Intro Math Problem Solving December 7

New Versions of Adjacency
The Traveling Salesman Problem
Example V (5 Cities)
Brute Force Algorithm & Permutations
48 State Capital Example
Random Sampling Algorithm
Nearest Neighbor Method

Reference

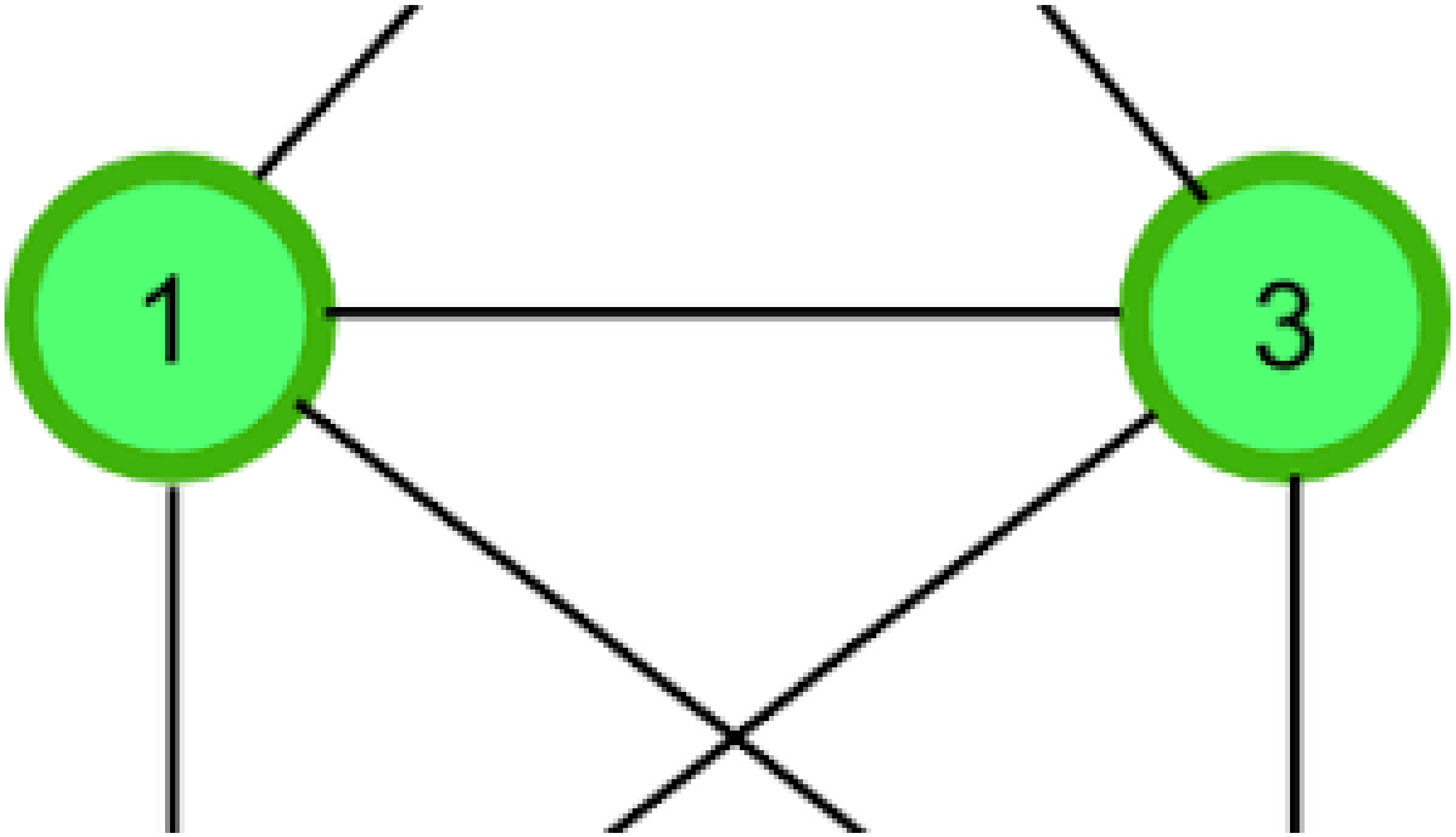
Chapter 15 of our textbook covers Optimization; section 1, “Shortest Route: The Combinatoric Explosion”, considers the traveling salesman problem.

The TSPLIB website is full of information on the traveling salesman problem, including the history, maps, programs, and references.

https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/

.

New Versions of Adjacency Matrix



New Versions of Adjacency

The classic adjacency matrix Adj(I,J) for a graph records a 1 if nodes I and J are neighbors, and 0 otherwise.

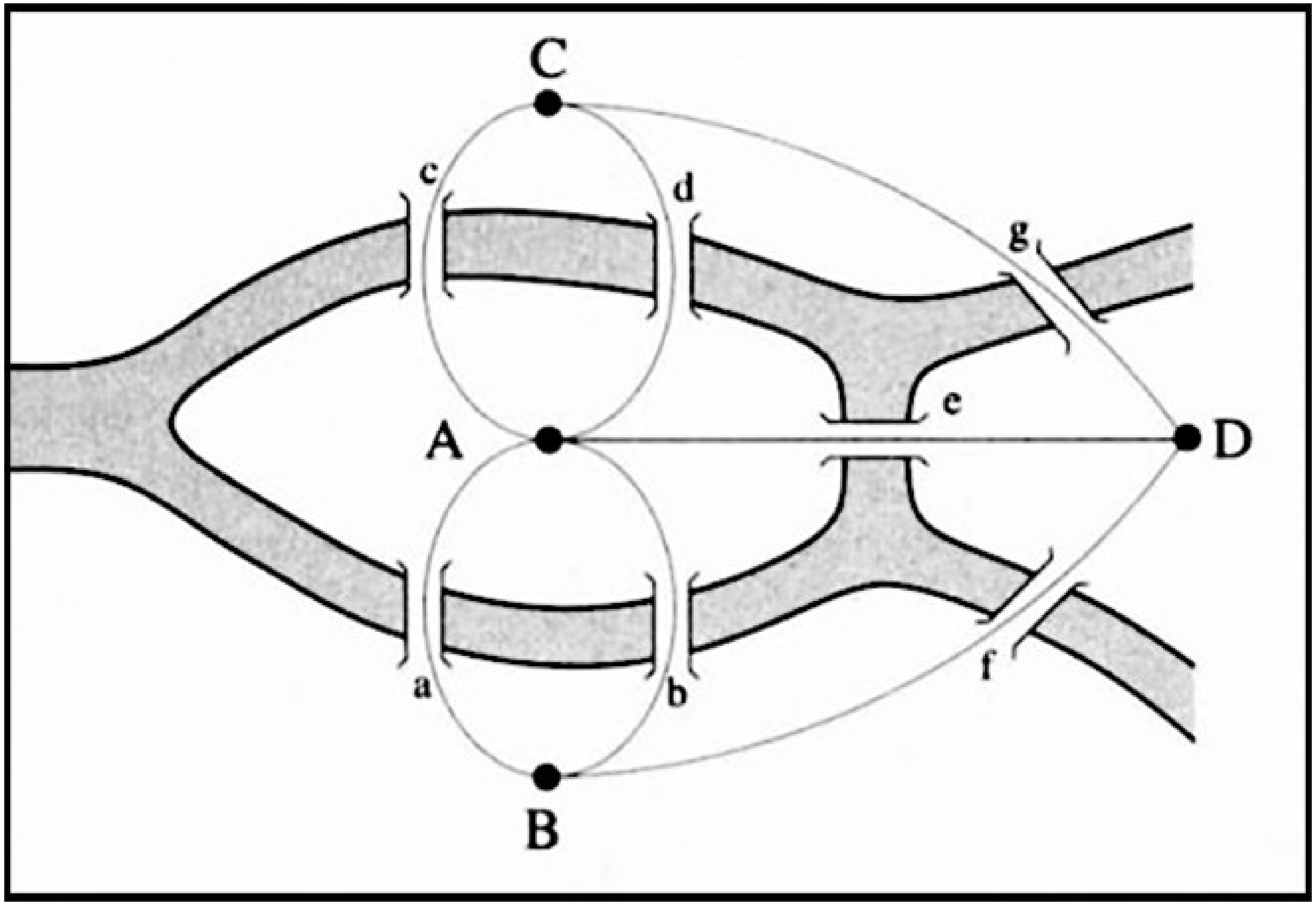
In MATLAB, we can also think of 1 and 0 values are representing TRUE (nodes are connected) and FALSE (they are not).

Graphs are so useful that they have been adapted to more complicated problems.

If we are willing to modify our definition of the adjacency matrix, then we can also describe these more complicated graphs.

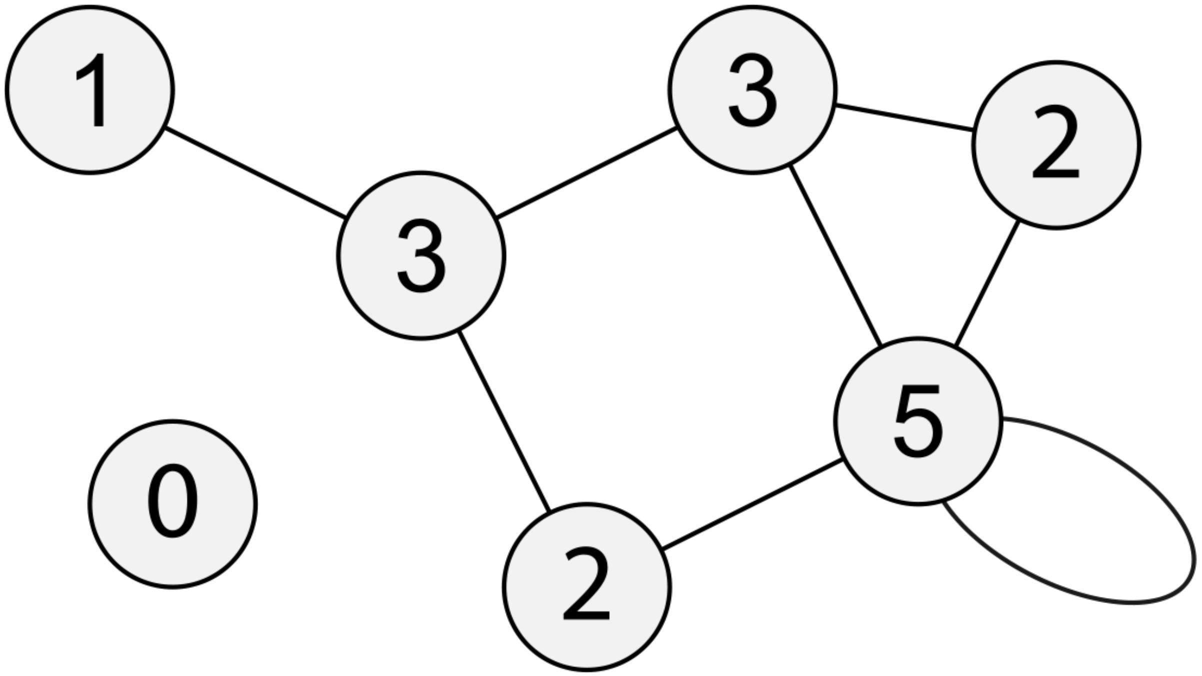
Multiple Edges between Nodes

We can handle this by letting Adj(I,J) count the edges. This is sometimes called a "multigraph". Here, some entries of Adj will be 2.



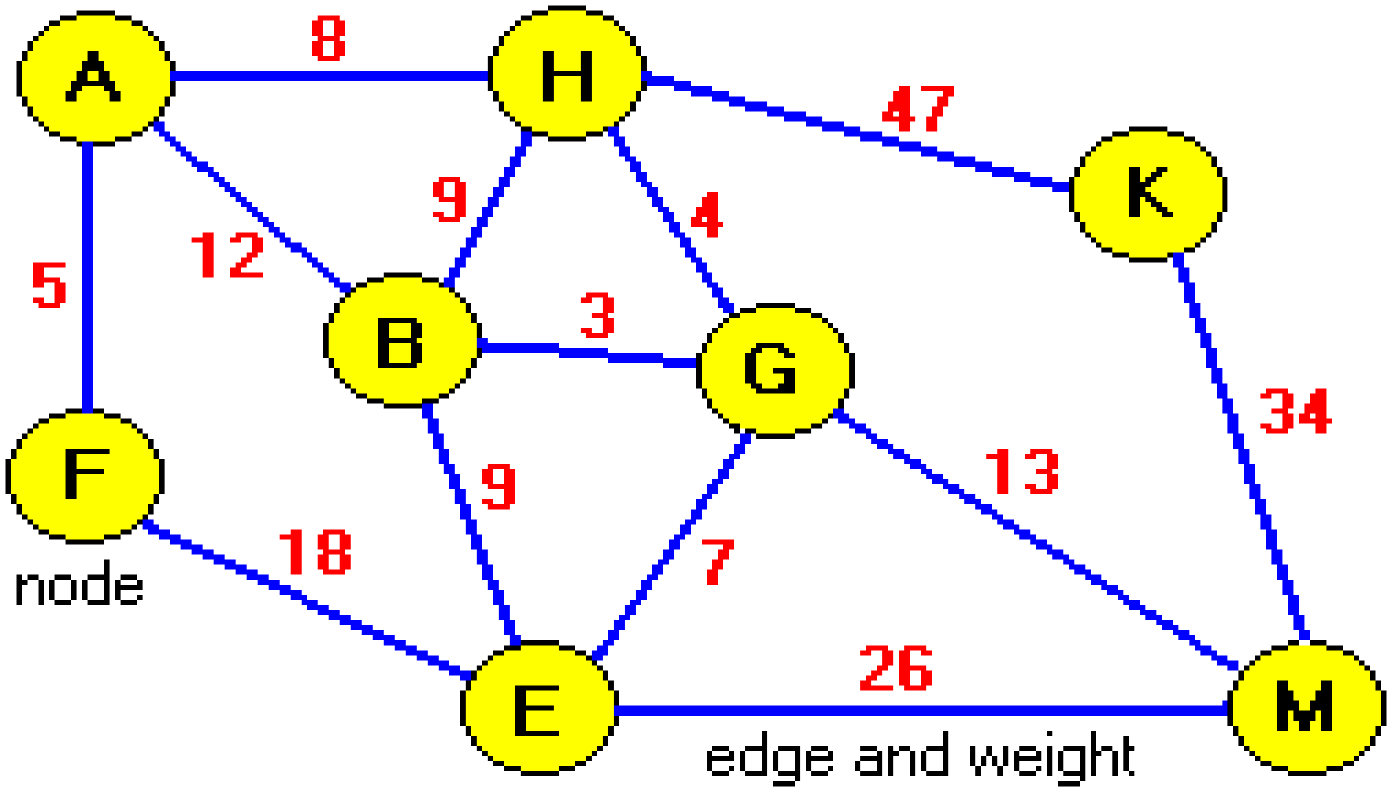
A node can be its own neighbor

Adj(i,i) = 1 is now possible. This is sometimes called a graph with “loops” or “self-loops”.



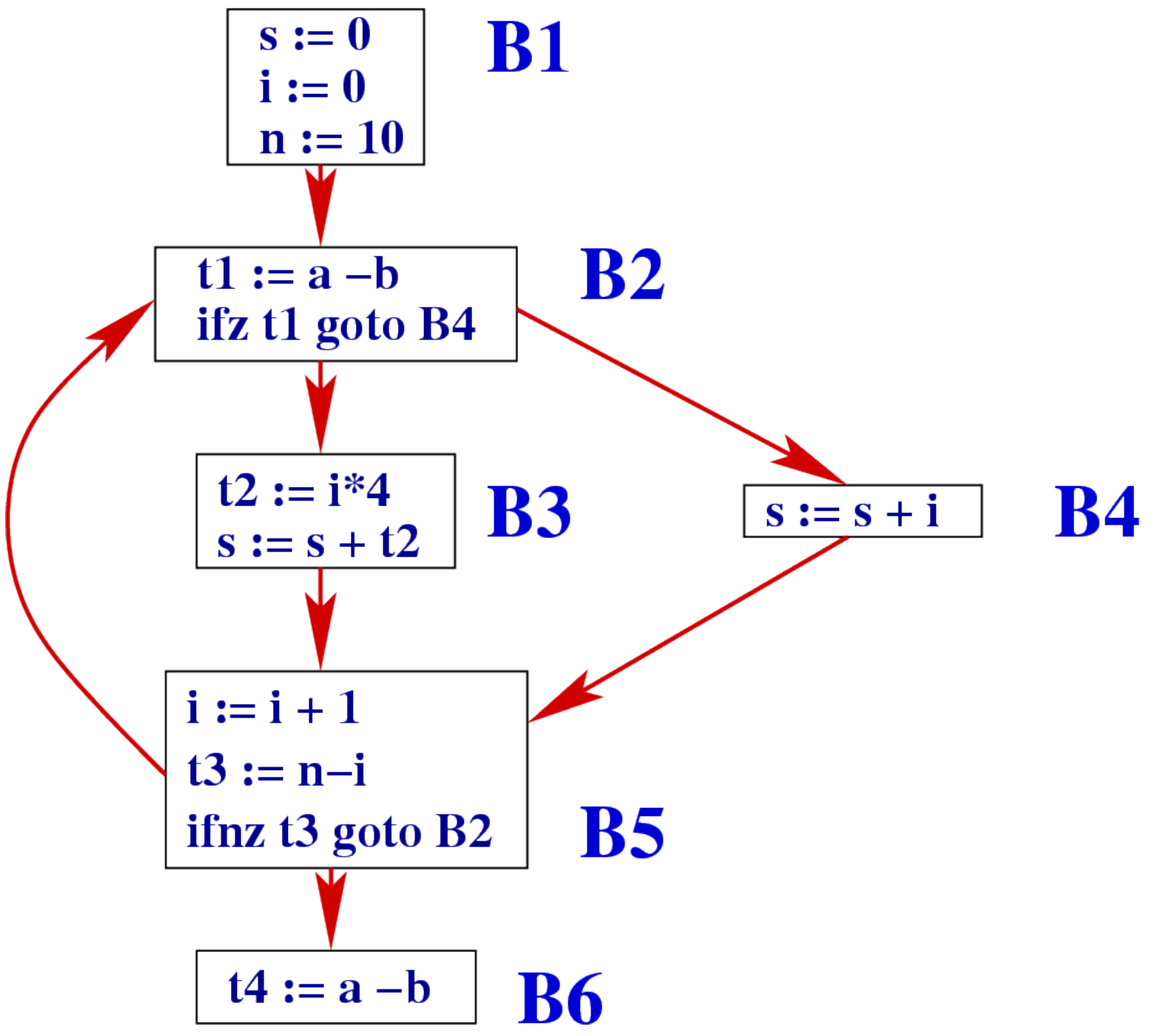
Edges can have length or weight

Adj(i,j) = 7.5 means the “road” from node I to node J is 7.5 miles long. This allows us to compute city-to-city distances. This is called a weighted graph.



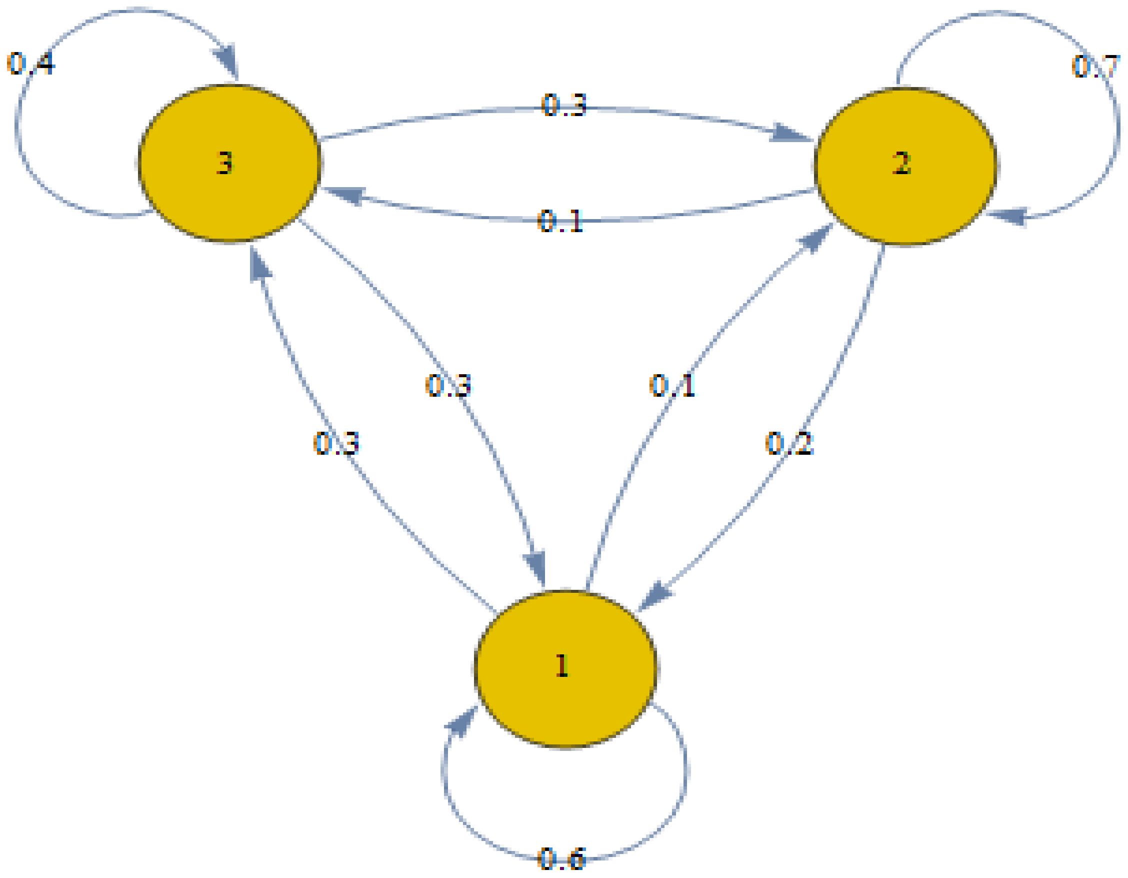
Edges can have direction

Adj(i,j)=1 means there is an edge FROM I TO J, but it does not mean there is an edge in the other direction. This is a directed graph or digraph.

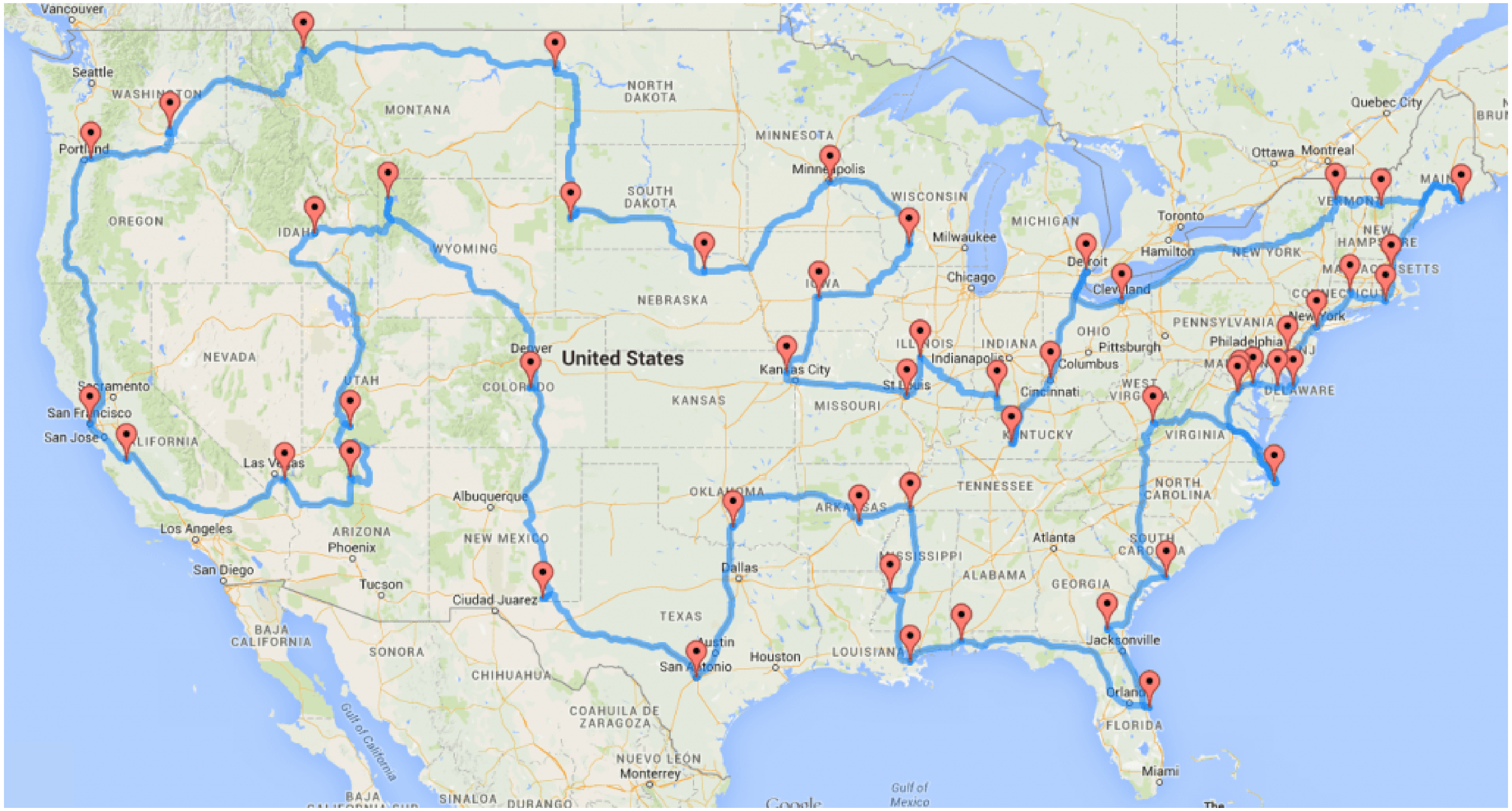


Edges with direction and weight

Our transition problems, studier earlier, can be represented by a graph with loops, direction, and weight (probability), a weighted digraph.



The Traveling Salesman Problem



TSP

The traveling salesman problem, known as “TSP” is a classic example in operations research and computer science.

A traveler must visit every city on a list and return home.

The "adjacency matrix" is now a table of city-to-city distances. Dist(I,J) is the distance between cities I and J.

If travel between cities I and J is not possible, Dist(I,J) is set to "Infinity". In MATLAB, this is the special value Inf.

The traveler wants a round trip that lists the cities in the order they are visited.

The traveler seeks the round trip of *shortest total distance*.

Seeking the Best Route

The traveling salesman problem is a simplified version of an issue that arises in many ways:

* scheduling a machine to drill holes in a circuit board;

* DNA sequencing, connecting local genome maps;

* minimize the movements of a telescope that must examine a set of stars;

* planning the route of a school bus;

* shortest cable network linking all offices;

Abstract Version of Problem

We assume there are N cities, identified by their index 1 through N.

We will assume we are given an NxN distance matrix Dist(*,*).

* all entries of Dist are nonnegative;

* D(I,J) = Infinity is allowed, indicating direct travel not possible.

* for every city I, Dist(I,I)=0,

* for every pair of cities I and J, Dist(I,J)=Dist(J,I).

A round trip is a list T of the N cities in any order.

The length of the trip is the sum of Dist (T(K), T(K+1)), for K = 1 to N.

We assume the trip finishes back at the first city in the list, so in the formula for the trip length, when K is N, T(N+1) will actually be T(1).

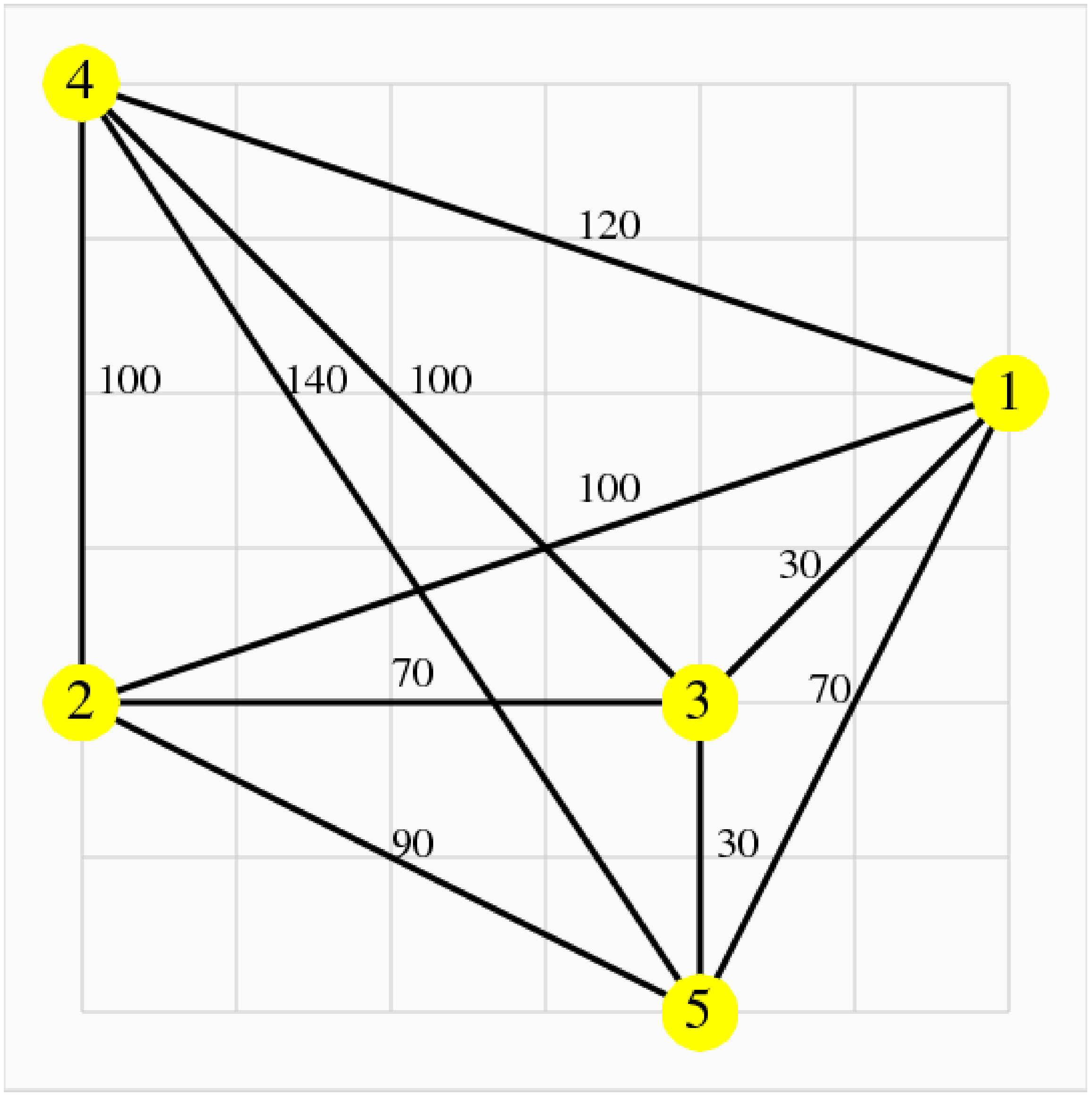
How Hard is this Problem?

1) There is at least one satisfactory solution to this problem, because we can imagine a list of all possible trips, and there must be a shortest one. It's of course logically possible for more than one trip to have the same shortest length; the point is, there's at least one solution!

2) Every possible ordering of the N cities represents a round trip. There are N! such orderings, and when N=10, this is more than 3 million possibilities. If we were visiting all 50 state capitals, the number of possibilities is unbelievable.

3) Although the problem involves geometry, it is not clear that there are any geometric ideas that will help us.

Example V: Five Cities



Distance Table

	1	2	3	4	5
1	0	100	30	120	70
2	100	0	70	100	90
3	30	70	0	120	50
4	120	100	120	0	140
5	70	90	50	140	0

v_distance.m

```
function dist = v_dist ( )

  dist = [   0, 100,  30, 120,  70;
           100,   0,  70, 100,  90;
            30,  70,   0, 120,  50;
           120, 100, 120,   0, 140;
            70,  90,  50, 140,   0 ];

  return
end
```

trip_distance.m

```
function total = trip_distance ( dist, t )

%% TRIP_DISTANCE computes the total length of a round trip.
%
  n = length ( t );
%
%  Copy first city to end of list to make a round trip.
%
  t = [ t, t(1) ];

  total = 0.0;
  for i = 1 : n
    total = total + dist ( t(i), t(i+1) );
  end

  return
end
```

Sample Round Trips

dist = v_distance ( );

1: Cities in order:

T = [1,2,3,4,5]

length = trip_distance (dist, t) = 500

2: Go around outside first:

T = [1,4,5,2,3]

length = trip_distance (dist, t) = 450

3: Zigzag up

T = [1,5,3,2,4]

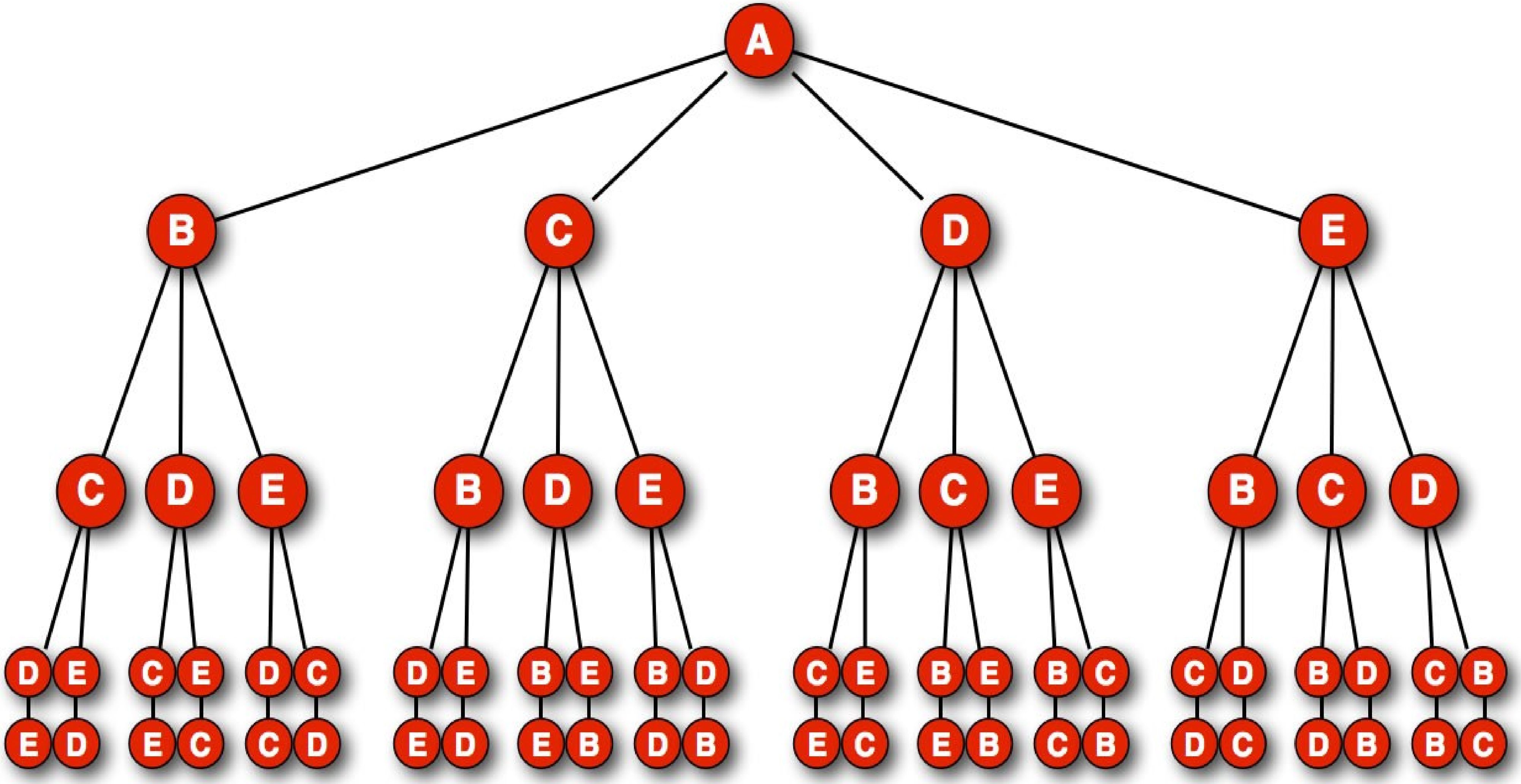
length = trip_distance (dist, t) = 410

4: Random

T = randperm (5) = [3, 5, 1, 2, 4]

length = trip_distance (dist, t) = 440

Brute Force & Permutations



Our Problem is Simple

Our problem is neat and simple in several ways.

There are only "so many" possible trips to consider. That means we could imagine creating a list of all these trips.

For each trip, the distance matrix allows us to compute the total length, so we can imagine that our list includes both the itinerary and the length.

We seek the shortest trip. We could imagine sorting our list of trips and lengths, so that the shortest trip appears first on our list:

length: city#1, city#2, ..., city#n

We have considered all possibilities, so our result is guaranteed to be the best.

A solution approach like this, which doesn't bother with mathematical elegance, but simply marches through the data sequentially, is called a brute force method.

Permutations

Our brute force method must create a list of all possible trips, T.

Each trip T is a permutation of the integers 1 through N, a list that includes each integer exactly once.

To list all possible trips, we must list all possible permutations of the N integers.

To do this in a systematic fashion, we would like to think of the permutations as having some natural ordering.

A natural choice is lexicographic ordering.

Lexicographic Ordering

Here are the 24 permutations of 1 through 4, in lexicographic order:

```
# 1: 1, 2, 3, 4        #13: 3, 1, 2, 4
# 2: 1, 2, 4, 3        #14: 3, 1, 4, 2
# 3: 1, 3, 2, 4        #15: 3, 2, 1, 4
# 4: 1, 3, 4, 2        #16: 3, 2, 4, 1
# 5: 1, 4, 2, 3        #17: 3, 4, 1, 2
# 6: 1, 4, 3, 2        #18: 3, 4, 2, 1
# 7: 2, 1, 3, 4        #19: 4, 1, 2, 3
# 8: 2, 1, 4, 3        #20: 4, 1, 3, 2
# 9: 2, 3, 1, 4        #21: 4, 2, 1, 3
#10: 2, 3, 4, 1        #22: 4, 2, 3, 1
#11: 2, 4, 1, 3        #23: 4, 3, 1, 2
#12: 2, 4, 3, 1        #24: 4, 3, 2, 1
```

For permutations, lexicographic order looks somewhat like numerical order, if we ignored the commas, so "1234" is first, than "1243", and so on, up to "4321".

Generating Permutations

To make our list of possible trips, we need to generate all the permutations of N integers, one at a time, in order, without missing any.

It's obvious that there is a mathematical pattern going on here, but it's not obvious, right away, how to follow it.

Permutations are an object of study in the mathematical fields of combinatorics and of abstract algebra.

There are many algorithms for the generation problem.

Algorithm: nextperm

Start with 1, 2, 3, ..., n

1) Starting with last entry, p(n), select the longest string of decreasing digits, indices i through n. (If i = 1, we are done.)

2) Find the smallest value p(j) (i <= j <= n) that is bigger than p(i-1).

3) Swap p(j) and p(i-1).

4) Sort entries p(i) through p(n) in increasing order.

Demonstrate nextperm

0) Our current permutation is:

p = 7 9 10 5 8 3 4 6 2 1

We need to compute the “next” permutation.

1) Longest string of decreasing digits, starting at the end, is "6, 2, 1" in positions 8, 9, 10.

2) p(7)=4, and in the string "6, 2, 1", smallest value that is bigger than 4 is p(8)=6.

3) Swap p(7) and p(8):

p = 7 9 10 5 8 3 6 4 2 1

4) Sort p(8) through p(10):

p = 7 9 10 5 8 3 6 1 2 4

nextperm.m

```
function p = extperm ( p )

% Find non-increasing suffix
  n = length ( p )
  i = n;
  while ( i > 1 && p(i - 1) >= p(i) )
    i = i - 1;
  end
  if ( i <= 1 )
    p = [];
    return;
  end

% Find successor to pivot

  j = n;
  while ( p(j) <= p(i – 1) )
    j = j - 1;
  end
  temp = p(i - 1);
  p(i - 1) = p(j);
  p(j) = temp;

% Reverse suffix
  p(i : end) = p(end : -1 : i);
  return
end
```

Our Brute Force Algorithm

```
while ( true )
  initialize P=1:n, or get next P from nextperm
  compute length of this trip
  if this trip is shorter than best_length so far,
    save best_P, best_length
  end
end
```

tsp_brute.m

```
function [ tsp_distance, tsp_trip ] = tsp_brute ( distance )

  [ n, ~ ] = size ( distance );
  i = 0;

  while ( true )

    i = i + 1;

    if ( i == 1 )
      this_trip = 1 : n;
    else
      this_trip = nextperm ( this_trip );
      if ( isempty ( this_trip ) )
        break;
      end
    end

    this_distance = trip_distance ( distance, this_trip );

    if ( i == 1 || this_distance < tsp_distance )
      tsp_distance = this_distance;
      tsp_trip = this_trip;
    end

  end

  return
end
```

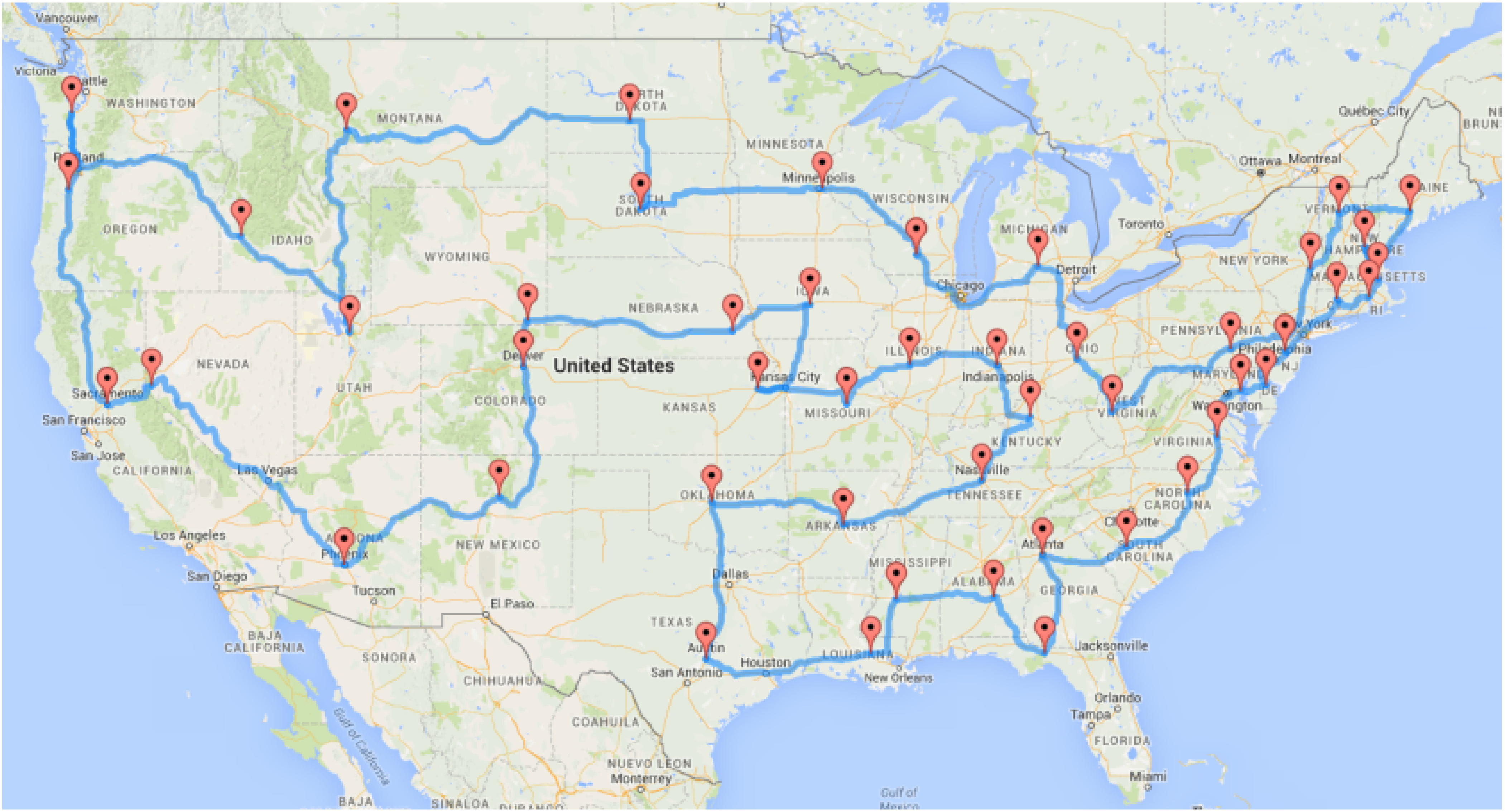
Demo

```
dist = v_dist ( );
[ tsp_distance, tsp_trip ] = tsp_brute ( dist )

tsp_distance =
   270

tsp_trip =
     1     3     5     2     4
```

48 US Capital Example



48 State Capital Example

To make a more realistic example, let’s consider the problem of a lobbyist who has to visit all 48 state capitals (excluding Hawaii and Alaska to keep the map simple.)

We will assume the lobbyist has a private jet, so that the distance between any two capitals is simply the flying distance. This way, we don’t have to deal with the highway map.

But how do we compute the flying distance between two cities?

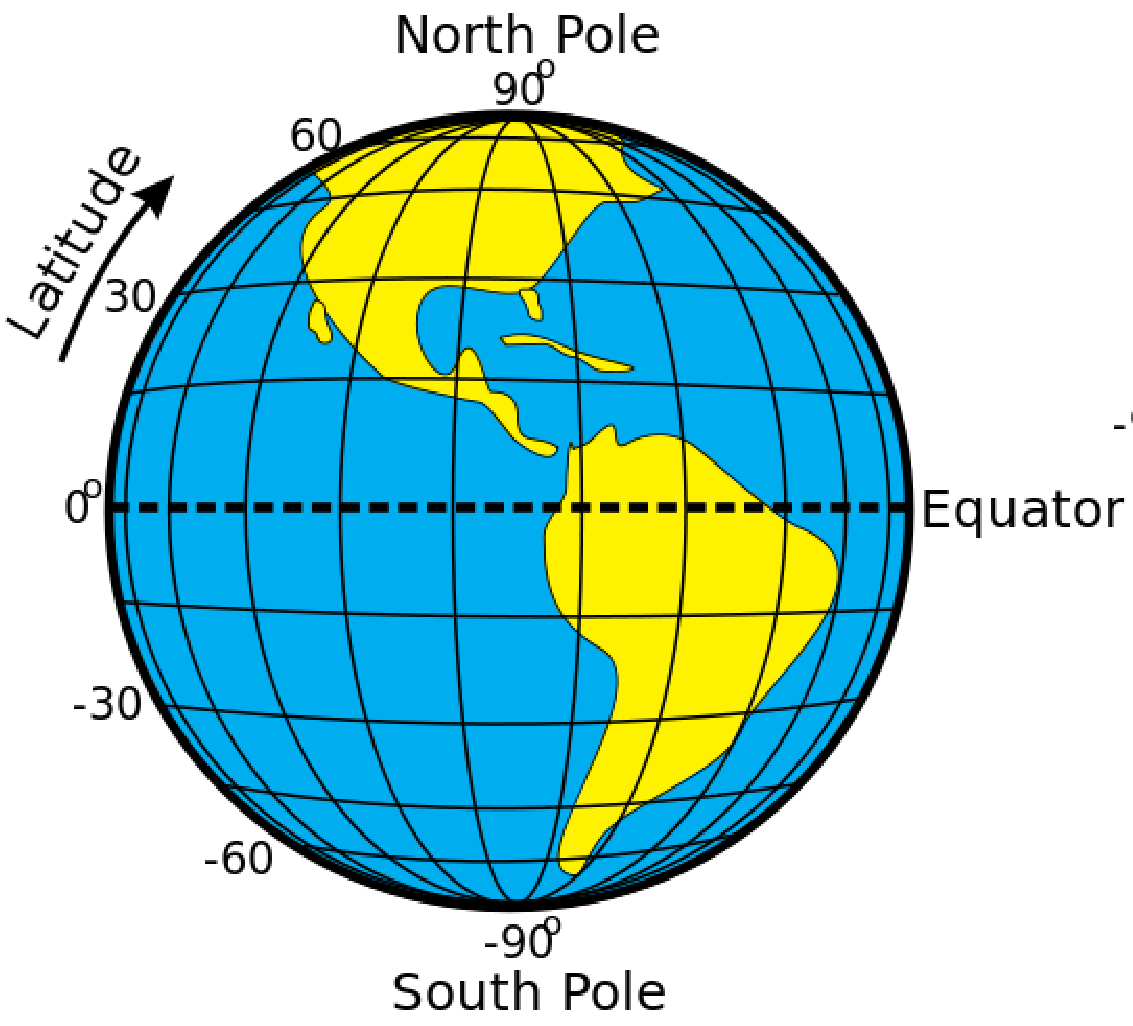
Longitude and Latitude Location

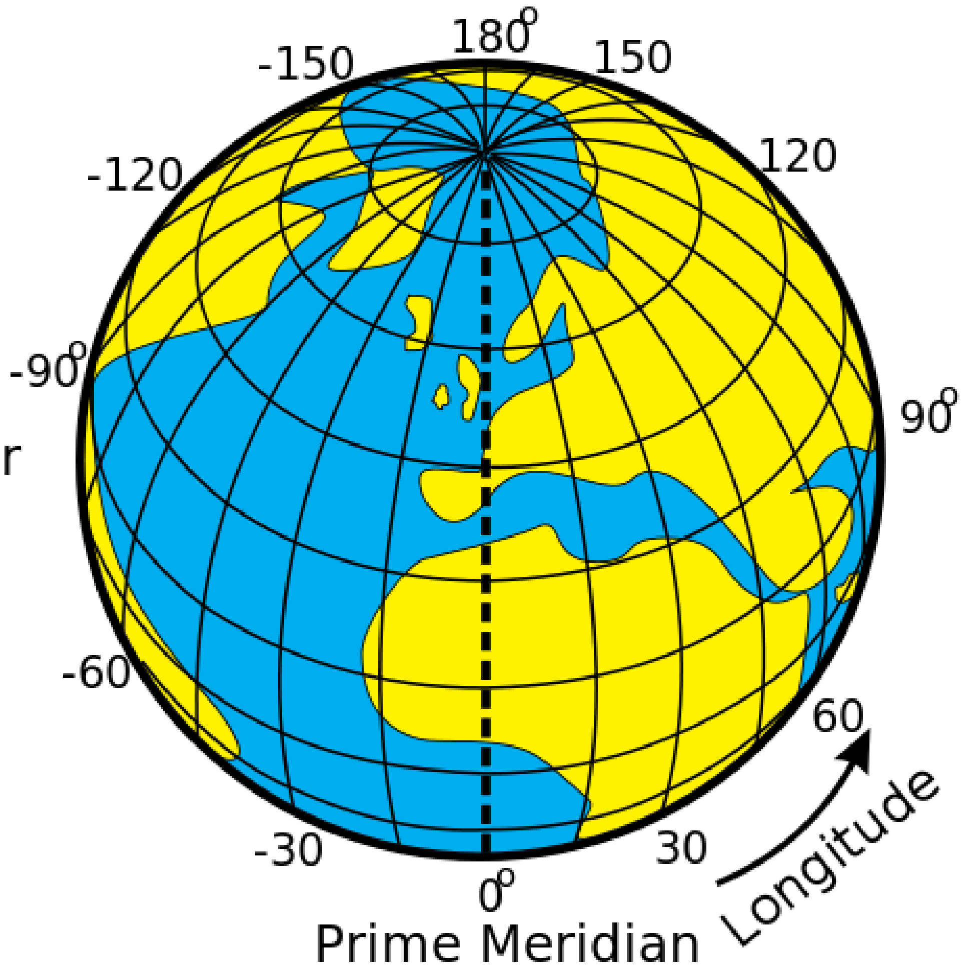
City locations are registered using longitude and latitude.

Latitude is an angle measured north or south of the equator. The North Pole is at Latitude 90 degrees North, and Blacksburg is at 37.22 degrees North.

Longitude is an angle measured east or west of a line that runs from pole to pole through Greenwich in the United Kingdom. Blacksburg is at 80.41 degrees east.

Longitude and Latitude





Longitude and Latitude Distance

To compute the distance between two cities whose coordinates are (Lat1,Lon1) and (Lat2,Lon2), we also need to know the radius of the earth, R = 3,959 miles

We convert degrees to radians:

```
theta1 = Lat1 * pi / 180; phi1 = Lon1 * pi / 180
theta2 = Lat2 * pi / 180; phi2 = Lon2 * pi / 180
```

and then:

```
X = cos(theta1)*cos(phi1) – cos(theta2)*cos(phi2)
Y = cos(theta1)*sin(phi1) – cos(theta2)*sin(phi2)
Z = sin(theta1) – sin(theta2)
XYZ = sqrt ( X^2 + Y^2 + Z^2 )
```

to compute the distance in miles:

```
D(City1,City2) = 2 * R * asin ( XYZ / 2  )
```

Compute Distance Table, Let’s Go!

The function “capitals.m” stores the name, latitude and longitude of each of the 48 capitals.

The function “city_dist_table.m” takes the information from capitals() and creates a 48x48 city-to-city distance table “Dist”.

We seem to have everything we need in order to schedule an efficient round trip for the traveling lobbyist!

When we start the program, we get no result, but no error. What could be going on? Why is the 5 city problem solved quickly, but the 48 city problem is choking?

We can experiment by using small versions of the distance matrix. In other words, we can pass in the submatrix Dist(1:5,1:5) to solve a problem with just 5 state capitols, and see if the program works, and then move on from there.

Experiments: 48 versus 5

```
city = capitals () ;
distance = city_dist_table ( city );
[ tsp_distance, tsp_trip ] = tsp_brute ( distance );
```

...Long long wait (*seconds, minutes, hours*) with no result...

```
d5 = distance(1:5,1:5);
[ tsp_distance, tsp_trip ] = tsp_brute ( d5 );
```

...Answer comes back almost **immediately**!

Solve a sequence of problems

```
function capital_timing()
  city = capitals ( );
  distance = city_dist_table ( city );
  for s = 5 : 12
    tic
    [ tsp_d, tsp_t ] = tsp_brute ( distance(1:s,1:s) );
    toc
  end
return
end
```

```
>> capital_timing
5: Elapsed time is 0.000596 seconds.
6: Elapsed time is 0.002287 seconds.
7: Elapsed time is 0.013985 seconds.
8: Elapsed time is 0.067820 seconds.
9: Elapsed time is 0.483606 seconds.
10: Elapsed time is 4.799959 seconds.
11: Elapsed time is 53.652944 seconds.
12: Elapsed time is 708.086179 seconds.
```

Problem N is N times as hard as Problem N-1

```
>> capital_timing
5: Elapsed time is 0.000596 seconds.
6: Elapsed time is 0.002287 seconds.        6 *   0.000596 =      0.0036
7: Elapsed time is 0.013985 seconds.        7 *   0.002287 =      0.0160
8: Elapsed time is 0.067820 seconds.        8 *   0.013985 =      0.1119
9: Elapsed time is 0.483606 seconds.        9 *   0.067820 =      0.6104
10: Elapsed time is 4.799959 seconds.      10 *  0.483606 =      4.8361
11: Elapsed time is 53.652944 seconds.     11 *  4.799959 =     52.7995
12: Elapsed time is 708.086179 seconds.  12 * 53.652944 = 643.8348
```

Brute Force Impossible for 48 Cities

If a 12 city calculation takes about 10 minutes, then a 13 city calculation takes about 13 * 10 minutes, and a 48 city calculation would take about 48*47*46*...*13*10 minutes or more than 2*10^53 (2 followed by 53 zeros) minutes.

The universe is about 13.7 billion years old, which works out to about 7*10^15 (7 followed by 15 zeros) minutes.

Our algorithm is correct, but our problem becomes unbelievably difficult as the size N increases.

What Is To Be Done?

Every day, businesses and researchers need to solve large versions of the TSP.

The Brute Force method cannot be used to provide an exact answer except for very small problems.

If we can't guarantee the best solution, perhaps we can try for reasonable approximations.

TSP Random Sampling

Our brute force search checked every possible permutation, but we see there are too many to do a complete check.

MATLAB provides a function to compute a random permutation of integers 1 to N:

```
p = randperm ( n );
```

Perhaps we could use this function to sample a large number of possible trips and take the best one we encounter as an **approximation** to the solution.

tsp_random.m

```
function [ tsp_distance, tsp_trip ] = tsp_random ( distance, trip_num )

  [ n, ~ ] = size ( distance );

  for i = 1 : trip_num

    this_trip = randperm ( n );

    this_distance = trip_distance ( distance, this_trip );

    if ( i == 1 || this_distance < tsp_distance )
      tsp_distance = this_distance;
      tsp_trip = this_trip;
    end

  end

  return
end
```

48 Capital Test

Samples	Shortest Length	
100	37,275 miles	
1,000	37,939	
10,000	36,973	
100,000	33,978	
1,000,000	30,976	
10,000,000	31,834	
100,000,000	29,316	(338 seconds)

Limits to Sampling

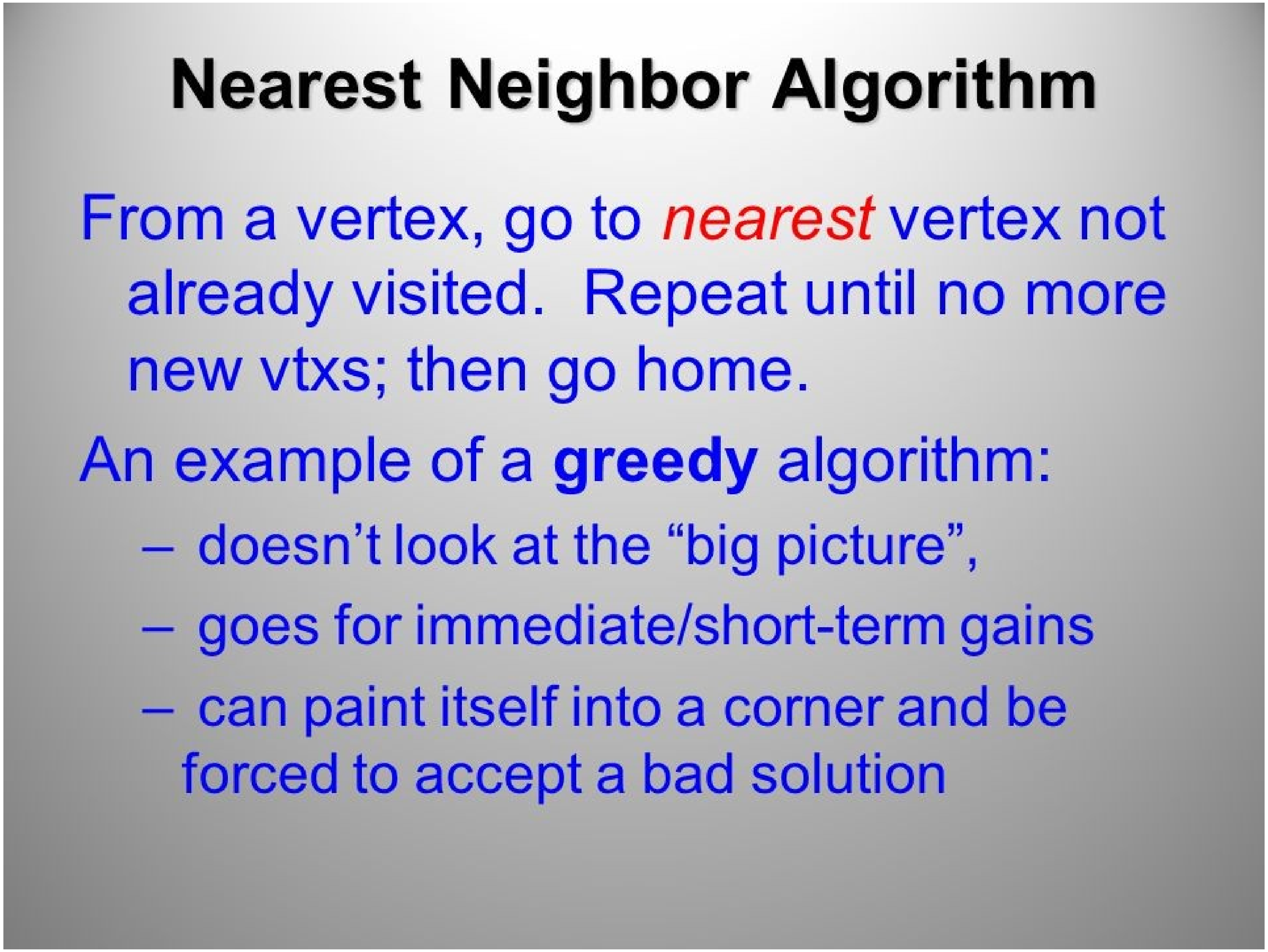
Our random sampling method seems to produce better answers as we increase the sample size.

However, because the results drop substantially as we increase the sample size, we are probably still far from a good route.

In fact, the shortest route has length of about 10,618 miles, so we really are far off.

Randomness may not be enough to solve this problem. Perhaps we need to try to add some simple rules of thumb to guide our code to a better solution faster.

Nearest Neighbor Method



Nearest Neighbor Algorithm

From a vertex, go to *nearest* vertex not already visited. Repeat until no more new vtxs; then go home.

An example of a **greedy** algorithm:

- doesn’t look at the “big picture”,
- goes for immediate/short-term gains
- can paint itself into a corner and be forced to accept a bad solution

The Nearest Neighbor Idea

Suppose we start our round trip without a plan, but we know we want to minimize the total travel distance.

Instead of worrying about our entire itinerary, let's just pick our next city. If we're going to minimize distance, it makes sense to travel to the closest city.

Our next choice is similar: travel to the nearest city (as long as you haven't already been there.)

This simple rule will produce a complete itinerary very quickly. It may not be the best solution, but let's see how it compares to our tsp_random results!

Nearest Neighbor Program

1) We need to specify a starting city.
2) If we are at city I, we look in row I of the distance matrix for the smallest value.
3) Actually, the smallest value associated with a city we haven’t yet visited.
4) As we visit each city, we can reset its distances to Infinity, so we automatically won’t visit it again.

tsp_nearest.m

```
function t  = tsp_nearest ( dist, nearest )

  [ n, n ] = size ( dist );
  t = zeros ( 1, n );

  for i = 1 : n

   here = nearest
    dist(1:n,here) = Inf;
    t(i) = here;

    dmin = Inf;
    nearest = 0;
    for j = 1 : n
       if ( dist(here,j) < dmin )
          dmin = dist(here,j);
          nearest = j;
      end
    end

  end

  return
end
```

Demo on Five City Example

```
Start  Trip            Length
-----  --------------  ------
   1:  1, 3, 5, 2, 4   270 miles
   2:  2, 3, 1, 5, 4   310
   3:  3, 1, 5, 2, 4   290
   4:  4, 2, 3, 1, 5   270
   5:  5, 3, 1, 2, 4   280
```

Actually, 270 miles is the shortest route, but for this small problem, our success is not a surprise. What happens if we look at the bigger problem?

Will it take a long time?

Will it get a good result?

Demo on the 48 US Capital Data

```
dist = capital_distance ( );
t = tsp_nearest ( dist, 1 );
tsp_dist = trip_distance ( dist, t );
tsp_dist => 11,378  (less than 1 second)

t = tsp_nearest ( dist, 2 );
tsp_dist => 11,738
t = tsp_nearest ( dist, 3 );
tsp_dist => 11,657
```

and so on.

Remember that the exact solution is 10,618 miles. Our brute force method never finished, and our random sampling gave trips of lengths 30,000 or more, so this nearest neighbor idea is a huge improvement.