Dijkstra Warm-Up

Lab 13

Wednesday, May 1, 2013

CSC 062: Spring, 2013

This lab will be a warm-up for Assignment 13. You are to implement a simplified version of Dijkstra's algorithm that assumes all edge weights in the graph are 1.

Getting Started

Read Assignment 13. Make sure you understand the sorts of algorithms you will be expected to implement for the assignment.

Make a new directory for this lab. Copy either our version of `priorityqueue62` into your working directory from the Assignment 13 starter or copy your version. You will use this data structure in your simplified Dijkstra's algorithm.

Graph Representation

For Assignment 13, we will ask you to write several graph operations, all of which will assume a simplified graph representation. Your graph will be a C++ `map<int, list<int> >` (notice that this is very different from the inefficient representation seen in class). That is, it will map a vertex identifier (stored as an `int`), to a list of `int`s, representing the vertex identifiers of all adjacent nodes. Note that there are no edge weights represented in the graph. Edges that exist, then, are assumed to be of weight 1. Note also that the vertex identifiers do not need to be sequential. For example, it is possible, that a graph may consist of three nodes – 7, 15, 101.

Dijkstra's

We have provided you with an incomplete algorithm for your simplified version of Dijkstra's algorithm at the end of the lab write-up. You should use the lab period today to make sure you understand how to use the C++ STL objects properly, and to finish the implementation. Specifically, you will want to understand how to iterate through different data types.

Iterators

Iterators in C++ behave very much like pointers, but they are their own special type. In keeping with the C++ philosophy of objects as primitives, iterator types overload standard operators so that their appearance is very different from Java iterators, and much more like primitive objects.

Consider the following example:

```
map<string,int> stringCounts;

...

map<string,int>::iterator iter;

for( iter = stringCounts.begin(); iter != stringCounts.end(); iter++ ) {
  cout << "word: " << iter->first << ", count: " << iter->second << endl;
}
```

We declare an iterator of type `map<string,int>::iterator`. We then initialize the iterator by calling the maps `begin` method. This will return an iterator, which you can think of as a pointer to the first element of the map. The `for` loop iterates until the iterator is equal to the value returned by the `end` function. This will typically be `NULL` or `eof`, or whatever is appropriate for the iterator type. Notice that the `++` operator has been overloaded to return a pointer to the next item in the data type.

At each iteration of the loop, `iter` functions like a pointer pointing to an element. We can use the "`->`" operator to access the elements of the `map`. The logic of building the `map` is not shown, but the implication is that `iter->first` refers to a `string`, and `iter->second` refers to the count of that string in some context. Similarly, if we want to access that entire object pointed to by the iterator, we must dereference it: `(*iter)`.

Remember that in C++, objects may be qualified as `const`. In these cases, the iterators they return will be of `const_iterator` type. In our above example, if `stringCounts` had been declared as follows:

```
const map<string,int> stringCounts;
```

our iterator declaration would change to:

```
map<string,int>::const_iterator iter;
```

and it will mean that we cannot alter the components of `stringCounts`.

String processing

Once you finish your Dijkstra implementation, you will want to review some string processing techniques that will be useful for Assignment 13.

Search for the `istringstream` class documentation from our C++ reference link on the course web page. `istringstream`s allow us to do processing of strings without having to do character-level processing. Look at the constructor. You can create a new `istringstream` as follows:

```
string movie_line = "32:197,4;615,4;680,1;";
istringstream in(movie_line, istringstream::in);
```

The first parameter is the `string` we want to process and the second tells it that we're going to be reading from this `string`.

Now, look at the `operator>>` method of `istringstream`. Notice that there are *many* overloaded versions of this operator. How this benefits us is that depending on what is on the right hand side of the `>>` operator, the stream will read as many characters as possible that fit the type of the variable. For example,

```
int num;
in >> num;
```

would result in `num` containing "32". If we then did:

```
char c;
in >> c;
in >> num;
```

What are the values of `c` and `num`?

Once you're comfortable with this, write a method:

```
pair<int, list<pair<int,int> > > parse_line(string line)
```

That takes a line formatted like our movie review file and returns a `pair` consisting of the reviewer id and a list containing the pairs of movie id and movie review.

Like other streams we've seen, you can check for when the `istringstream` is at the end of the `string` using `in.eof()`.

What to submit

Write a simple test method that creates a small graph of the correct type and pass it your Dijkstra implementation to verify that it works as desired. Also write a simple test method that passes a few sample movie review inputs to your `parse_line` function. Verify that it is working properly by printing the values stored in the returned `pair`. You will need to iterate over the list that is stored as the `second` member of the pair to make sure it is correct.

Store your test code and function implementations in a file named "lastname-Lab13.cpp," or "lastname1-lastname2-Lab13.cpp" if you worked with a partner, where you replace *lastname* with your last name. Drag your file to the cs62 dropbox.

Dijskstra's algorithm from class

```
/*
 *  Simplified Dijkstra's single-source shortest path algorithm
 *
 *  Arguments: a starting vertex
 *             an unweighted graph presented as an adjacency map
 *             (assumes all edge weights are 1)
 *
 *  Result: a map of parents in a tree of shortest paths
 *
 *  Rett Bull
 *  April 28, 2009
 *      Modified -- Dave 4/23/2010
 *               -- Kevin 4/23/2012
 */
map<int,int> shortest_paths(int start, const map<int,list<int> > & graph) {
  map<int,int> parents;
  priorityqueue62 frontier;

  parents[start]=start;
  frontier.push(start, 0);

  while (!frontier.is_empty()) {

    int v = frontier.top_key();
    int p = frontier.top_priority();
    frontier.pop();

    for (/* each neighbor n of v */) {
      if (n == parents[v])
        ; // do nothing
      else if (/* n is not in the frontier and has not been visited */) {
        parents[n] = v;
        frontier.push(n, p + 1);
      }else if (p + 1 < frontier.get_priority(n)) {
        parents[n] = v;
        frontier.reduce_priority(n, p + 1);
      }

    }

  } // end while

  return parents;
}
```