Optimization of Discrete Problems

http://people.sc.fsu.edu/∼jburkardt/isc/week11
lecture_21.pdf

....

ISC3313:
Introduction to Scientific Computing with C++
Summer Semester 2011

....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 20 July 2011

Optimization of Discrete Problems

INTRO: Schedule

Next Class:

- Image Processing

Assignment:

- Programming Assignment #8 is due today.
- Programming Assignment #9 will be due July 28.

INTRO: Two Weeks Left For Projects!

Please remember that your project is due on Tuesday, August 2nd, at class time, 11am! This includes:

- a 5 minute oral presentation;
- a 3-5 page report to be turned in;
- a C++ program, to be turned in.

If you do not present your talk and turn in your report and programs on time, you will not receive a grade for the course.

Optimization of Discrete Problems

EXAMPLE: # 1, the Combination Lock

The Combination Lock:

EXAMPLE: # 1, the Combination Lock

A typical numeric combination lock contains **N** positions, each of which can be set to any digit from 0 to 9. We can think of the combination **c** as a number, or as an array of digits **c[0]** through **c[n-1]**. For our purposes, it will be better to use the array approach.

We assume there is a **bool** function **open(n,c)** which returns **true** if **c** contains the correct set of digits.

A typical combination lock will only open for one combination, but we might have similar problems in which several combinations will be acceptable.

EXAMPLE: # 2, the Button Lock

The Button Lock:

EXAMPLE: # 2, the Button Lock

A button lock often involves 10 buttons labeled 0 through 9, as well as **open** and **clear** buttons.

The length of the secret combination might be anything from 1 to 10 digits.

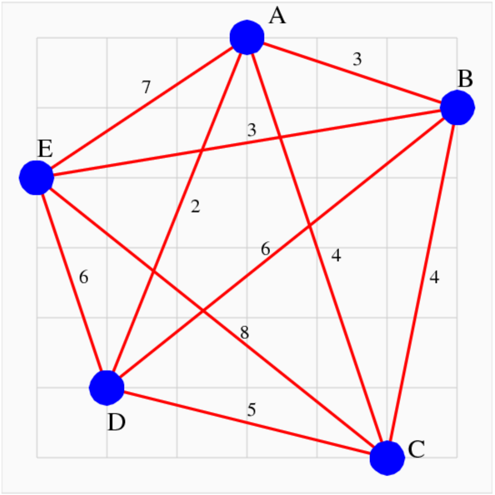
The user tries a combination by pressing down one button, then another, and so on, and finally pushing the **open** button.

If the combination is incorrect, all the buttons pop back up, and the user must try again.

The interesting feature here is that no digit can be repeated in the combination.

EXAMPLE: # 3, the Round Trip

The Round Trip



EXAMPLE: # 3, the Round Trip

Given a list of cities, and the distance between any pair of cities, we want to plan a trip, starting at city 1, that visits every city once. What is the length of the shortest round trip we can choose?

To make life simpler, we can number the cities 1 through 5. If we start at city 1, a round trip might be “1, 4, 3, 2, 5, 1”. Since we know we start at 1, we can shorten this to “4, 3, 2, 5, 1”, so that each of the five cities appears exactly once on the list. In that case, a round trip can be described as a permutation!

One way to find the shortest distance is to count every possible round trip, computing the length of each trip, and remembering the trip corresponding to the shortest total distance.

Optimization of Discrete Problems

BRUTE:

A discrete problem is one in which the variables do not constitute a continuous range (like the real numbers) but rather form a set that can be listed (like the integers).

If the list is finite, and not enormous, then one way to determine the best set of variables is simply to examine every possible set. Such an approach is called a brute force method, because, it seems, no mental power has been applied to the problem, and we are simply plowing through the data mindlessly.

However, even a problem that is to be handled by brute force can have some complications. For instance, it may not be a simple matter to produce an orderly list of all the possibilities. And w may discover that the possibilities, although finite, are simply to many to examine.

BRUTE: A Wrong Combination Tells Us Nothing

The combination lock problem is very hard. If you are unsuccessful with one combination, this tells you nothing about whether you are close to the right answer or not.

In contrast, if we are trying to find the zero of a function $f()$, then the fact that $f(x)$ is not zero tells us that the value x doesn't solved the problem, but if its value is small or large, that hints that we are close or far, and as $f()$ changes we believe we are getting closer or further away to the answer.

But with the combination lock problem, it seems the only way to open the lock is to try every combination.

BRUTE: “Counting” Through The Possibilities

Let us suppose there are 5 numbers to set, and that the digits run from 1 to 8. This means, in particular, that there are 8*8*8*8*8 = 32,768 possible combinations to try.

If a computer can do a billion things a second, then as long as each combination doesn’t take too long to set up in our program, we can solve the problem in under a second.

One way to check all the combinations is to march from (1,1,1,1,1) on up. If we think about the digits as representing a number, then we can plan to check the combinations in the order:

BRUTE: "Counting" Through The Possibilities

```
1 1 1 1 1 <--- Increment digit 5.
1 1 1 1 2 <--- Increment digit 5.
1 1 1 1 3 <--- Increment digit 5.
.........
1 1 1 1 7 <--- Increment digit 5.
1 1 1 1 8 <--- Can't increment digit 5, try digit 4.
1 1 1 2 1 <--- Increment digit 5.
1 1 1 2 2 <--- Increment digit 5.
1 1 1 2 3 <--- Increment digit 5.
.........
1 1 1 8 8 <--- Can't increment digit 5 or 4, try 3.
1 1 2 1 1 <--- Increment digit 5.
.........
8 8 8 8 8 <--- Can't increment any digit.  STOP!
```

BRUTE: “Counting” Through The Possibilities

Let’s turn this simple counting idea into an algorithm.

We’ll assume we have an array **d[]** of length **n**, and that each entry of **d[]** is a digit between **dmin** and **dmax**.

We’ll start with the combination in which every entry is equal to **dmin**, and, one by one, generate the next combination. The next combination can be found by starting at the last digit, and

- if digit less than **dmax**, increment by 1 for next combination;
- otherwise, reset it to **dmin** and check the previous digit;
- but if there is no previous digit, we are done.

BRUTE: A Counting Algorithm

counting.cpp: assume minimum digit is 1, maximum is 8.

```
  for ( i = 0; i < n; i++ )
  {
    d[i] = 1;
  }

  while ( true )
  {
//  Working backwards, set 8's to 1, otherwise increment.

    inc = -1;              <-- Setting inc to -1 indicates we haven't found it yet.
    for ( i = n - 1; 0 <= i; i-- )
    {
      if ( d[i] == 8 )
      {
        d[i] = 1;          <-- Change 8's to 1
      }
      else
      {
        d[i] = d[i] + 1;  <-- Increment the first non-8 you find and break!
        inc = i;
        break;
      }
    }
//  If nothing to increment, exit!

    if ( inc == -1 )       <-- If inc is still -1, we never found anything but 8's.
    {
       break;
    }
  }
```

BRUTE: Solving the Combination Lock Problem

```
  found = false;

  for ( i = 0; i < n; i++ )
  {
    d[i] = 1;
  }

  while ( true )
  {
//  Try the combination.

    if ( open ( n, d ) )
    {
      found = true;
      cout << "  Found the combination!\n";
      break;
    }
//  Working backwards, set 8's to 1, otherwise increment.
...(same as previous page)...
//  If nothing to increment, exit!
...(same as previous page)...
  }

  if ( !found )
  {
    cout << "Could not find the combination!\n";
  }
  return 0;
}
```

BRUTE: The OPEN function

```
bool open ( int n, int d[] )
{
  bool value;
  if ( d[0] == 1 && d[1] == 8 && d[2] == 4 && d[3] == 5 )
  {
    value = true;
  }
  else
  {
    value = false;
  }
  return value;
}
```

and the output from running the program should be:

```
Found the combination:   1 8 4 5
```

BRUTE: the Button Lock

To simplify this problem, let's assume that the combination always uses every digit. That is, a combination is some particular ordering of the ten digits 0 through 9.

Could we write a program which would generate each possible combination in some orderly fashion?

Obviously, one way is simple to count from 0000000000 to 999999999, but tossing out every combination with a repeated digit. This will work, but it is very slow.

A second approach is to realize that we are asking for all possible permutations of the digits 0 through 9. There are already algorithms available to generate these permutations, one at a [illegible] which is exactly what we need.

BRUTE: “Next Permutation” Algorithm

The code we will use is called **next_perm.cpp**.

The code has the declaration:

```
void next_perm ( int n, int p[], int &rank );
```

The first time we call, we set **n** to the number of entries or digits in the permutation, and we set **rank** to -1. The program computes the first permutation, which is (0,1,2,...,n-1), and changes the value of **rank** to 1. If we call again, the next permutation is returned, and this keeps happening until the program runs out of permutations. At that point, it sets **rank** to -1.

BRUTE: “Next Permutation” Algorithm

perm_test.cpp:

```
# include <cstdlib>
# include <iostream>
void perm_next ( int n, int p[], int &rank );
using namespace std;

int main ( )
{
  int i, n = 4, p[4], rank;

  rank = -1;

  while ( true )
  {

    perm_next ( n, p, rank );  <-- Try to get next permutation

    if ( rank == -1 )          <-- Quit if no more
    {
      break;
    }

    for ( i = 0; i < n; i++ )  <-- In this example, we simply print it.
    {
      cout << "  " << p[i];
    }
    cout << "\n";

  }
  return 0;
}
```

BRUTE: Counting 3 Digit Combinations

button_perm.cpp: Thus, our algorithm to try all the combinations on the button lock is simply:

```
found = false;
rank = -1;

while ( true )

  next_perm ( n, p, rank );

  if ( no more permutations )
    break;
  end

  if ( open ( n, p ) )              <--- Does this permutation open the lock?
    found = true;
    break;
  end

end
```

and this is an example of the way that, in a brute force approa the hard part can be figuring out how to try all possibilities i a orderly fashion!

BRUTE: the Round Trip

trip_counting.cpp: If **p[]** is a vector listing the five cities we visited, then a brute force approach would be:

```
set dmin = 1000000;

while ( true )

  generate the next trip vector p (permutation)

  if no more permutations
    break

  d = sum ( city to city distances for this trip )

  if ( d < dmin )
    dmin = d
  end

end
```

```
Shortest round trip has length 19
  0  2  1  4  3
  A  C  B  E  D
```

Optimization of Discrete Problems

HEURISTICS: A More Realistic Problem

How many round trip itineraries are possible with 48 cities?

HEURISTICS: Brute Force Fails

Given **n** cities, the number of permutations is $n!$. If we try to find a round trip through the capitals of the "lower 48" states, simply to count the possible permutations requires 48! steps, which is an astronomical number. It is bigger, for instance, than the largest integer you can represent with a C++ **int**!

Since problems this big (and much bigger) need to be solved, people have looked at methods for getting good approximate solutions. Sometimes, you can make a convincing case why a particular approach has a chance of producing a good solution. A method that is not proved to work, but which suggests that it often does well, is called a heuristic method.

A heuristic is a method or procedure for making a good guess.

HEURISTICS: Move to the Nearest Unvisited City. Repeat.

Our heuristic for the round trip problem works as follows:

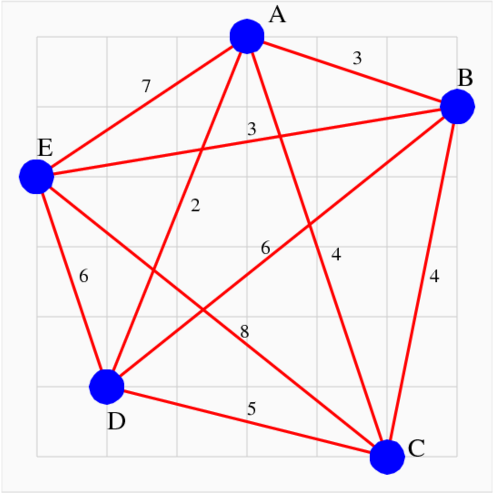
Pick a starting point at random, then build an itinerary by moving from your current location to the nearest unvisited city, again and again, until you must return home.

If you have **n** nodes, then you can be more methodical, and do the computation once for each possible starting city.

After generating many itineraries, choose the shortest one.

HEURISTICS: the Round Trip

The Round Trip



TSP: The Nearest City Heuristic

Let's use our heuristic on our simple 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.

Optimization of Discrete Problems

HILL: Try to Make a Good Solution Better

In the optimization method called hill climbing, we start with a candidate **A** for the best solution, which might be chosen at random. We evaluate the function $f(A)$, and wonder if there is a better solution.

We also have some idea of how to make small variations in the solution **A**. If **A** can be represented as a sequence of integers, the small change might be to swap two integers, or to increase or decrease one value by 1, for instance. Thus, we are considering other solutions that are “nearby” to **A**.

As soon as we find a neighbor **B** such that $f(A) < f(B)$, we m **B** our new candidate solution, and repeat the “twiddling” pr But if no small change to **A** makes an improvement, we stop.

HILL: Improving a Good Round Trip

We can use a kind of hill climbing idea for the round trip problem.

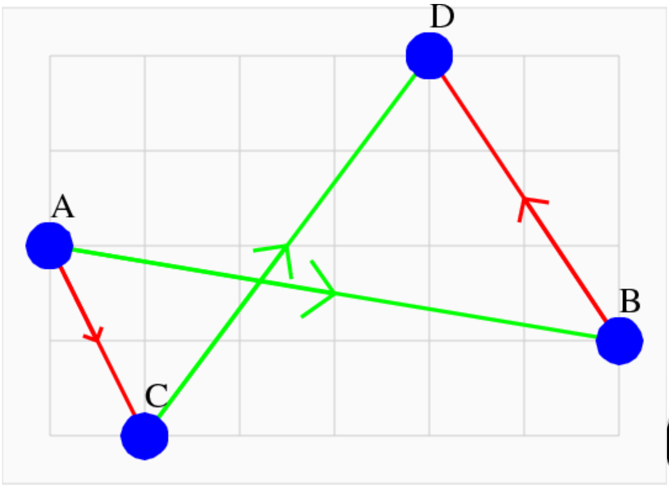
Suppose that the intercity distances are “real” distances, so that they satisfy the triangle inequality:

```
  distance (A to C) <= distance (A to B) + 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.

HILL: Replace Green Lines by Red Ones

Quiz: Can you see that the new itinerary must be shorter!



HILL: Reversing One Segment

Another possible way to improve a good round trip is simply to consider the effect of reversing the order of one segment of the trip. That is,

```
This:        ... -> C4 -> C5 -> C6 -> C7 -> C8 -> C9 -> ..
becomes this ... -> C4 -> C8 -> C7 -> C6 -> C5 -> C9 -> ..
```

By reversing the order, we have reduced the length by Dist(C4,C5) + Dist(C8,C9), and increased it by Dist(C4,C8) + Dist(C5,C9), so it's possible this reduces our distance, but the only way to know is to check.

Thus, the hill climbing technique allows us to keep our good solution, and to consider small modifications that might impr . If we find a good modification, we take it, and repeat the proces

HILL: A Random Permutation

Both ideas for the round trip problem require that we start with some solution, which we could select using the heuristic idea, or at random... except...how do we pick a permutation at random?

It's like picking cards out of a deck. Here's one way to do it, supposing that we have **int random_int(int a,int b)**, our function which picks a random integer between **a** and **b**:

```
int p[n] = { 0, 1, 2, ..., n-1};

for ( i = 0; i < n - 1; i++ )
{
  j = random_int ( i, n - 1 );
  t    = p[j];
  p[j] = p[i];
  p[i] = t;
}
```

Optimization of Discrete Problems

SHORT:

The shortest path problem is similar, in some ways, to the round trip problem.

In particular, we can think of our problem data as involving a set of cities, and the distances between them.

For the round trip problem, it was possible to move directly from any city to any other, as though we were using an airplane. In the shortest path problem, however, it is common to assume that there are direct routes only between some pairs of cities. This means the problem is more like taking a road trip. Presumably, you can get from any city A to any other city B, but perhaps you must go through a few other cities to get there.

SHORT: What is the Length of a Path?

We can think of a path **P** from city **A** to city **B** as a sequence of steps of the form {A,J}, {J,W}, {W,K}, {K,B}, (for instance), with the property that the first step starts at **A**, the last step ends at **B**, and we never visit a city twice.

Assuming we have a matrix or array **dist[][]**, then the length of the path P is simply the sum of the inter-city distances, which we might write as:

```
length(P) = dist[A][J] + dist[J][W]
          + dist[W][K] + dist[K][B].
```

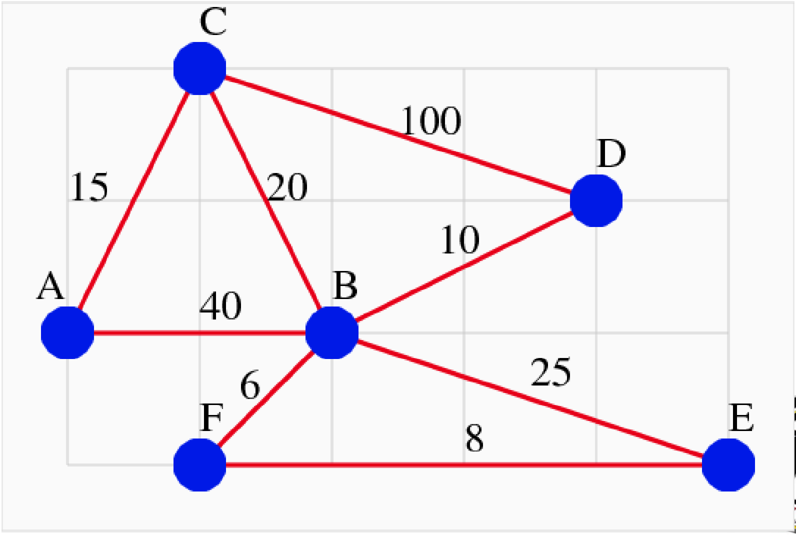
SHORT: The Shortest Path Problem

The shortest path problem asks for the length of the shortest path between any two given cities.

For convenience, let's assume the starting city is always city A, which, we can also think of as city #1 or, if we're working in C++, as city #0.

If there isn't a direct road from city I to city J, then we'd like to put the value ∞ in the **dist[i][j]** matrix. For now, it will be good enough just to put in a very large value, such as 1,000,000.

SHORT: An Example Road Map



SHORT: An Example DIST[][] Array

Our road map would be:

```
  0   40   15   INF   INF   INF
 40    0   20    10    25     6
 15   20    0   100   INF   INF
INF   10  100     0   INF   INF
INF   25  INF   INF     0     8
INF    6  INF   INF     8     0
```

or, in C++

```
dist[5][5]={{  0,  40,   15, 1e6, 1e6,  1e6 },
            { 40,   0,   20,  10,  25,    6 },
            { 15,  20,    0, 100, 1e6,  1e6 },
            {1e6,  10,  100,   0, 1e6,  1e6 },
            {1e6,  25,  1e6, 1e6,   0,    8 },
            {1e6,   6,  1e6, 1e6,   8,    0 } };
```

SHORT: Constructing the DISTMIN Array

Now we understand that the **dist[][]** array is keeping track of the direct distance between two cities. Where the array records an “infinity” value, it is actually possible to reach one city from the other, by traveling through intermediate cities.

Moreover, we can imagine that it is possible that a two-step journey might actually be *shorter than the direct route.*

So our task is to create determine the minimum distance from city A to city B, a task which:

- considers all possible paths between two cities, not just the one step path;
- records the length of the shortest path.

SHORT: A Brute Force Approach

How would we design a *brute force* approach?

We would have to generate all possible paths, one at a time, computing their lengths and keeping the shortest one.

Of course, once again, the disadvantage of a brute force method is that you have to have some idea of how to create every possible solution. In this case, you have to figure out how to set up every possible path from one city to the other.

There is a method called "depth first search", from graph theory, that would let us do this, so the brute force approach can be made to work. However, it only works for very small problems, because there are so many possible paths to check. So let us move on smarter procedure.

SHORT: Two Algorithms

The shortest path problem is important enough that people have worked hard to find efficient and simple algorithms to solve it.

One method by Edsger Dijkstra solves the problem for a single starting city, and requires the natural assumption that the direct distances are never negative (you can imagine the problems a negative distance creates!).

Instead, we will consider a more general method, by Robert Floyd, which will compute the shortest distance between every pair of cities, and which works even if the direct distance from city A to city B is not the same as the distance going the other way. (It ca even allow a limited number of negative distances!)

SHORT: The Data

Floyd's algorithm uses a single 2D array **d[][]**.

When we initialize the problem, we put into **d[A][B]** the direct distance from city **A** to city **B**. If there is no direct link, we would like to set the value of **d[A][B]** to ∞, but instead we put in a "large value". For us, a value of 1000 will be large enough.

The input is **n**, the number of cities, and the array **d[][]**.

The output is a modified version of **d[][]**. The output value of **d[A][B]** is the shortest distance from city **A** to city **B**, using any number of intermediate roads to get there.

If there is no route whatsoever between the two cities, then the corresponding entry of **d[][]** will be at least 1,000, which repr[illegible]s ∞, that is, "no way to get there!"

SHORT: How the Algorithm Works

The algorithm is surprisingly simple.

Suppose we already know the shortest distance between cities A and B that only uses intermediate cities numbered less than the value K.

What does that tell us about the shortest distance if we are allowed to include city K? If adding city K gives us a better route, then it comes about because we traveled from city A to city K, and then from city K to city B.

So if we add city K to the list of cities we are allowed to visit, we update the distance between cities A and B by:

```
d[A][B] = min ( d[A][B], d[A][K] + D[K][B] );
```

The case $K = 0$ is where we start, we stop at case $K = N$.

SHORT: A C++ Version

```
void i4mat_floyd ( int n, int d[] )
{
  int i, j, k;

  for ( k = 0; k < n; k++ )                                          <-- Include intermediate cities 0 through K
  {
    for ( j = 0; j < n; j++ )                                        <-- Start at city I...
    {
      for ( i = 0; i < n; i++ )                                      <-- ...and travel to city J.
      {
        d[i][j]   = i4_min ( d[i][j],   d[i][k]   + d[k][j]   );    <-- What we'd like to write.
        d[i+j*n] = i4_min ( d[i+j*n], d[i+k*n] + d[k+j*n] );        <-- What we have to write instead!
      }
    }
  }
  return;
}
```

It's not easy to pass a "matrix" or 2D array to a function in C++ if you allow the dimensions to vary. The FLOYD library assumes the 2D array is stored as a 1D array, listing rows one after another. That means that **d[i][j]** is expressed, instead, as **d[i+j*n]**.

SHORT: The FLOYD Library

The FLOYD library includes four “interesting” functions:

- **i4mat_floyd(n,d)** carries out Floyd’s algorithm when **d** is an **int** array;
- **r8mat_floyd(n,d)** carries out Floyd’s algorithm when **d** is a **double** array;
- **i4mat_print(m,n,d,title)** prints an **m** by **n int** array, with a title;
- **r8mat_print(m,n,d,title)** prints an **m** by **n double** array, with a title.

Although our data is a 2D matrix, these functions all expect the 2D matrix to be stored as a 1D vector, that is, with the rows listed one after another. This is easy to set up in the initialization li you just list one row after another.

SHORT: Calling the Floyd Library

```
# include <cstdlib>

using namespace std;

# include "floyd.hpp"

int main ( )

//  FLOYD_TEST tests the FLOYD algorithm.
//
{
  int a[6*6] = {                               <-- For "technical reasons", set up
     0, 40,   15, 1000, 1000, 1000,                the 2D array as a 1D array!
    40,  0,   20,   10,   25,    6,
    15, 20,    0,  100, 1000, 1000,             <-- Note that "1000" counts as "infinity" here!
  1000, 10,  100,    0, 1000, 1000,
  1000, 25, 1000, 1000,    0,    8,
  1000,  6, 1000, 1000,    8,    0 };

  int n = 6;

  i4mat_print ( n, n, a, "  Direct Distance" );  <-- Print initial data

  i4mat_floyd ( n, a );                          <-- Compute the shortest distance.

  i4mat_print ( n, n, a, "  Floyd Distance" );   <-- Print results

  return 0;
}
```

SHORT: A Test

Here is the result of running the example case:

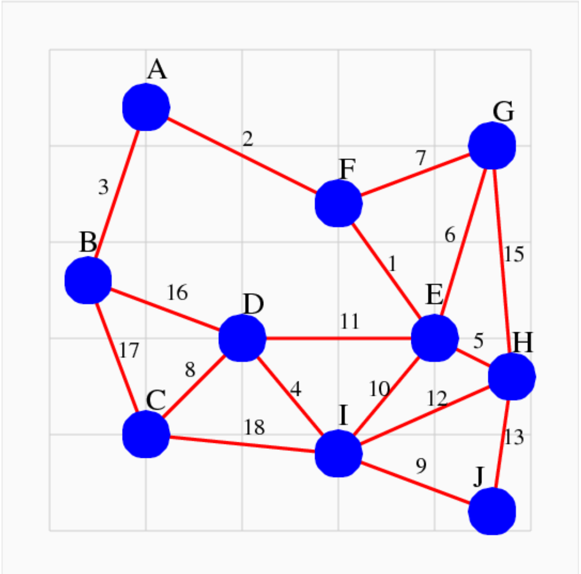
```
 Direct Distance

          1       2       3       4       5       6

 1        0      40      15    1000    1000    1000
 2       40       0      20      10      25       6
 3       15      20       0     100    1000    1000
 4     1000      10     100       0    1000    1000
 5     1000      25    1000    1000       0       8
 6     1000       6    1000    1000       8       0


Floyd Distance

          1       2       3       4       5       6

 1        0      35      15      45      49      41
 2       35       0      20      10      14       6
 3       15      20       0      30      34      26
 4       45      10      30       0      24      16
 5       49      14      34      24       0       8
 6       41       6      26      16       8       0
```

The output matrix is symmetric (because the input data was), and no city is "infinity" away from any other one.

Optimization of Discrete Problems

ASSIGNMENT #9: A Ten-City Road Map

Consider the following road map relating 10 cities.



ASSIGNMENT #9: Shortest Distance Between All Cities

Copy the programs **floyd_test.cpp**, **floyd.cpp** and the file **floyd.hpp**.

Modify the file **floyd_test.cpp** so that it solves the shortest path problem for the ten city road map. You only need to modify the values for the matrix **d** and the number of cities **n**.

A typical compiler command would be

```
g++ floyd_test.cpp floyd.cpp
mv a.out floyd_test
floyd_test > short.txt
```

Remember that **floyd.hpp** needs to be in the same directory where you do the compiling.

ASSIGNMENT #9: Things to Turn in

Email to Detelina:

- the output file **short.txt**:
- a copy of your modified program **floyd_test.cpp**.

The program and output are due by Thursday, July 28.

Note that this is the last program homework assignment.