```
----------------------------------------------------------------
graph_floydwarshall.cpp: all-pairs shortest path (vers 1)
                         transitive closure (vers 2)
----------------------------------------------------------------

#include <...>
using namespace std;

template <typename Tkey, typename Twgt>
class graph {

  see graph_wgt.cpp for basic definitions

  public:
    void allpairs_shortestpath();
    void show_route(const string &, const string &);

    void transitive_closure();

  private:
    vector< vector<Twgt> > vdist;
    vector< vector<int> > vlink;

    vector< vector<char> > vreach;
};

int main(int argc, char *argv[]) {
  graph<string,int> G;
  G.read(argv[...]);

  if (vers1) {
    G.allpairs_shortestpath();

    string source, sink;

    while (1) {
      cout << "route> ";
      cin >> source >> sink;
      if (cin.eof()) break;

      G.show_route(source, sink);
    }
  }

  else
  if (vers2)
    G.transitive_closure();
}
```

```
template <typename Tkey, typename Twgt>
void graph<Tkey,Twgt>::allpairs_shortestpath() {
  int N = (int)V.size();
  Twgt infinity = numeric_limits<Twgt>::max()/2;

  vdist.assign(N, vector<Twgt>(N, infinity));
  vlink.assign(N, vector<int>(N, -1));

  for (int i=0; i<N; i++) {
    for (int k=0; k<(int)E[i].size(); k++) {
      int j = E[i][k];
      Twgt wij = W[i][k];

      if (i != j) {
        vdist[i][j] = wij;
        vlink[i][j] = i;
      }
    }

    vdist[i][i] = 0;
  }

  for (int k=0; k<N; k++) {
    for (int i=0; i<N; i++) {
      for (int j=0; j<N; j++) {
        if (vdist[i][j] > vdist[i][k] + vdist[k][j]) {
          vdist[i][j] = vdist[i][k] + vdist[k][j];
          vlink[i][j] = vlink[k][j];
        }
      }
    }
  }
}
```

```
----------------------------------------------------------------
Hint: The Floyd-Warshall algorithm recursively checks to see if
the distance from i to j can be lowered by going thru k. Using
dynamic programming, this is implemented using iteration and a
cost matrix. In order to more easily extract the correponding
routes, a link matrix is used to keep track of the lowest cost
paths.

Hint: Each row in the cost matrix holds the distances from that
vertex to all other vertices.

Hint: Each row in the link matrix holds the information needed
to extract the source-to-sink route.
----------------------------------------------------------------
```

```
template <typename Tkey, typename Twgt>
void graph<Tkey,Twgt>::show_route(
  string &source_key, string &sink_key) {

  if (key_map.find(source_key) == key_map.end()) {
    cerr << "error: " << source_key << " not found!\n";
    exit(1);
  }

  if (key_map.find(sink_key) == key_map.end()) {
    cerr << "error: " << sink_key << " not found!\n";
    exit(1);
  }

  int source = key_map[source_key];
  int sink = key_map[sink_key];

  if (vlink[source][sink] == -1) {
    cout << "no route found\n";
    return;
  }

  stack<int> S;

  for (int j=sink; j != source; j=vlink[source][j])
    S.push(j);
  S.push(source);

  while (!S.empty()) {
    int i=S.top();
    S.pop();
    cout << V[i] << " "
         << vdist[source][i] << "\n";
  }
}
```

Hint: A source-to-sink route is extracted by starting at the sink and then repeatedly looking up predecessors until the source is reached. As usual, a stack is used to reverse the order when printing the result.

```
template <typename Tkey, typename Twgt>
void graph<Tkey,Twgt>::transitive_closure() {
  int N = (int)V.size();

  vreach.assign(N, vector<char>(N, 0));

  for (int i=0; i<N; i++) {
    for (int k=0; k<(int)E[i].size(); k++) {
      int j = E[i][k];
      vreach[i][j] = 1;
    }

    vreach[i][i] = 0;
  }

  for (int k=0; k<N; k++) {
    for (int i=0; i<N; i++) {
      for (int j=0; j<N; j++) {
        vreach[i][j] |= vreach[i][k] && vreach[k][j];
      }
    }
  }

  int w = max_width_vertex_label(V);

  for (int i=0; i<N; i++) {
    cout << setw(w) << V[i];
    for (int j=0; j<N; j++)
      cout << setw(4) << (int)vreach[i][j];
    cout << "\n";
  }
}
```

Hint: The transitive closure of a graph produces indicator matrix that says which vertices can be reached from one another: if path exists from i to k and another path exists from k to j, then path exists from i to j.

Hint: Edge weights are initialized to absence (0) or presence (1) of edge. The min operator is replaced by OR. The add operator is replaced by AND.

Hint: The indicator matrix produced can be analyzed to reveal (groups of) vertices that cannot be reached or left once reached.
