**CS 280 - Programming Assignment Graph**

**Information**

* [Handout](handout.docx)
* ALGraph: [Text](ALGraph.h)
* An example driver program: [Text](driver-sample.cpp)
* To experiment with the [Graphviz](http://www.graphviz.org/) package on your own computer, you'll need to install it.
  + Here's another [GUI front-end](GVizEd.exe). It's more convenient than the one that comes with the Graphviz package. (It's still a work-in-progress, though.)
* Some sample output from the driver is provided below.
  + Output from the [TestDijkstra0(1)](outputs/output-D0-1.txt) driver function. [Graphviz file](graphviz_files/Dijkstra0-gviz.txt)  [Image (dot)](images/Dijkstra0-gviz.txt.png)
  + Output from the [TestDijkstra1(1)](outputs/output-D1-1.txt) driver function. [Graphviz file](graphviz_files/Dijkstra1-gviz.txt)  [Image (dot)](images/Dijkstra1-gviz.txt.png)
  + Output from the [TestDijkstra4(1)](outputs/output-D4-1.txt) driver function. [Graphviz file](graphviz_files/Dijkstra4-gviz.txt)  [Image (dot)](images/Dijkstra4-gviz.txt.png)
  + Output from the [TestDijkstra4a(1)](outputs/output-D4a-1.txt) driver function. [Graphviz file](graphviz_files/Dijkstra4a-gviz.txt)  [Image (dot)](images/Dijkstra4a-gviz.txt.png)
  + Output from the [TestDijkstra5(1)](outputs/output-D5-1.txt) driver function. [Graphviz file](graphviz_files/Dijkstra5-gviz.txt)  [Image (dot)](images/Dijkstra5-gviz.txt.png)
  + Output from the [TestDijkstra9(1)](output-D9-1.txt) driver function. [Graphviz file](graphviz_files/Dijkstra9-gviz.txt)  [Image (dot)](images/Dijkstra9-gviz.txt.png)
  + Output from the [TestDijkstra10(1)](outputs/output-D10-1.txt) driver function. [Graphviz file](graphviz_files/Dijkstra10-gviz.txt)  [Image (dot)](images/Dijkstra10-gviz.txt.png)
  + Output from the [TestDijkstra10(14)](outputs/output-D10-14.txt) driver function.
  + Output from the [TestAllDijkstra()](outputs/output-D10-all.txt) driver function.
  + Output from the [TestAllLists()](outputs/output-D10-all-lists.txt) driver function.
* Stress-test outputs from 3 calls to TestBig:
  + [TestBig(10, 100)](outputs/output-Big-10x100.txt)  [Graphviz file](graphviz_files/TestBig-10-100-gviz.txt)   [Image (dot)](images/TestBig10-100.txt.png)
  + [TestBig(30, 10)](outputs/output-Big-30x10.txt)  [Graphviz file](graphviz_files/TestBig-30-10-gviz.txt)   [Image (dot)](images/TestBig30-10.txt.png)
  + [TestBig(99, 2)](outputs/output-Big-99x2.txt)  [Graphviz file](graphviz_files/TestBig-99-2-gviz.txt)   [Image (dot)](images/TestBig99-2.txt.png)
  + [All 3 tests using g++](outputs/output-Big-10x100-30x10-99x2-gnu.txt)
  + [All 3 tests using Borland's bcc32](outputs/output-Big-10x100-30x10-99x2-bor.txt)
* Sample command lines:
* **g++** driver.cpp ALGraph.cpp -O -Wall -Wextra -pedantic -Wconversion
* **cl** driver.cpp ALGraph.cpp /EHa /Za /W4 /MTd /WX /D\_CRT\_SECURE\_NO\_DEPRECATE
* **bcc32** -v -vG -w -w-8091 -w-8092 -w-8026 driver.cpp ALGraph.cpp
* Note the ordering of the egdes in the adjacency list for Dijkstra4 below. The edges are first sorted by weight, and if there are duplicates, they are sorted by destination node ID.

**C++ and STL Information**

Using the STL will greatly simplify your tasks. However, if you've not used the STL much outside of CS225, you may run into some problems with your coding.

1. Creating a priority queue with the default behavior is trivial:

*// Create a PQ that uses the default std::less<> class.*

*// std::less uses operator< so whatever you place in the PQ*

*// must support operator< (the MyAdjInfo implements this).*

std::priority\_queue<MyAdjInfo> edges;

This priority queue will keep things sorted from smallest to largest.

1. Creating a priority queue that provides a different behavior requires slightly more effort:

*// Create a comparison functor to override the default behavior.*

*// We can use std::greater to get the correct behavior. MyAdjInfo*

*// also must implement operator> for this.*

std::greater<MyAdjInfo> cmp;

*// Create a PQ using the functor above. Since you are providing a value for*

*// the 3rd template parameter, you must provide all three parameters.*

*// You must also pass the comparison functor to the constructor.*

std::priority\_queue<MyAdjInfo, std::vector<MyAdjInfo>, std::greater<MyAdjInfo> > edges( cmp );

Now the queue will sort in the reverse order. You may need to use these priority queues to help implement the algorithms. It's quite possible that you do not need to change the behavior, depending on how you implement it.

**Example Diagrams for Dijkstra's Algorithm**

1. An undirected, weighted graph with 6 vertices and 10 edges: (Dijkstra1 in the driver)

|  |  |
| --- | --- |
| https://faculty.digipen.edu/~mmead/www/Courses/2013/fall/cs280/project6/Graphs-Dijkstra-1.gif | ALGraph g(6);  g.AddUEdge(1, 2, 8);  g.AddUEdge(1, 3, 16);  g.AddUEdge(1, 5, 13);  g.AddUEdge(2, 3, 7);  g.AddUEdge(2, 4, 17);  g.AddUEdge(2, 5, 11);  g.AddUEdge(2, 6, 10);  g.AddUEdge(3, 4, 5);  g.AddUEdge(4, 5, 14);  g.AddUEdge(4, 6, 6); |

**Output:**

Adjacency list:

-------------------------------

ID: [ 1] -- 8 --> [ 2] -- 13 --> [ 5] -- 16 --> [ 3]

ID: [ 2] -- 7 --> [ 3] -- 8 --> [ 1] -- 10 --> [ 6] -- 11 --> [ 5] -- 17 --> [ 4]

ID: [ 3] -- 5 --> [ 4] -- 7 --> [ 2] -- 16 --> [ 1]

ID: [ 4] -- 5 --> [ 3] -- 6 --> [ 6] -- 14 --> [ 5] -- 17 --> [ 2]

ID: [ 5] -- 11 --> [ 2] -- 13 --> [ 1] -- 14 --> [ 4]

ID: [ 6] -- 6 --> [ 4] -- 10 --> [ 2]

Cost to reach all nodes from node 1:

------------------------------------

Node: 1: Cost: 0 Path: 1

Node: 2: Cost: 8 Path: 1 2

Node: 3: Cost: 15 Path: 1 2 3

Node: 4: Cost: 20 Path: 1 2 3 4

Node: 5: Cost: 13 Path: 1 5

Node: 6: Cost: 18 Path: 1 2 6

1. A directed, weighted graph with 16 vertices and 33 edges: (Dijkstra4 in the driver).

Pay attention to the order of the edges in the adjacency list when there are multiple edges with the same weight.

|  |  |
| --- | --- |
| https://faculty.digipen.edu/~mmead/www/Courses/2013/fall/cs280/project6/Graphs-Dijkstra-4-1.gif | ALGraph g(16);  g.AddDEdge(1, 2, 1);  g.AddDEdge(1, 5, 3);  g.AddDEdge(2, 3, 2);  g.AddDEdge(2, 5, 1);  g.AddDEdge(3, 4, 3);  g.AddDEdge(3, 7, 5);  g.AddDEdge(4, 8, 2);  g.AddDEdge(5, 6, 3);  g.AddDEdge(5, 9, 2);  g.AddDEdge(5, 10, 1);  g.AddDEdge(6, 2, 6);  g.AddDEdge(6, 10, 1);  g.AddDEdge(7, 2, 2);  **g.AddDEdge(7, 8, 1);**  **g.AddDEdge(7, 6, 1);**  **g.AddDEdge(7, 10, 1);**  g.AddDEdge(8, 3, 1);  g.AddDEdge(9, 13, 4);  g.AddDEdge(9, 14, 5);  g.AddDEdge(10, 9, 2);  g.AddDEdge(10, 14, 1);  g.AddDEdge(11, 7, 3);  **g.AddDEdge(11, 10, 2);**  **g.AddDEdge(11, 12, 2);**  g.AddDEdge(12, 7, 2);  g.AddDEdge(12, 8, 3);  g.AddDEdge(12, 16, 1);  g.AddDEdge(14, 13, 2);  g.AddDEdge(14, 15, 1);  g.AddDEdge(15, 10, 5);  g.AddDEdge(15, 11, 2);  g.AddDEdge(16, 11, 3);  g.AddDEdge(16, 15, 2); |

**Output:**

Adjacency list:

-------------------------------

ID: [ 1] -- 1 --> [ 2] -- 3 --> [ 5]

ID: [ 2] -- 1 --> [ 5] -- 2 --> [ 3]

ID: [ 3] -- 3 --> [ 4] -- 5 --> [ 7]

ID: [ 4] -- 2 --> [ 8]

ID: [ 5] -- 1 --> [10] -- 2 --> [ 9] -- 3 --> [ 6]

ID: [ 6] -- 1 --> [10] -- 6 --> [ 2]

ID: [ 7] **-- 1 --> [ 6] -- 1 --> [ 8] -- 1 --> [10]** -- 2 --> [ 2]

ID: [ 8] -- 1 --> [ 3]

ID: [ 9] -- 4 --> [13] -- 5 --> [14]

ID: [10] -- 1 --> [14] -- 2 --> [ 9]

ID: [11] **-- 2 --> [10] -- 2 --> [12]** -- 3 --> [ 7]

ID: [12] -- 1 --> [16] -- 2 --> [ 7] -- 3 --> [ 8]

ID: [13]

ID: [14] -- 1 --> [15] -- 2 --> [13]

ID: [15] -- 2 --> [11] -- 5 --> [10]

ID: [16] -- 2 --> [15] -- 3 --> [11]

Cost to reach all nodes from node 1:

------------------------------------

Node: 1: Cost: 0 Path: 1

Node: 2: Cost: 1 Path: 1 2

Node: 3: Cost: 3 Path: 1 2 3

Node: 4: Cost: 6 Path: 1 2 3 4

Node: 5: Cost: 2 Path: 1 2 5

Node: 6: Cost: 5 Path: 1 2 5 6

Node: 7: Cost: 8 Path: 1 2 3 7

Node: 8: Cost: 8 Path: 1 2 3 4 8

Node: 9: Cost: 4 Path: 1 2 5 9

Node: 10: Cost: 3 Path: 1 2 5 10

Node: 11: Cost: 7 Path: 1 2 5 10 14 15 11

Node: 12: Cost: 9 Path: 1 2 5 10 14 15 11 12

Node: 13: Cost: 6 Path: 1 2 5 10 14 13

Node: 14: Cost: 4 Path: 1 2 5 10 14

Node: 15: Cost: 5 Path: 1 2 5 10 14 15

Node: 16: Cost: 10 Path: 1 2 5 10 14 15 11 12 16