A Programming Project 98b:
Finding Shortest Paths in a Graph

Due: 1/6/1998

Last updated: April 22, 1998

1 Introduction to the project

The goal of this project is to write a program named plgraph, which solves several types of problems of finding shortest paths in graphs. The program reads its input from a file and from the standard input. The output is written either to a file, or to the standard output as describe below. Each line in the input file represents a neighbors list of a vertex in a graph. plgraph accepts the file-name and the problem type from the user. There are two possible problems to solve: the single-source shortest-path problem and the all-pairs shortest paths problem. The user should choose:

- which of the two problem she wishes to solve.
- Which algorithm(s) to use (see description below).
- Which data structure to use (see description below).

We supply details below how the plgraph should communicate with the user. The algorithm that should be implemented are:

**Dijkstra algorithm** The Dijkstra algorithm for the single-source shortest-paths on a weighted, directed graph. All edge weights are nonnegative.

**Bellman-Ford algorithm** The Bellman-Ford algorithm for the single-source shortest-paths on a weighted, directed graph. Edge weights can be negative and positive.

**Matrix multiplication algorithm** Finding all-pairs shortest paths using the Matrix Multiplication algorithm on a weighted, directed graph.

**Repeated Dijkstra algorithm** Finding shortest paths between all pairs of vertices using single-source shortest paths previous algorithms on a weighted, directed graph. All edge weights are nonnegative. There is no ‘actual’ multiplication of matrices, but the basic operations we perform (and their attributes) are very similar to multiplication of matrices.

**Floyd algorithm** The Floyd algorithm for the all-pairs shortest-paths problem on a weighted, directed graph. Edge weights could be negative or positive.
1.1 Graphs — Definition and Representation

We consider a directed weighted graph, \( G(V, E) \), with no parallel edges, and no self loops, such that there is a weight \( w(u,v) \) associated with each edge \( (u,v) \in E \). The weight is an integer between \(-2^{16}\) and \(2^{16}\). This weight could be negative or positive, but never zero. See Cormen's book\(^1\) pages 465–568. There are two standard ways to represent a graph \( G(V,E) \): Either as a collection of adjacency lists, or as an adjacency matrix. In the project some of the algorithms assume the first form while others assume the second. The letter \( n \) always denotes the number of vertices in the graph, and \( m \) denotes the number of edges. We assume that the vertices are numbered \( 1, 2, \ldots, n \) in some arbitrary order.

The adjacency-list representation consists of an array \( L[1 \ldots n] \) of pointer, one for each vertex of \( V \). For each \( u_i \in V \), the pointer \( L[u_i] \) points to the first element of a (possibly empty) linked list. Each element in the list represents a different edge \( (u_i,v_j) \in E \), and since there are no parallel edges in the graph, we can do by indicating the neighbor \( v \) and the weight \( w(u_i,v) \). So the element corresponds to \((u_i,v) \in E\) is a structure contains, the neighbor vertex \( v \), the weight of the edge \( w(u_i,v) \), and a pointer to the next element in the linked list (or NULL if this is the last element). See example above. Clearly, if \((u_i,v) \notin E\) then \( v \) does not appear in the list.

In the adjacency-matrix representation we use a matrix \( M \) of \( n \times n \) entries. The matrix, \( m_{ij} \) is \( w(i,j) \) if \((i,j) \in E\). Otherwise, \( m_{ij} = 0 \). See example below.

1.2 Format of Input File

The input file contains information about the graph to be handled. It might contains remarks, where a remark line starts with a \# symbols at the first character. The first non-remark line contains an integer \( n \) of vertices in the graph. The only bound on \( n \) is that it is smaller than \(2^{20}\). The program should ignore remarks lines, and denote only nonremark lines. Each other non-remark line consists of three numbers, \( i, j \) and \( w(i,j) \), where \( i, j \) are labels of vertices, and we assume that the (directed) edge \((i,j) \in E \). The term \( w(i,j) \) is the weight of the edge. We show an example of a graph and its two representations.

```
adjacency list representation
1 -> (2,12)->(4,17)->NULL
2 -> (1,10)->NULL
3 -> (1,40)->(2,50)->(4,15)->NULL
4 -> NULL
5 -> (6,70)->NULL
6 -> NULL

adjacency matrix representation

The adjacency matrix
0 12 0 17 0 0
10 0 0 0 0 0
40 50 0 15 0 0
0 0 0 0 0 0
0 0 0 0 0 70
```

\(^1\)Introduction to Algorithms, Cormen, Leiserson and Rivest. A very popular book. - lots of copies in the library. Was also translated to Hebrew by the Open University. The first part could be already bought in the Open-University book store, and the other part will be ready shortly. To ease the studying, one also find it helpful to read the reading guide of the Open-University
2 Data Structure

There are several data structures each of the algorithms uses. Most of them are simple, and you would discover them yourself. An exception is the Dijkstra algorithm. This algorithm uses *heaps*, which is a data structure that supports the operations of insertion a new element, decreasing the value of an element, and deleting an element. In addition to the binary heaps, which are precisely the heaps you meet at the Data Structure course, you need to implement Binomial Heaps and Fibonacci Heaps. You will find the description of them in the pages we copied, and (with more details) in Cormen's book. Your program MUST contain the following procedures, with the running time written next to them. These running time are in the worst case for Binary and Binomial Heaps. It is in the *amortized* sense for FibonacciHeaps, which, informally could be understood as the average time to perform an operation, while averaging over a long series of operations. In the bounds written below, \( k \) is the number of elements in the heap. In particular you should support Binomial heaps and FibonacciHeaps.

The operations includes

<table>
<thead>
<tr>
<th>Function-name</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary-Heap-Make</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Binomial-Heap-Make</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Fibonacci-Heap-Make</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Binary-Heap-Insert</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Binomial-Heap-Insert</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Fibonacci-Heap-Insert</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Binary-Heap-Minimum</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>Binomial-Heap-Minimum</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Fibonacci-Heap-Minimum</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Binary-Heap-Extract-Min</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Binomial-Heap-Extract-Min</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Fibonacci-Heap-Extract-Min</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Binary-Heap-Union</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Binomial-Heap-Union</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Fibonacci-Heap-Union</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Binary-Heap-Decrease-Key</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Binomial-Heap-Extract-Min</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Fibonacci-Heap-Extract-Min</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>Binary-Heap-Delete</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Binomial-Heap-Delete</td>
<td>( O(\log n) )</td>
</tr>
<tr>
<td>Fibonacci-Heap-Extract-Delete</td>
<td>( O(\log n) )</td>
</tr>
</tbody>
</table>

After creating the empty heaps Using the Make operation, your program should construct them for the vertices of the graph, using the Insert operation. The 'Union' operation merge two heaps into a single heap. You need the 'Union' operations (in our application) only as a subroutine for other operations. The 'Minimum' operation returns the element (vertex) with minimal key among the elements in the heap, while the operation 'Extract-Min' remove this element from the heap. The operation 'Decrease-Key' decrease the key of an element in the heap, and updates the structure of the heap accordingly. 'Delete' operation receives a pointer
3 Representing shortest paths

The single-source shortest paths problem is finding, given a source \( s \), the length of shortest paths from \( s \) to each \( v \in V \) (which can be positive or negative). The length of a path is the sum of weights of the arcs of the path. This number could be positive or negative. A path is a cycle if its starting and ending vertices is the same vertex. A negative-length cycle is a cycle whose weight is negative. Of course, if our graph contains a negative length cycle, the distance between vertices is undefined, and if our program determines that such a cycle exists, it should stop computing the cycles. In addition, the solution should includes a shortest-paths tree, denoted as \( SPT(G) \), which is a compact way to represent all such shortest paths for a node \( s \) to all vertices. The representation is based on the lemma, which is not hard to prove, that by drawing all shortest paths from \( s \), we form a tree whose root is \( s \), which contains all vertices of the graph. \( plgraph \) should output this tree by finding for each vertex \( v \in V \setminus \{s\} \), its predecessor \( \pi[v] \) in \( SPT(G) \), denoted as as \( \pi[v] \). We define \( \pi[s] = \text{NULL} \).

We use \( SPT(G) \) as follows. Once seeking the shortest path from \( s \) to some vertex \( v \in V \), by first traversing this path backward (that is, from \( v \) to \( s \)). This is obtained by moving from \( v \) to \( v_1 = \pi[v] \), from \( v_1 \) to \( v_2 = \pi[v_1] \) etc. until we reach \( s \).

4 The algorithms

These are the algorithm \( plgraph \) should implement.

4.1 Dijkstra Algorithm

The algorithm computes the distance of each vertex in the graph from a fixed vertex \( s \in V \). When the algorithm terminates, the distance of a vertex \( v \) from \( s \) is stored at a variable \( d[v] \). The predecessor of \( v \) at the \( SPT(G) \) tree is stored in \( \pi[v] \). All vertices whose distance from \( s \) is not determined yet, are stored in a set \( S \subseteq V \). In each iteration, the algorithm chooses a vertex from \( S \), and updates the values of \( d[\_] \), \( \pi[\_] \) of (some of the) vertices in \( V \setminus S \) as described below. Be alarmed that the algorithm works properly only when the weights of all edges are non-negative. The running time is \( O(n \log n + m \cdot T(m)) \) where \( T(m) \) is the time required to perform a “Decrease” operation etc.

**Initialization:**  \[ S = \emptyset; \ d[s] = 0; \ d[v] = \infty, \forall v \in V \setminus \{s\}. \]

**While** \( V \neq S \) **Do** Begin

**Choose** a point \( u \in V \setminus S \) that minimizes \( d[u] \) and add \( u \) to \( S \).

**For** neighbor \( v \) of \( u \) which is inside \( V \setminus S \) **Do**

**If** \( d[v] > d[u] + w(u, v) \) **Then** Begin

\[ d[v] = d[u] + w(u, v) \]

\[ \pi[v] = u \]

End ;

End **While**

**Report** \( \pi[\_], d[\_]. \)
Observe that the algorithm performs \( m \) Decrease-key operations, and \( n \) deletions on any of the types of heaps mentioned above.

### 4.2 Repeated Dijkstra

If all edge-weights are positive, we can solving the all-pairs distances problem by simply applying Dijkstra algorithm from each vertex in \( V \). That is, each vertex \( v \) is the source in its turn. The running time \( O(n^2 \log n + nm \cdot T(m)) \) where \( T(m) \) is the time required to perform a “Decrease” operation etc.

### 4.3 Bellman-Ford Algorithm

This algorithm Computes the distance of each vertex of the graph from a pre-determined vertex \( s \). Its running time is larger than Dijkstra algorithm, but it can also be implemented when weights of (some of the) edges are negative. The running time is \( O(mn \log n) \). It is also useful to discover when negative-length cycle exist in \( G \) (see definition above).

<table>
<thead>
<tr>
<th>Initialization</th>
<th>( d[s] = 0; \ d[v] = \infty, \ \forall v \in V )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repeat ( n - 1 ) times</td>
<td>For each edge ((u, v) \in E)</td>
</tr>
<tr>
<td></td>
<td>If ( d[v] &gt; d[u] + w(u, v) ) Then Begin</td>
</tr>
<tr>
<td></td>
<td>( d[v] = d[u] + w(u, v) )</td>
</tr>
<tr>
<td></td>
<td>( \pi[v] = u )</td>
</tr>
<tr>
<td></td>
<td>End</td>
</tr>
<tr>
<td></td>
<td>End</td>
</tr>
<tr>
<td>Apply again the inner loop (starting with “For ”).</td>
<td>If the value of ( d[v] ) is changed in the last operation,</td>
</tr>
<tr>
<td></td>
<td>Then report that the graph contains a negative-length cycle.</td>
</tr>
<tr>
<td></td>
<td>Else report ( \pi[\cdot], d[\cdot] ).</td>
</tr>
</tbody>
</table>

### 4.4 Floyd’s Algorithm

This algorithm Computes the distance between each pair of vertices \( x, y \in V \). The edges weights could be negative and positive. For simplicity we assume that the vertices are labeled 1, 2, \ldots \( n \). We note by \( d^{(i)}(u, v) \) (for any \( u, v \in V \)) the distance of the shortest path from \( u \) to \( v \), that does not pass through any of the vertices \( i + 1, i + 2, \ldots n \), that is, that none of the vertices \( i + 1, i + 2, \ldots n \) is a vertex on the path (vertices from \( i + 1, i + 2, \ldots n \) might, however, be the first or the last one on the path). So \( d^{(i)}(u, v) \) is just the \( w(u, v) \) if an edge from \( u \) to \( v \) exists in \( E \), or \( \infty \) if \((u, v) \notin E \). Note that \( d^{(n)}(u, v) \) is the distance from \( u \) to \( v \). In the implementation of the algorithm, \( d^{(j)}(u, v) \) is implemented by two two-dimensional matrices (think how). The running time is \( O(n^3) \).

In addition, we define \( \pi[u, v] \) as follows. \( \pi[u, v] \) stores a vertex \( x \in V \) if \( x \) is the successor vertex to \( v \) on the shortest path from \( u \) to \( v \). So the vertices on that path (in reverse order are \( u, \pi[u, v], \pi[u, \pi[u, v]], \pi[u, \pi[u, \pi[u, v]]], \text{ etc.} \)
Floyd Algorithm

**Initialization:** For each $u, v \in V$,
- If $(u, v) \notin E$
  - Then $d^{(0)}[u, v] = \infty$ and $\pi^{(0)}[u, v] = \text{NULL}$
  - Else $d^{(0)}[u, v] = w(u, v)$ and $\pi^{(0)}[u, v] = u$

For $k = 1, 2 \ldots n$ Do
  For each $u, v \in V$
    \[ d^{(k)}[u, v] = d^{(k-1)}[u, v] \]
    \[ \text{If } d^{(k)}[u, v] > d^{(k-1)}[u, k] + d^{(k-1)}[k, v] \text{ Then Begin} \]
      \[ d^{(k)}[u, v] = d^{(k-1)}[u, k] + d^{(k-1)}[k, v] \]
      \[ \pi^{(k-1)}[u, v] = \pi^{(k-1)}[k, v] \]
    \[ \text{End;} \]
Report $d^{(n)}[\cdot, \cdot], \pi[\cdot, \cdot]$

4.5 Matrix Multiplications

We define a sequence of matrices $D^{(1)}, D^{(2)}, \ldots D^{(n-1)}$, where $(D^{(k)})_{ij}$ equal the shortest path that contains at most $k$ edges and connect $i$ to $j$. Similarly we define the matrices $\Pi^{(1)}, \Pi^{(2)}, \ldots \Pi^{(n-1)}$, where $(\Pi^{(k)})_{ij}$ is $x$ if and only $x$ is the predecessor vertex to $j$ in the shortest path from $i$ to $j$ that contains at most $k$ edges. Clearly, $(D^{(n-1)})_{ij}$ is just the length of the shortest path from $i$ to $j$. For obtaining an efficient algorithm, we do not compute all these matrices. Instead we compute $D^{(1)}, D^{(2)}, D^{(4)}, D^{(8)} \ldots D^{(t)}, D^{(2t)}$, where $t = \lceil \log_2 n \rceil$. Let $d^{(p)}[i, j]$ denote $(D^{(p)})_{ij}$ and let $\pi^{(p)}[i, j]$ denote $(\Pi^{(p)})_{ij}$. We compute this matrix using a process which is very similar to matrix multiplication. The running time is $O(n^3 \log n)$. 

Programming Project  
April 22, 1998
Matrix Multiplication

Initialization: For each \( u, v \in V \),

\[
\begin{align*}
\text{If} & \quad (u, v) \notin E \\
\text{Then} & \quad d^{(1)}[u, v] = \infty \quad \text{and} \quad \pi^{(1)}[u, v] = \text{NULL} \\
\text{Else} & \quad d^{(1)}[u, v] = w(u, v) \quad \text{and} \quad \pi^{(1)}[u, v] = u
\end{align*}
\]

For \( k = 2, 4, 8 \ldots \) Do

/* Multiplication Operation */

For each \( u, v \in V \)

For \( p = 1, 2 \ldots n \) Do

\[
\begin{align*}
&\quad d^{(2k)}[u, v] = d^{(k)}[u, v] \\
&\quad \pi^{(2k)}[u, v] = \pi^{(k)}[u, v] \\
\text{If} & \quad d^{(2k)}[u, v] > d^{(k)}[u, p] + d^{(k)}[p, v] \\
\text{Then} & \quad \text{Begin} \\
&\quad d^{(2k)}[u, v] = d^{(k)}[u, p] + d^{(k)}[p, v] \\
&\quad \pi^{(2k)}[u, v] = \pi^{(k)}[p, v] \\
\text{End} ;
\end{align*}
\]

If \( d^{(2t)}[i, \cdot] \neq d^{(2t)}[\cdot, \cdot] \\
\text{Then} \quad \text{report that there are negative-length cycle.}

Else

Report \( d^{(t)}[i, \cdot], \pi^{(t)}[i, \cdot] \)

5 Dialog of plgraph with the user

The program asks the user a series of questions. It asks for an input file name. If the user does not specify a file, (and just pressed return) the default input file is “graph.inp”. plgraph should check the existence of the file and its correctness.

Next plgraph asks whether the uses wishes to solve the single-source problem, or the all-pairs problem. If the user wishes to solve the single-source problem, plgraph asks her to specify a source vertex (the one playing s rule). If the user specify the single source problem, and all edge-weights are non-negative, she is asked to specify which algorithm to use, and then which data structure (if relevant to the algorithm). In Case of an error during the dialog, the program should stop and report which error. If the graph contains negative-length cycle, plgraph should stop and report about it.

Otherwise the program should report for the output, as follows: The user is asked to specify a name of an output file. If she replies by return, then the default is “graph.out”. The output might (and preferbly would) contain comment lines, which are lines that begin with an # sign in the first character. The automatic checking of the program will not refer to these lines. They contains description of the output, (whatever you find helpful to make the output more readable). These lines contains all the information of the output of the program. In addition, the program should output the data it computed, as follows:

In the case of all-pairs shortest paths, plgraph should print \( n \times n \) lines, each containing four numbers: \( \bar{s}, \bar{j}, d^{(n)}[\bar{s}, \bar{j}], \pi[\bar{s}, \bar{j}] \), where \( d[\cdot] \) and \( \pi[\cdot] \) are as described in Section 4.4. The lines should be sorted lexicographically. These lines of course, do not contain the # sign (or any other symbol). If the distance \( d[s, i] \) is
$\infty$, the distance $d[s,i]$ is the integer MaxLong.

In the case of the single source problem, the non-remarks lines, are as follows; There should be $n$ lines, one for each vertex $i$ between 1 to $n$. Each line contains three numbers $i, d[s, i], \pi[s, i]$, where $d[s, i]$ is the distance from $s$ to $i$. The lines should be sorted by the number of the vertex.

6 Guidelines

- Submission dates

  Files and Printouts: 16.98 at 23:00 on libra.
  Any change you do make in the files later than that date will cost you points in your grade.

- More accurate instruction might appear during the semester, so keep yourself informed by looking at the homepage of the course.

- You may assume that no syntactic errors appear in the input. However, there might be tabs or spaces inside it.

- You are requested to open a directory named "$\sim/software2/proj" (subdirectory 'software2/proj' under your home directory). Make sure that the files containing your project reside EXACTLY where you were asked to put them and with the exact names, and that the project you were requested to write have the exact interface as requested. If you do not do all that, then your grade will be severely affected. Remember that the installation of your work is a part of it.

  Set the permission on your source files and the directories in the path leading to them, to be 705. That is,

  chmod 755 "$\sim/software2/proj"

- The project MUST be submitted both on the computer (compilable by g++ on libra, containing its documentation within the files) and on paper (printout of the files). The login name should be printed/written on the printouts.

- In addition to your source files, you are required to supply us with a makefile that compiles your project (with g++ -Wall). The make file should be "$\sim/software2/proj/Makefile". Running make should create the executable of your project named plgraph.

- Your project must be compiled under g++ with full warning mode. Moreover, your source MUST compile with NO warnings whatsoever.

- Add your name, id, and login in a remark in the top of each source file.

- Write clear and readable code. Use assertions to ensure your project correctness. In particular, your program should check (or assert) for errors returned by all the calls to library functions in your program, and respond accordingly. Moreover, give meaningful names to variables, constants, and functions in your project.

- Comment your project. But use your common sense! You will lose points for over commenting.

- We will implement the project. You might play and run and test it. Your implementation should have the same input/output format.

Programing Project

April 22, 1998
• The exercise will not be checked if the above is not fulfilled. Furthermore, to get a full grade, your program should output EQUIVALENT results to these of the demo program.

The demo program is found under: ~/systut/pl98b/proj/. Input examples are found under ~/systut/pl98b/proj/inputs/.

• Any changes or updates concerning the exercise will appear in the FAQ. Also look at the homepage of the course at the same account.

• The time bounds of the algorithm are what we expect to find. So do let your implementation to Dijkstra algorithm run in time larger than $O(n \log n + m \cdot T(m))$ where $T(m)$ is the time required to perform a "Decrease" operation etc.

• Bonus points will be given for an especially efficient code and for especially readable code. Points will be subtracted for especially inefficient or ugly code.

Good Luck and have fun!