mathbf{Adj} * \mathbf{v}$  counts walks of length 1 from  $\mathbf{A}$  to each node;
- $\mathbf{Adj}^2 * \mathbf{v}$  counts walks of length 2 from  $\mathbf{A}$ ;
- $\mathbf{Adj}^3 * \mathbf{v}$  counts walks of length 3, and so on.

Recall that a walk from  $\mathbf{A}$  to  $\mathbf{B}$  is absolutely *any* sequence of nodes which are joined by edges and which begin at  $\mathbf{A}$  and end at  $\mathbf{B}$ .

Such a walk may use an edge more than once, pass through any node more than once, and in fact, pass through  $\mathbf{B}$  several times.





# ADJACENCY: Counting Paths From Any Starting Point

Let **Adj** be the adjacency matrix for a graph  $\mathcal{G}$ .

Then the matrix:

- $Adj_{i,j}$  counts walks of length 1 from node  $i$  to node  $j$ ;
- $Adj_{i,j}^2$  counts walks of length 2,
- $Adj_{i,j}^3$  counts walks of length 3, and so on.

**QUIZ:** How can we determine whether the graph  $\mathcal{G}$  is connected?

- 1 If  $\mathcal{G}$  is connected, what must be true for any pair of nodes?
- 2 What is the maximum length of a path between two nodes?
- 3 When you count walks, you are including paths.



## ADJACENCY: What about Paths Instead?

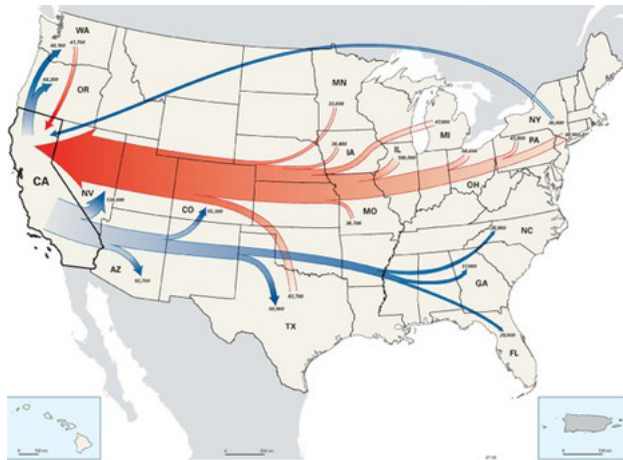
As we keep multiplying the adjacency matrix, the entries in the result vector keep growing. But this is, in a sense, way too much information. On a graph with 10 nodes, discovering a walk of 100 steps is not difficult, not interesting, and probably not so useful.

What might be more interesting is to figure out a corresponding procedure that works out the number of paths from one node to another. This is harder (because the adjacency matrix trick doesn't simply hand us the answer) but more interesting, because paths are more likely to be what we are looking for, and because we can guarantee that the procedure stops, for sure, after at most  $N-1$  steps. (...and that's because???)

What sort of algorithm could we sketch that might handle this problem? (We will discuss this for a few minutes in class.)



# ADJACENCY: California Population Exchange



RED: Population movement 1955:1960

BLUE: Population movement 1995:2000



## ADJACENCY: Modeling Transitions

Each year,  $1/10$  of the people outside of California move in; at the same time,  $2/10$  of the people in California move out!

This may not sound like a graph problem, but let's think about whether we could get somewhere anyway.

We do have places (California, and not-California) and connections (move from one place to the other.) Now the connections have a direction.  $C \rightarrow N$  means move from California to Noncalifornia, but  $N \rightarrow C$ , means I move the other way.

Moreover, if some people move out of California, some people stay. So if we think about the links as actions, then we have to add loops for this model, that is, a link that goes from C to C, and another that goes from N to N.

Finally, we have the probabilities, which we use to label the links.

**Do this now:** Draw the graph for this model.



## ADJACENCY: Modeling Transitions

Suppose we create a new adjacency matrix, which models this graph. Instead of a 1 or a 0, we put the probability of the event that the link represents.

	C	N
C	0.8	0.1
N	0.2	0.9

This means we have to agree on how to “read” the graph. So the rows will be where we move TO, and the columns will be where we move FROM. Assuming California is the first row and column, then  $A_{1,2}$  is the probability that I just moved this year TO California FROM Noncalifornia.

Now the US population is about 300 million, and California's is about 37 million. That's too hard, so let's call it 40 million! Now let's say that the simple model we have built is exactly what happens, and let's forget about births and deaths. Every year, 1/10 move in, 2/10 move out.



# ADJACENCY: Modeling Transitions

Just like with Euler's bridge problem, the first time you see a problem like this, you might say, "This is not math. It has numbers in it, but it is not mathematics!" We will see a little of the math today, and actually this problem opens up a huge world of simulation, linear algebra, probability, Markov chains, and on and on!

So what happens if we let  $\mathbf{A}$  be the adjacency matrix...no, let's call this the **transition matrix!**, and let  $\mathbf{v}$  be the population vector [40, 260], and compute  $\mathbf{A} * \mathbf{v}$ ?

$$\begin{aligned} \mathbf{A} * \mathbf{v} &= 0.8 * 40 + 0.1 * 260 = 58 \\ &= 0.2 * 40 + 0.9 * 260 = 242 \end{aligned}$$

Notice one good thing: we still have 300 million people! Also, California just got very crowded in one year!



# ADJACENCY: Modeling Transitions

Since this is a model of yearly change, let's see what happens over time. Presumably, everyone will end up living in California!

So what happens if we let  $\mathbf{A}$  be the adjacency matrix...no, let's call this the **transition matrix!**, and let  $\mathbf{v}$  be the population vector [40, 260], and compute  $\mathbf{A}^* \mathbf{v}$ ?

40	58	71	79	86	90	93	95	97	98	98	99	...
260	242	229	221	214	210	207	205	203	202	202	201	...

Wait a minute, this looks like math! And I think we can guess what's going to happen. Can you also guess why? Do you know the name for a vector  $\mathbf{v}$  with the property that  $\mathbf{v} = \mathbf{A}^* \mathbf{v}$ ?



## ADJACENCY: Summary

The simple idea of the adjacency matrix seemed almost magical when it was able to produce the number of walks from any node to any other.

With a little thought, we could also figure out a procedure for getting the number of paths, rather than walks, which might be more useful.

But, to give you just a hint of why this stuff is actually so interesting, we were able to make a complicated directed graph with edge weights and loops, which modeled (even if it's an inaccurate model) a real situation, and which even ended up turning into some math for us!





# Graph Algorithms

- Overview
- Representing a Graph
- Connections
- The Connection Algorithm in MATLAB
- Components
- Adjacency
- **Depth-First Search**
- Weighted Graphs
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- Permutations
- The Traveling Salesman
- Projects



# DFS: A Way of Checking All Nodes

In our search for connections and components and paths, we struggled to find a way to visit every node of a graph. The long-range connections of a graph are not obvious from the data, but this is extremely useful information to obtain.

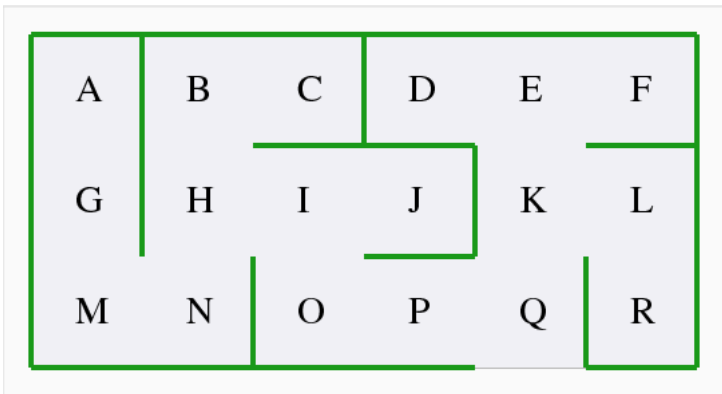
One standard technique for visiting every node of a graph is known as **depth-first search**; its name indicates that in this process, we follow one branch of possibilities all the way to its end, while remembering all the other possibilities that we must come back and examine later.

One version of the search uses *recursion*: the procedure calls itself. We will look at an equivalent version that uses a *stack*. For us, a stack is a list of unfinished work; the last item placed on the stack is the first thing we will take off.



# DFS: Visiting a Museum

Consider the problem of visiting every room of this museum, which we enter at room Q:



## DFS: Visiting a Museum

Assuming perfect memory, we could go as far as possible in new territory, then back up to the last unexplored room.

Q - use K, save P

K - use E, save L

E - use D, save F

D - dead end, backup to E

E - use F

F - dead end, backup to E

E - nothing left, backup to K

K - use L

L - use R

R - dead end, backup to L

L - nothing left, backup to K

K - nothing left, backup to Q

Q - use P

*...and so on...*



# DFS: The Algorithm in Words

Our search begins by choosing a starting node. That node has neighbors, and we must visit them all but we can only do one at a time, so we choose one neighbor node to visit, move there, and “remember” the other nodes by putting them onto the stack.

The new node has neighbors. One of its neighbors was the starting point (that’s how we got here) and we don’t want to visit it again. So we will need to remember which nodes we have visited already. Now we look at the unvisited neighbors of this node, pick one, and put the rest on the stack.

We continue doing this until we reach a node that has no unvisited neighbors. We have gone to maximum “depth” on this path.



# DFS: The Algorithm in Words

What next? Simply get the next node off the stack. Now we might actually have visited this node since we put it on the stack; if so, we keep drawing the next item from the stack until we find an unvisited node, and start wandering again.

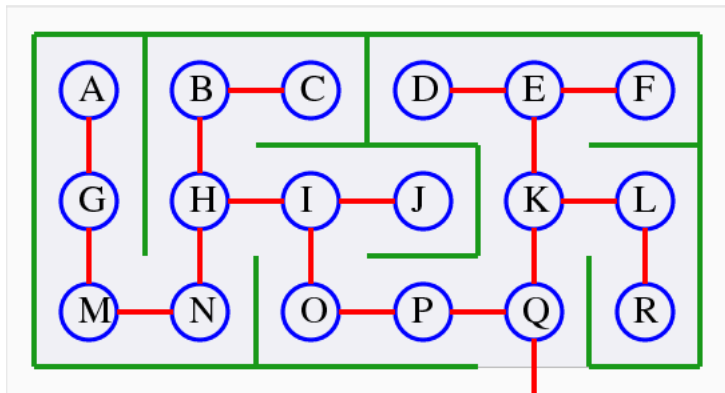
If the stack becomes empty, and we know the graph is connected, we are done.

If the graph is not connected, or we don't know, we simply check for any unvisited node, and restart the algorithm from that point to find some more paths.



# DFS: The Algorithm in Practice

Consider the graph for our museum:



# DFS: The Algorithm in Pseudocode

INITIALIZE

```
step = 0; visited(1:N) = 0;
```

TAKE SOMETHING OFF THE STACK:

```
do
```

```
  if ( step == 0 ) then room = Q
```

```
  elseif ( nothing left on stack ) then we're done
```

```
  else
```

```
    room = last stack item
```

```
    if ( room not yet visited ) then exit
```

```
  end
```

```
end
```

PUT THINGS ON THE STACK

```
step = step + 1; visited(room) = step;
```

```
Add each unvisited neighbor of room to the stack. 80 / 145
```





## DFS: The INITIALIZE Step

```
step = 0;
visited(1:n) = 0;
stack_num = 0;
stack(1:n) = 0;

while ( 1 )
%
% Find an unvisited room.
%
%   code to take something off the stack
%
% Visit new room, look for unvisited neighbors.
%
%   code to put neighbors on the stack
end
```



## DFS: FIND UNVISITED ROOM Step

```
% Find an unvisited room.  
%  
while ( 1 )  
    if ( step == 0 )  
        room = first; (first call)  
        break  
    elseif ( stack_num == 0 )  
        return (we are done)  
    else  
        room = stack(stack_num);  
        stack_num = stack_num - 1;  
        if ( visited(room) == 0 )  
            break;  
        end  
    end  
end  
end
```



## DFS: VISIT NEW ROOM Step

```
% Visit new room, look for unvisited neighbors.
% Use adjacency structure (cell array).
%
    step = step + 1;
    visited(room) = step;  (we have been here now.)

    neighbors = adj{room};  (extract neighbor list)
    neighbor_num = length ( neighbors );

    for i = 1 : neighbor_num
        room2 = neighbors(i);
        if ( visited(room2) == 0 )
            stack_num = stack_num + 1;
            stack(stack_num) = room2;
        end
    end
end
```



## DFS: VISIT NEW ROOM Step

```
% Visit new room, look for unvisited neighbors.
% Use adjacency list + adjacency pointer.
%
    step = step + 1;
    visited(room) = step;  (we have been here now.)

    for i = adj_pointer(room) : adj_pointer(room+1)-1
        room2 = adj_list(i);
        if ( visited(room2) == 0 )
            stack_num = stack_num + 1;
            stack(stack_num) = room2;
        end
    end
end
```



# DFS: Output

Step	Room	Stack	Step	Room	Stack
----	----	-----	----	----	-----
1	Q:	K P	11:	B:	K C
2	P:	K O	12	C:	K
3	O:	K I	13	K:	E L
4	I:	K H J	14	L:	E R
5	J:	K H	15	R:	E
6	H:	K B N	16	E:	D F
7	N:	K B M	17	F:	D
8	M:	K B G	18	D:	empty!
9	G:	K B A			
10	A:	K B			



## DFS: Summary

We have simply used DFS to visit each node. It should be clear that this is a powerful way to process the information in a graph.

If we did not know the graph was connected, we would have to count the number of nodes we found, and if necessary, restart the algorithm by picking an unvisited node at random.

The edges used in the search form a “spanning tree” of the graph; that is, if we only kept those edges, we’d have just enough to keep the graph connected. In our example, there were no excess edges (the graph was already a “tree”).

The **breadth first search** is a related method. It starts at a given node, and visits all the nodes that are just 1 edge away, then unvisited nodes that are just 2 edges away, and so on.



# Graph Algorithms

- Overview
- Representing a Graph
- Connections
- The Connection Algorithm in MATLAB
- Components
- Adjacency
- Depth-First Search
- **Weighted Graphs**
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- Permutations
- The Traveling Salesman
- Projects



# WEIGHT: Measuring the Edges

In simple graphs, an edge exists or doesn't, and that's the end.

This idea of an edge is a little too restricted for some of the real situations we might like to model.

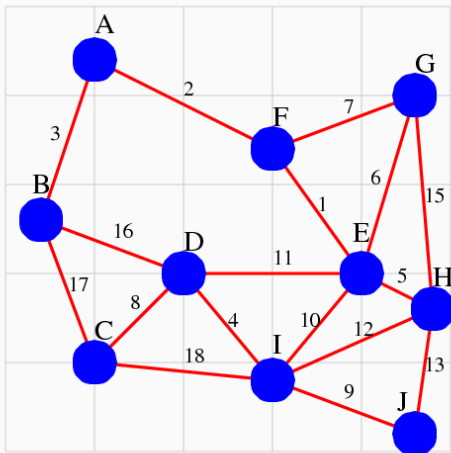
Suppose, for example, our nodes represented cities, and the edges represented direct highway links. A graph could be used to plan a trip, but only if we also knew the *mileage* of each possible highway section.

We can create a new kind of graph which includes a number associated with each edge. Usually, this number is positive, or at least nonnegative. In any case, if we start with a simple graph and assign a length, or weight, or some other numeric value to each edge, we get an **edge-weighted graph** or just **weighted graph**.





# WEIGHT: The MST (Minimum Spanning Tree) Example



## WEIGHT: Questions About a Weighted Graph

If we have a situation that can be modeled by a weighted graph, some new questions can be posed, and answered:

- *What is the shortest path from **A** to **H**?*
- *What is the shortest set of edges we could select that would still keep all cities connected?*
- *What is the shortest round trip that visits all the cities?*



# WEIGHT: Questions About a Weighted Graph

These particular questions are so common that they have been given names:

- The shortest path problem;
- The minimum spanning tree;
- The traveling salesman problem.

For each question, there are algorithms available.

The last problem is much, much harder than the others.



# WEIGHT: The Edge Length Matrix

One way to represent a weighted graph is by the **edge length matrix**. If we have **N** nodes, the edge length matrix **Length** will be an **N** by **N** matrix, and the **(I,J)** element is:

$$Length_{i,j} = \begin{cases} \text{length of the edge between nodes I and J;} \\ \infty, \text{ if nodes I and J are not directly connected.} \end{cases}$$

In MATLAB, the value  $\infty$  is represented by **Inf**, and we are allowed to make comparisons using this value.



# WEIGHT: The Edge Length Matrix

The edge length matrix for our MST graph is:

$$\text{Length} = \begin{bmatrix} 0 & 3 & \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \\ 3 & 0 & 17 & 16 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 17 & 0 & 8 & \infty & \infty & \infty & \infty & 18 & \infty \\ \infty & 16 & 8 & 0 & 11 & \infty & \infty & \infty & 4 & \infty \\ \infty & \infty & \infty & 11 & 0 & 1 & 6 & 5 & 10 & \infty \\ 2 & \infty & \infty & \infty & 1 & 0 & 7 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 6 & 7 & 0 & 15 & \infty & \infty \\ \infty & \infty & \infty & \infty & 5 & \infty & 15 & 0 & 12 & 13 \\ \infty & \infty & 18 & 4 & 10 & \infty & \infty & 12 & 0 & 9 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 13 & 9 & 0 \end{bmatrix}$$



# Graph Algorithms

- Overview
- Representing a Graph
- Connections
- The Connection Algorithm in MATLAB
- Components
- Adjacency
- Depth-First Search
- Weighted Graphs
- **The Shortest Path**
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- Permutations
- The Traveling Salesman
- Projects



## SHORT: What is the Length of a Path?

Recall that a **path**  $\mathbf{P}$  from node  $\mathbf{A}$  to node  $\mathbf{B}$  is a sequence of edges of the form  $\{\mathbf{A},\mathbf{J}\}$ ,  $\{\mathbf{J},\mathbf{W}\}$ ,  $\{\mathbf{W},\mathbf{K}\}$ ,  $\{\mathbf{K},\mathbf{B}\}$ , (for instance), with the property that the first edge includes  $\mathbf{A}$ , the last edge includes  $\mathbf{B}$ , and every consecutive pair of edges has a node in common, with no nodes repeated.

If we have a weighted graph, then the **Length** matrix records the the length of every edge.

Therefore, given a path  $\mathbf{P}$ , we can define the **distance** traveled by a path  $\mathbf{P}$  as the sum of the lengths of the edges it uses.



# SHORT: The Shortest Path Problem

Now suppose we are given a weighted graph  $\mathcal{W}$ .

The **shortest path problem** requires us to determine the distance of the shortest path between a particular pair of nodes **A** and **B**.

The **all pairs shortest path problem** requires us to solve the shortest path problem for all pairs of nodes.





# SHORT: A Brute Force Approach

How would we design a *brute force* approach?

We would generate all possible paths, always “remembering” the shortest one we have seen so far.

Usually, one advantage of a brute force method is that it doesn't require a lot of thought. However, do you have any idea how to generate all possible paths from **A** to **B**?

In a shortest path, no edge is used twice, no node visited twice.

**QUIZ:** What assumption guarantees these facts?

A shortest path will never have more than  **$N-1$**  edges.

**QUIZ:** Explain why, please!



## SHORT: Generating Paths from one node to another

“Depth first search” can generate all paths from one node to another. Let’s get all paths from **A** to **B** in the weighted graph example.

0: Start with a list  $\mathcal{L}$  of just one partial path, namely **A**.

1: If there are no partial paths left in  $\mathcal{L}$ , we are done.

2: Remove one partial path from  $\mathcal{L}$ , and create new partial paths by taking one more step to a new node.

2a: If no steps were possible, go back to step 1.

2b: If the new step took you to node **B**, then this is a complete path, so print it.

2c: Add all remaining partial paths back to  $\mathcal{L}$ , and go to step 1.



# SHORT: Generating Paths from one node to another

A? -> AF? -> AFG? -> AFGE? -> AFGEH? -> AFGEHJ? -> ...  
AB AFE? AFGH? AFGEI? AFGEHI?  
AFGED?

... --> AFGEGI? -> AFGEHJID? -> AFGEHJIDB  
AFGEHJIC? AFGEHJIDC? -> AFGEHJIDCB

At this point, we've generated three paths from **A** to **B**, but we still have six more partial paths to work through.

The point is that this is a systematic approach to generating all possible paths - in other words, a computer can do it.

**QUIZ:** Prove the 1-step path from **A** to **B** is the shortest one!



# SHORT: Algorithm Classification

The "depth first" search that we employed for the shortest path problem is an example of the **decrease and conquer** technique.

In order to find the shortest path from A to B, we needed to find all paths from A to B. We decrease the problem size by looking for all paths from a neighbor of A, to B. On the second step, we look for all paths from a neighbor of a neighbor of A, to B. On each step, as we add a node to our path, we are reducing the problem size, that is, the number of remaining nodes we must consider.



# Graph Algorithms

- Overview
- Representing a Graph
- Connections
- The Connection Algorithm in MATLAB
- Components
- Adjacency
- Depth-First Search
- Weighted Graphs
- The Shortest Path
- **Dijkstra's Shortest Path Algorithm**
- The Minimum Spanning Tree
- Permutations
- The Traveling Salesman
- Projects



# DIJKSTRA: An Algorithm For the Shortest Path

Edsger Dijkstra is one of the founders of computer science. We will examine his solution to the shortest path problem.

The idea is to get the answer in a way that is simple to program, efficient to run, and provably correct.

Our input is a pair of nodes, **A** and **B**, and an edge length matrix **Length**, which will be  $\infty$  for nodes with no direct connection.

Our result is the value **Dist(B)**, the shortest distance from **A** to **B**, or, if there is no path, the value  $\infty$ .



# DIJKSTRA: The Algorithm

- 1 For all nodes **C**, initialize:  
 $\text{Dist}(\mathbf{C}) = \infty$ ,  
 $\text{CONNECT}(\mathbf{C}) = 0$ .  
Then set  $\mathbf{P} = \mathbf{A}$ ,  
 $\text{Dist}(\mathbf{A}) = 0$ .
- 2 Connect node **P**, that is,  $\text{CONNECT}(\mathbf{P}) = 1$ .
- 3 For every unconnected node **C** with a finite edge (C,P):  
 $\text{Dist}(\mathbf{C}) = \min ( \text{Dist}(\mathbf{C}), \text{Dist}(\mathbf{P}) + \text{Length}(\mathbf{C},\mathbf{P}) )$
- 4 If there are no more unconnected nodes with a finite value of  $\text{Dist}()$ , we are done. There is no path to **B**.
- 5 Set **P** to the “closest” unconnected node, that is, the node with minimum value of  $\text{Dist}$ .
- 6 If **P** is **B**, we are done, and  $\text{Dist}(\mathbf{B})$  contains the shortest distance from **A** to **B**.
- 7 Go to step 2.



# DIJKSTRA: Shortest Paths to ALL Other Nodes

Suppose that we want the shortest paths from **A** to *every* node.

We can modify Dijkstra's algorithm to solve this problem as well.

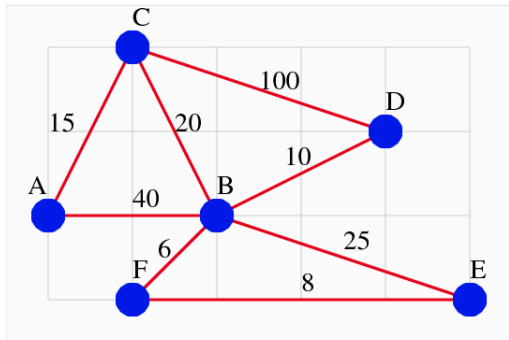
Replace steps 4, 5 and 6 of the previous version by:

- 4 If there are no more unconnected nodes with a finite value of **Dist()**, we are done.
- 5 Set **P** to the "closest" unconnected node, that is, the node with minimum value of **Dist**.
- 6 Go to step 2.





# DIJKSTRA: The DIJKSTRA Example Graph



# DIJKSTRA: A Worked Example

Step	P	-----Connect-----						-----Dist-----					
		A	B	C	D	E	F	A	B	C	D	E	F
1	A	1	-	-	-	-	-	0	40	15	∞	∞	∞
2	C	1	-	1	-	-	-	0	35	15	115	∞	∞
3	B	1	1	1	-	-	-	0	35	15	45	65	41
4	F	1	1	1	-	-	1	0	35	15	45	49	41
5	D	1	1	1	1	-	1	0	35	15	45	49	41
6	E	1	1	1	1	1	1	0	35	15	45	49	41



# DIJKSTRA: Algorithm Classification

Dijkstra's algorithm for the shortest path problem can be classified as a **greedy algorithm**.

At each step, we simply take add the edge that looks best (shortest length) without any regard to long term strategy. The only checks we make are that the new edge connects an "old" node to a "new" one, and has the lowest value of **D** among all such edges.

The advantage of a greedy algorithm is that the decision process is simple and efficient.



# Graph Algorithms

- Overview
- Representing a Graph
- Connections
- The Connection Algorithm in MATLAB
- Components
- Adjacency
- Depth-First Search
- Weighted Graphs
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- **The Minimum Spanning Tree**
- Permutations
- The Traveling Salesman
- Projects



# SPAN: What is a Tree?

A **tree** is:

- a simple graph;
- and connected;
- and if any edge is removed from the tree, the remaining graph becomes disconnected.

A **cycle** in a graph is a path (of distinct edges) which begins and ends at the same node.

**QUIZ:** Prove that in a tree, there are no cycles.



# SPAN: What is a Spanning Tree

A **spanning tree** of a connected graph  $\mathcal{G}$  is a tree which is formed by deleting all edges of  $\mathcal{G}$  that are not needed to keep the remaining graph connected.

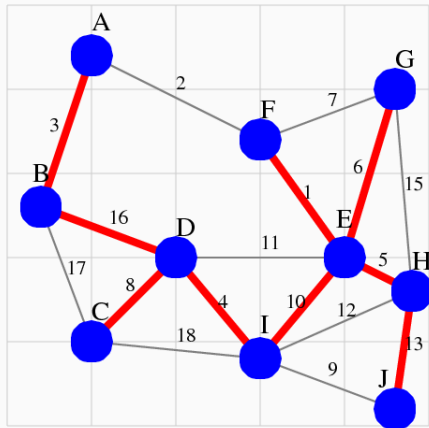
Usually, there are many different spanning trees for a given graph.

However, if there are **N** nodes in the graph, a spanning tree will *always* have exactly **N-1** edges.



# SPAN: A Spanning Tree for the MST Graph

The length of this spanning tree is **66 units**.



# SPAN: The Minimal Spanning Tree

The **minimal spanning tree** of a connected *weighted* graph  $\mathcal{W}$  is the spanning tree with the property that the the total edge length is the smallest possible value.

Given a weighted graph  $\mathcal{W}$ , a minimal spanning tree has 3 tasks:

- 1 it must eliminate all but **N**-1 edges;
- 2 the remaining edges must keep all the nodes connected;
- 3 the total of the lengths of these edges must be the lowest possible value.





# SPAN: Kruskal's Algorithm

One approach to the minimal spanning tree problem is known as **Kruskal's algorithm**. It is very easy to describe.

Our tree will be described by  $\mathcal{T}$ , a list of some of the edges of  $\mathcal{W}$ . (We actually know that exactly **N**-1 edges will be needed.)

At each step, we add one new edge to  $\mathcal{T}$ . The edge we choose is the shortest unused edge that does not create a cycle. (Remember what a cycle is?)



# SPAN: Kruskal's Algorithm Worked Out

	AB	AF	BC	BD	CD	CI	DE	DI	EF	EG	EH	EI	FG	GH	HI	HJ	IJ	Sum
	3	2	17	16	8	18	11	4	1	6	5	10	7	15	12	13	9	
1									EF									1
2		AF							EF									3
3	AB	AF							EF									6
4	AB	AF						DI	EF									10
5	AB	AF						DI	EF		EH							15
6	AB	AF						DI	EF	EG	EH							21
7	AB	AF			CD			DI	EF	EG	EH		xx					29
8	AB	AF			CD			DI	EF	EG	EH						IJ	38
9	AB	AF			CD			DI	EF	EG	EH	EI					IJ	48





# SPAN: Pseudocode for Kruskal's Algorithm

Sort the list of edges  $\mathbf{E}$  by length, smallest first.  
There are  $\mathbf{NE}$  edges.

$\mathbf{T} = \{\}$ ;  $\mathbf{NT} = 0$ ;

for  $i = 1$  to  $\mathbf{NE}$

    if ( edge  $\mathbf{E}(i)$  does not add a cycle to  $\mathbf{T}$  )

$\mathbf{T} = \mathbf{T} + \mathbf{E}(i)$ ;

$\mathbf{NT} = \mathbf{NT} + 1$ ;

        if (  $\mathbf{NT} == \mathbf{N} - 1$  )

            done;

        end

    end

end



## SPAN: Checking for a Cycle

How can we check whether edge  $\mathbf{E(i)}$  adds a cycle to  $\mathcal{T}$ ?

$\mathbf{E(i)}$  is a pair of nodes  $\{\mathbf{A,B}\}$ .  $\mathbf{E(i)}$  adds a cycle to  $\mathcal{T}$  if, and only if, both nodes  $\mathbf{A}$  and  $\mathbf{B}$  already occur somewhere in the set  $\mathcal{T}$ .

In our worked example, we considered adding edge  $\{\mathbf{F,G}\}$ , but rejected it because at that point the list of edges was  $\{ \{\mathbf{A,B}\}, \{\mathbf{A,F}\}, \{\mathbf{D,I}\}, \{\mathbf{E,F}\}, \{\mathbf{E,G}\}, \{\mathbf{E,H}\} \}$ .

That is, both nodes  $\mathbf{F}$  and  $\mathbf{G}$  occurred in the list.

So we moved on to the next longer edge,  $\{\mathbf{C,D}\}$ .

**QUIZ:** why was edge  $\{\mathbf{C,D}\}$  acceptable?



Kruskal's algorithm for the minimal spanning tree problem can be classified as a **greedy algorithm**.

At each step, we simply take add the edge that looks best (shortest length) without any regard to long term strategy. The only check we make is that the new edge can't add a loop.



# Graph Algorithms

- Overview
- Representing a Graph
- Connections
- The Connection Algorithm in MATLAB
- Components
- Adjacency
- Depth-First Search
- Weighted Graphs
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- **Permutations**
- The Traveling Salesman
- Projects



# PERM: Introduction

Before we look at the traveling salesman problem, we must first learn a little about **combinatorial computation**.

Combinatorial computation studies objects which we can think of as arrangements of items.

For each class of objects, we seek algorithms for a few simple tasks.

Counting the number of paths from **A** to **B** on a graph is a kind of combinatorial problem.

Other examples include the number of ways you can fold a map, or how many ways you can rearrange the letters in your name.

The traveling salesman problem requires us to choose the best itinerary for visiting **N** cities, which is a **permutation**.





# PERM: Combinatorial Objects

The objects of combinatorial computations include:

- subsets of a set of  $\mathbf{N}$  items;
- subsets of size  $\mathbf{K}$  of a set of  $\mathbf{N}$  items;
- permutations of  $\mathbf{N}$  items (ordered sequence);
- combinations of  $\mathbf{K}$  out of  $\mathbf{N}$  items (unordered set);
- compositions of an integer  $\mathbf{N}$  into  $\mathbf{K}$  parts;
- partitions of a set of  $\mathbf{N}$  items into  $\mathbf{K}$  subsets.



# PERM: Combinatorial Tasks

Once we have chosen a set of objects to study, they can usually be numbered (or “*ranked*”) and listed.

Now combinatorial algorithms are available for these tasks:

- **enumerate**: how many objects in the list?
- **generate**: produce the next object in the list;
- **sample**: generate one object at random;
- **rank**: given an object, find its rank;
- **unrank**: given a rank, find the object.



# PERM: Combinatorial Tasks

Our combinatorial objects will be *permutations*.

A permutation of the numbers 1 through  $\mathbf{N}$  (also called a permutation of *order*  $\mathbf{N}$ ), is an ordered list in which each number appears exactly once.

There are 6 permutations of order 3, and an ordered list of them is:

1: 1,2,3

2: 1,3,2

3: 2,1,3

4: 2,3,1

5: 3,1,2

6: 3,2,1

There are 24 permutations of order 4,  
and  $\mathbf{N!}$  permutations of order  $\mathbf{N}$ .



# PERM: Combinatorial Tasks for TSP

For the traveling salesman problem, each permutation is an *itinerary*, that is, the order in which we visit the cities.

For small problems, we can check every possible itinerary;  
for large problems, we'll improve a randomly chosen itinerary.

Therefore, our combinatorial tasks will be:

- generate all permutations one at a time;
  - this will allow us to check every possible itinerary;
- select a permutation at random;
  - this will give us a random starting itinerary.



# PERM: Random Permutation

We start with the task of choosing a permutation at random.

The algorithm for a random permutation of order  $N$  is:

```
Initialize  $P$  to [ 1, 2, ...,  $N$ ];
```

```
for  $I = N$  down to 2
```

```
    Choose a random integer  $J$  between 1 and  $I$ ;
```

```
    Swap  $P(I)$  and  $P(J)$ ;
```

```
end
```

Oops! Now we have to work out how to pick a random integer!!



## PERM: Picking a Random Integer

We use the facts that **rand** ( ) is strictly between 0 and 1, and that **ceil** rounds up:

```
function j = randint ( i )  
    j = ceil ( i * rand ( ) );  
    return  
end
```

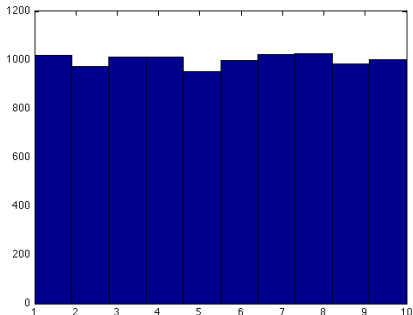
We'll check our algorithm with the **hist** command:

```
n = 10000;  
t = zeros(n,1);  
for k = 1 : n  
    t(i) = randint ( 10 );  
end  
hist ( t )
```



# PERM: Histogram for RANDINT

Our histogram looks OK, so we probably know how to select a random integer between 1 and  $\mathbf{I}$ ...  
so we can do a random permutation!



# PERM: The Next Permutation

The algorithm for the next permutation is a little too complicated to describe. However, you should be able to guess a little bit about how it works by looking at the beginning of the sequence for permutations of order 4:

```
1234 <-- 4 single,          34 increases.
1243 <-- 43 decreases, but  243 increases.
1324 <-- 4 single,         but  24 increases.
1342 <-- 42 decreases, but  342 increases.
1423 <-- 3 single,         but  23 increases.
1432 <-- 432 decreases, but 1432 increases
2134 <-- 4 single,         but  34 increases.
2143
2314
...
4312 <-- 2 single,         but  12 increases
4321 <-- 4321 decreases, nothing left.
```





# PERM: The Next Permutation

I hate to give you an algorithm without at least some idea of what is going on.

Can you match these ideas to the sequence on the previous slide?

- 1 If  $\mathbf{P}$  is all zeros, return with  $\mathbf{P}=\mathbf{1:N}$ ;
- 2 Seek  $\mathbf{I}$ , the highest index for which  $\mathbf{P}(\mathbf{I}) < \mathbf{P}(\mathbf{I}+1)$ ;
- 3 If no such  $\mathbf{I}$ , return with  $\mathbf{P}=[]$ ;
- 4 Look for the highest  $\mathbf{J}$  index so that  $\mathbf{P}(\mathbf{I}) < \mathbf{P}(\mathbf{J})$ ;
- 5 Interchange  $\mathbf{P}(\mathbf{I})$  and  $\mathbf{P}(\mathbf{J})$ ;
- 6 Reverse the order of  $\mathbf{P}(\mathbf{I}+1:\mathbf{N})$  and return.



## PERM: The PERM\_NEXT Function

The implementation of **perm\_next** is

```
function p = perm_next ( p )
```

To print every permutation, you might write this:

```
p = zeros(5,1);           <-- Initialize to [0,0,0,0,0].
while ( 1 )
    p = perm_next ( p );
    if ( isempty ( p ) ) <-- [] is returned at end.
        break
    end
    fprintf ( 1, 'P = %d %d %d %d %d\n', p(1:5) );
end
```

For small TSP problems, we would use a loop like this to find the cost of every possible itinerary.



- Overview
- Representing a Graph
- Connections
- The Connection Algorithm in MATLAB
- Components
- Adjacency
- Depth-First Search
- Weighted Graphs
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- Permutations
- **The Traveling Salesman**
- Projects



The traveling salesman problem involves a salesman who has been given a list of cities to visit. The salesman must choose a starting city, visit every city on the list, and then return to the starting city.

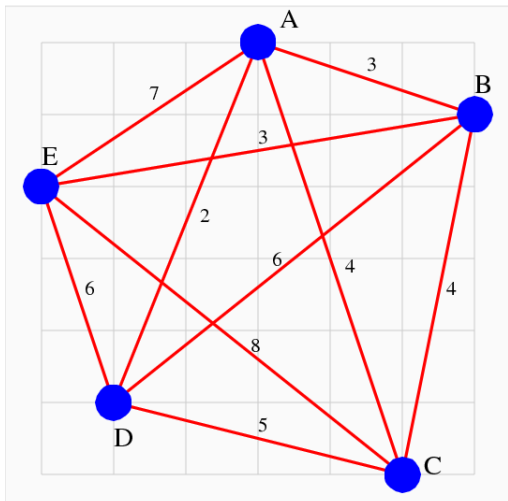
If the salesman travels by plane, then we assume that there is a direct connection between any pair of cities.

If the salesman is driving, then probably only a few cities are directly connected to the current city. We can make this problem like the previous one by simply assigned the value  $\infty$  as the distance to each unreachable city.

The salesman has been given an expense account, and wants to minimize the cost of the trip. And this means finding the round trip that visits all the cities, but involves the least total distance traveled.



# TSP: A Simple Traveling Salesman Graph



# TSP: A Brute Force Approach

A “brute force” algorithm is one that is guaranteed to work, but which does not try to find any short cuts, or use any insight into the problem.

A brute force solution of the TSP would be as follows:

- Given: a set of  $N$  cities and their pairwise distances.
- Initialize  $R^* = []$ ,  $D^* = \infty$ ;
- While (more routes available),
  - Generate the “next” route  $R$ ;
  - Measure  $D$ , the length of  $R$ ;
  - If  $D < D^*$ , update  $D^*$  and  $R^*$ .
- $R^*$  is the best possible route, and  $D^*$  is its length.



## TSP: Use Brute Force on Example

There are  $5! = 120$  routes, but they come in sets of 10.  
For instance:

$$\begin{aligned} ABCDE &= BCDEA = CDEAB = DEABC = EABCD = \\ AEDCB &= BAEDC = CBAED = DCBAE = EDCBA = 25. \end{aligned}$$

However, if we just use brute force, checking all 120 possibilities, we come up with a best route, which is  $(1,3,2,5,4)$  as a permutation:

$$ACBED = 4 + 4 + 3 + 6 + 2 = 19.$$

(Remember that we include the cost of the return from **D** to **A**.)



## TSP: Brute Force Solver

```
p = zeros ( 5, 1 ); dstar = Inf; pstar = [];  
while ( 1 )  
    p = next_perm ( p );  
    if ( isempty ( p ) )  
        break  
    end  
    d = 0.0;  
    i = n;                                <--Explain!  
    for j = 1 : n  
        d = d + a(p(i),p(j)); <--Explain!  
        i = j;  
    end  
    if ( d < dstar )  
        dstar = d; pstar = p;  
    end  
end  
end
```





# TSP: A More Realistic Problem

How many itineraries are possible with 48 cities?



The brute force approach cannot solve TSP problems of any interesting size. And yet, solutions, or good approximate solutions, have been found for problems with as many as 85,000 “cities”.

Sometimes we can try to come close to a good solution to a hard combinatorial problem because there is a **heuristic algorithm**.

A heuristic for an optimization problem is a method of making a good guess, or improving a random guess.



Our heuristic for the TSP repeats the following process:

*Pick a starting point at random, then build an itinerary by moving from your current location to the nearest unvisited city, again and again, until you must return home.*

After generating many itineraries, choose the shortest one.

**Quiz:** If we are prepared to compute up to 100 itineraries, what is different about problems of 50, 150, and 10,000 nodes?



# TSP: The Nearest City Heuristic

Let's use our heuristic on our simple TSP problem:

Start:	Length
-----	-----
A-(2)-D-(5)-C-(4)-B-(3)-E-(7)-A	21
B-(3)-A-(2)-D-(5)-C-(8)-E-(3)-B	21
C-(4)-A-(2)-D-(6)-B-(3)-E-(8)-C	23
D-(2)-A-(3)-B-(3)-E-(8)-C-(5)-D	21
E-(3)-B-(3)-A-(2)-D-(5)-C-(8)-E	21

We never found the optimal route of 19, but we did OK.



## TSP: Improving a Good Route

Another technique allows us to make a good route better.

Suppose that the intercity distances are “real” distances, so that they satisfy the **triangle inequality**:

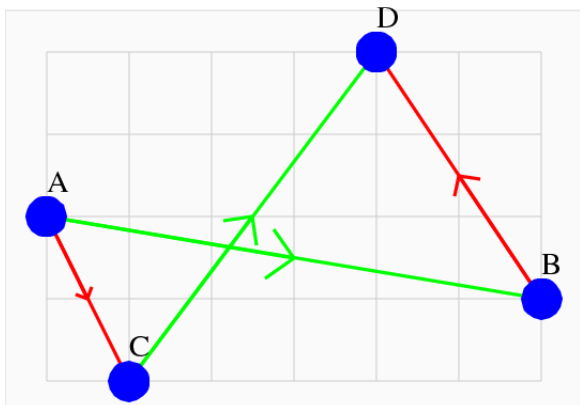
$$\text{distance (A to C)} \leq \text{distance (A to B)} + \text{distance (B to C)}$$

Then if our good route crosses itself, that is, the path from A to B and the path from C to D cross over each other, then we can make a better route by changing the itinerary so that we go from A to C and from B to D.



# TSP: Replace Green Lines by Red Ones

**Quiz:** Prove that the new itinerary must be shorter!



The TSP problem is one of very great interest for industrial design, transportation authorities, communications systems, and, of course, traveling salesmen with limited budgets!

Two very useful websites about the traveling salesman problem are:

<http://www.tsp.gatech.edu/>

<http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>



# Graph Algorithms

- Overview
- Representing a Graph
- Connections
- The Connection Algorithm in MATLAB
- Components
- Adjacency
- Depth-First Search
- Weighted Graphs
- The Shortest Path
- Dijkstra's Shortest Path Algorithm
- The Minimum Spanning Tree
- Permutations
- The Traveling Salesman
- **Projects**





# PROJECTS: Suggestions

Some graph topics suitable for your final project:

- the **Euler** all-edge cycle problem;
- the **Hamilton** all-node cycle problem;
- the **traveling salesman** problem;
- the **maze** solving problem;
- the **graph coloring** problem;
- counting all possible trees with the **Prüfer code**.

Look these problems up and see if you'd be interested in working on one of them.

