Jay Dey cs100vaj
Joshua Yuen cs100vbc

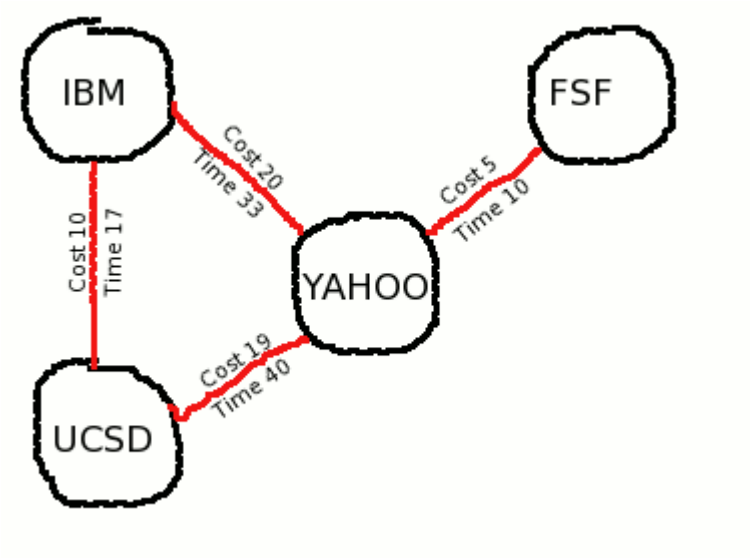
P4: NETPLAN DOCUMENTATION

PROGRAM DESCRIPTION

This program examines financial cost and transit time cost of long-distance computer networks, through the use of Graph data structures. A graph is made of vertices connected by edges. By listing links with corresponding costs and times in a text file (which will be passed in as a command line argument), like so:

```
ibm.com yahoo.com    20    33
yahoo.com    ucsd.edu 19    40
ucsd.edu ibm.com 10    17
yahoo.com    fsf.org  5     10
```

we build a network of links, populating a graph with vertices and edges. In each line in the input file, the two computer names are vertices, linked by an edge with cost (the first number) and transit time (the second number) respectively. The resulting graph can be visualized as so:



We can call operations on this graph, such as calculating the total cost of the network, finding the minimum spanning tree of the graph, and running Dijkstra's shortest path method to find the transit time cost between computers.

Compilation with the makefile will create an executable netplan, which can be run with the following command:

```
$ ./netplan infile
```

The output will be six numbers, each on a separate line:

1. The total cost of building all of the possible network links.
2. The total cost of building the cheapest network that will permit packets to travel from any computer to any other.
3. The amount of money saved in building that minimum-cost network instead of the all-possible-links network.
4. The total transit time to send a packet between all pairs of computers if all possible network links were built.
5. The total transit time it would take to send a packet between all pairs of computers in the minimum-cost network.
6. The increase in the "total time" required for packet travel in the cheap network (compared to having built all possible links).

APPLICATION PROGRAMMING INTERFACE
Below you will find a brief API of the participating data structures - Vertex, Edge, and Graph. This will give you a list and description of the structure's attributes and methods.

VERTEX

private:

```
string name; /* name of current vertex */
Vertex * pre; /* pointer to preceding index */
bool visited; /* flag to tell if vertex has been visited */
vector<Edge> conList; /* list of edges connected to this vertex */
```

public:

```
int dist; /* used in shortest path sum in dijkstra algorithm */

* constructor */
Vertex( string id );

/* destructor
 * calls clear on edgeList, destruction of vertex in Graph
 */
~Vertex();

/* self-explanatory helper methods for access to private attributes */
bool isVisited();
void setVisited( bool status );
string getName();
Vertex * getPre();
void setPre( Vertex * newPre );
vector<Edge> getEdges();

/* output edgeList */
void printList();

/* check if edge exists between this and v */
bool existEdge( Vertex * v );

/* add vertex with this as originator, connect edge between this and adj, and add to conList of both vertices
 * edge will have provided cost and time
 */
void addAdjVertex( Vertex * adj, int cost, int time );
```

EDGE

private:

```
Vertex * start; /* start point of edge */
Vertex * end; /* end point of edge */
int cost; /* cost of edge (specified in infile) */
int time; /* timecost of edge( specified in infile) */
```

public:

```
/* constructor */
Edge( Vertex * st, Vertex * en, int inCost, inTime )

/* self explanatory helper methods for access to private attributes */
Vertex * getStart();
Vertex * getEnd();
int getCost();
int getTime();
```

GRAPH

```
private:
  vector<Vertex *> vList; /* vector containing list of all vertices in Graph */
  bool cycle; /* flag for cycle detection */

public:
  /* Constructor */
  Graph();

  /* Destructor */
  ~Graph();

  /* find and return min spanning tree */
  Graph * MST();

  /* run dijkstra's algorithm on Graph
   * return total time cost
   */
  int dijkstra();

  /* output Graph by listing all vertices in Graph and corresponding connected edges */
  void displayGraph();

  /* total cost of entire Graph */
  int totalCost();

  /* total timecost of entire Graph */
  int totalTime();

  /* clears visited status of each vertex in Graph */
  void reset();

  /* adds vertex to vList */
  void addVertex( Vertex * ve );

  /* search graph for Vertex by name
   * return vertex if found, else return NULL
   */
  Vertex* findVertex( string name );
```