```
-----------------------------------------------------------------
graph_hascycle.cpp: detect if graph contains a cycle
-----------------------------------------------------------------

#include ...
using namespace std;

template <typename Tkey>
class graph {

  see graph1_usage.cpp for basic definitions

  public:
    void dfs_hascycle();

  private:
    bool dfs_hascycle(int, int p=-1);

    typedef enum { WHITE, GRAY, BLACK } vcolor_t;
    vector<vcolor_t> vcolor;
};

template <typename Tkey>
void graph<Tkey>::dfs_hascycle() {
  const char *wgb = "WGB";

  for (int i=0; i<(int)V.size(); i++) {
    vcolor.assign(V.size(), WHITE);

    bool status = dfs_hascycle(i);

    cout << setw(3) << right <<   i  << " "
         << V[i] << " = {";

    for (int j=0; j<(int)V.size(); j++)
      cout << " " << wgb[vcolor[j]];

    if (status) cout << " } <-- has cycle\n";
    else        cout << " }\n";
  }
}
```

```
template <typename Tkey>
bool graph<Tkey>::dfs_hascycle(int i, int p) {
  vcolor[i] = GRAY;

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

    if (j == p)
      continue;

    if (vcolor[j] == GRAY)
      return true;

    if (vcolor[j] == WHITE) {
      if (dfs_hascycle(j, i) == true)
        return true;
    }
  }

  vcolor[i] = BLACK;
  return false;
}

int main(int argc, char *argv[]) {
  if (argc != 2) {
    cerr << "usage: " << argv[0]
         << " graph.txt\n";
    return 0;
  }

  graph<string> G;

  G.read(argv[1]);
  G.dfs_hascycle();
}

-----------------------------------------------------------------
Hint: DFS cycle search is based on detecting GRAY vertex in path
Vertices start out WHITE, become GRAY when visited, and are made
BLACK when all adjacent (neighbor) vertices have been processed.
-----------------------------------------------------------------
```

```
-----------------------------------------------------------------
graph_distance.cpp: edge-distance from source to all vertices
-----------------------------------------------------------------

#include ...
using namespace std;

template <typename Tkey>
class graph {

  see graph1_usage.cpp for basic definitions

  public:
    void bfs_distance(Tkey &);

  private:
    void bfs_distance(int);

    vector<int> vdist;
};

template <typename Tkey>
void graph<Tkey>::bfs_distance(TKey &source_key) {
  if (key_map.find(source_key) == key_map.end()) {
    cerr << "error: " << source_key << " not found!\n";
    exit(1);
  }

  bfs_distance(key_map[source_key]);
}
```

```
-----------------------------------------------------------------
Hint: BFS distance calculation updates distance to vertex once.

Hint: BFS source-to-sink route calculation on the next page
stops when sink is reached. A link array maintains information
wrt which edges were traversed (update distance -> update link).
-----------------------------------------------------------------
```

```
template <typename Tkey>
void graph<Tkey>::bfs_distance(int source) {
  vdist.assign(V.size(), INT_MAX);
  vdist[source] = 0;

  queue<int> Q;
  Q.push(source);

  while (!Q.empty()) {
    int i=Q.front();
    Q.pop();

    for (int k=0; k<(int)E[i].size(); k++) {
      int j = E[i][k];
      if (vdist[j] == INT_MAX) {
        vdist[j] = vdist[i] + 1;
        Q.push(j);
      }
    }
  }

  for (int i=0; i<(int)V.size(); i++) {
    cout << setw(3) << left << i << " "
         << V[i] << ": ";
    if (vdist[i] == INT_MAX) cout << "na\n";
    else                     cout << vdist[i] << "\n";
  }
}

int main(int argc, char *argv[]) {
  if (argc != 3) {
    cerr << "usage: " << argv[0]
         << " source graph.txt\n";
    return 0;
  }

  string source = argv[1];

  graph<string> G;

  G.read(argv[2]);
  G.bfs_distance(source);
}
```

```
------------------------------------------------------------------
graph_route.cpp: determine bfs-route from source to sink
------------------------------------------------------------------

#include <...>
using namespace std;

template <typename Tkey>
class graph {

  see graph1_usage.cpp for basic definitions

  public:
    void bfs_route(Tkey &, Tkey &);

  private:
    void bfs_route(int, int);
    void show_route(int, int);

    vector<int> vdist;
    vector<int> vlink;
};

template <typename Tkey>
void graph<Tkey>::bfs_route(Tkey &source_key, Tkey &sink_key) {
  modified two-argument version of bfs_distance above
}

template <typename Tkey>
void graph<Tkey>::show_route(int source, int sink) {
  if (vdist[sink] == INT_MAX) {
    cout << "No path from\n";
    return;
  }

  stack<int> S;

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

  while (!S.empty()) {
    int i=S.top();
    S.pop();
    cout << setw(3) << i << " "
         << V[i] << " "
         << setw(3) << vdist[i] << "\n";
  }
}
```

```
template <typename Tkey>
void graph<Tkey>::bfs_route(int source, int sink) {
  vdist.assign(V.size(), INT_MAX);
  vlink.assign(V.size(), -1);

  vdist[source] = 0;
  vlink[source] = source;

  queue<int> Q;
  Q.push(source);

  while (!Q.empty()) {
    int i=Q.front();
    Q.pop();

    if (i==sink)
      break;

    for (int k=0; k<(int)E[i].size(); k++) {
      int j = E[i][k];
      if (vdist[j] == INT_MAX) {
        vdist[j] = vdist[i] + 1;
        vlink[j] = i;
        Q.push(j);
      }
    }
  }

  while (!Q.empty())
    Q.pop();

  show_route(source, sink);
}

int main(int argc, char *argv[]) {
  if (argc != 4) {
    cerr << "usage: " << argv[0]
         << " source sink graph.txt\n";
    return 0;
  }

  string source = argv[1];
  string sink = argv[2];

  graph<string> G;

  G.read(argv[3]);
  G.bfs_route(source, sink);
}
```