CSE221 Data Structures (Fall 2014)
Instructor: Prof. Won-Ki Jeong
Due date: Nov 30, 2014, 11:59 pm.

Assignment 5: Dijkstra Shortest Path Computation using Heap

In this assignment, you will implement O((n + e) log n) Dijkstra shortest path method using a heap data structure (n is # of vertices, e is # of edges). The input is a list of weighted edges of a graph and two end vertices. The output is the shortest path connecting the two vertices. Two main classes you need to implement are `MinHeap` and `Dijkstra`.

1. MinHeap data structure (30 pts)

MinHeap data structure is a complete binary tree satisfying MinHeap property, i.e., a parent key value is no larger than its children's. In this assignment, you need to implement MinHeap using array. You will use MinHeap for shortest path computation, so the heap element consists of the vertex index and the distance value. For MinHeap, the vertex distance will be used as a key value. An example of heap element class is as follows:

```
class heapElem {
public:
     unsigned int idx; // vertex index
     double dist;  // current minimum distance
};
```

For MinHeap class, you can freely use the code shown in the textbook p.281~286, but make sure you implement at least the functions listed below:

`void Push(const heapElem& e)`
Insert a new heap element e into the MinHeap

`const heapElem & Top()`
Get the minimum element in the MinHeap

`void Pop()`
Delete the minimum element in the MeanHeap

`void Modify(const heapElem& e)`

Modify dist value of the heap element and restructure the heap

`Modify()` is required because we only need to update adjacent vertices when a new vertex is included in the set S (set of vertices whose minimum path is computed). To efficiently implement `Modify()`, you need a constant time access of internal array index for a given heapElem. That means, since heapElem location may change when the heap is restructured, for a given vertex index v, its location in the array must be found in constant time. For this, you can define another array that maps the vertex index to heap array index, and whenever the heap is updated you must update this map accordingly.

2. Dijkstra shortest path method (60 pts)

You will implement a class Dijkstra to find the one-to-one shortest path in the graph using a min heap-based Dijkstra algorithm. The basic algorithm is given in the textbook p.361~364, but there are two differences:

1) You need to find one-to-one shortest path, and
2) You must use Heap to make the algorithm $O((n + e) \log n)$ instead of $O(n^2)$.

You may assume that the input edge weights are nonnegative. Minimum requirement of this class is as follows:

Graph I/O:
You need to implement I/O function to read in the directed graph and build an adjacency list internally. The input file is an ASCII text and the format is as follows:

```
10 20  : total number of vertices (10) & edges (20)
0 1 15 : <0,1>, weight is 15
...
```

You can assume that the vertex index starts from 0 (that means, if there are n vertices, then the indices are 0,1,2, ... , n-1)

class Dijkstra:
For Dijkstra class, you need to implement at least the following functions:

```
void ReadGraph(const char* file)
```

Read in the input graph text file and build the adjacency list of the input graph.

`double FindPath(const long int v0, const long int v1)`
Find the shortest path from v0 to v1, and print out the path. For example, if 0->2->3->9->12 is the shortest path between vertex 0 and 12, then you must print out the result by listing the vertices with commas, such as: `0,2,3,9,12` and returns the path length. If the path does not exist between v0 and v1, then you must print out "No path" and returns -1.

This is the pseudo-code for the Dijkstra algorithm using the heap:

```
for all v in the graph
{
    dist[v] = INF;
    s[v] = false;
}

set dist[v0] = 0 and push it to heap

while heap is not empty
{
    v = pop the top element from the heap;
    s[v] = true;
    if v == v1 break;
    for all adjacent w of v
    {
        if(!s[w] && dist[w] > dist[v] + length[v][w])
        {
             dist[w] = dist[v] + length[v][w]
             store v as the previous vertex of w;
             if w is not in the heap, push w to heap;
             else update heap for w;
        }
    }
}
```

`Backtrace from v1 to v0 using previous vertices and print out each in reverse order (from v0 to v1)`

Note that to be able to trace the minimum cost path, you need to store previous vertex index per each vertex. That means, when you find the shortest path for the vertex v, then you need to store the vertex index coming to v.

Once you reach the vertex v1, then you can backtrace the previous index to reconstruct the path.

`main.cpp` is the main file to test your code. This file contains testing code to evaluate the correctness of your implementation. If your implementation is correct, the result should be as follows:

```
Shortest path between 0 and 6 : 0,1,4,6
Distance : 8
```

Large graph datasets are also included (graph-medium.txt and graph-large.txt). Your program should be able to load and run on those large graphs too. Here's some paths found in those graphs.

graph-medium.txt

```
Shortest path between 0 and 9201 :
0,3,23,112,358,678,2077,4373,6486,7864,8669,9201
Distance : 112
```

graph-large.txt

```
Shortest path between 0 and 36422 :
0,1,392,1130,115474,222,562,36422
Distance : 45
```

3. Compile and submit

You must log in uni06~10.unist.ac.kr for coding and submitting the assignment. You can compile the code using the included Makefile. You can simply `make` and then the code will be compiled. The output executable name is `assign_5`.

You are required to submit `dijkstra.cpp` and `minheap.cpp` and the report describing your work (worth 10 pts). Once you are ready to submit the code, use the `dssubmit` script as follows:

```
> dssubmit assign5 assign5.zip
```

Good luck and have fun!