# Programming Assignment 2

## 1. Data Structure

I construct a class "DijkstraMap" to model this graph as below

// typo: all DijstraMap should be Dij"k"straMap
  
class DijstraMap
  
{
  
 public:
  
 DijstraMap ();
  
 ~DijstraMap ();
  
   
 // get function ...
  
 // reset function ...
  
 // set function ...
  
 // print function ...
  
   
 private:
  
 int \*\*parent; // record retrace path
  
 double \*\*distance; // record the distance from source
  
 bool \*\*hConnected, \*\*vConnected, \*\*viaConnected; // record connection
  
 short \*\*vCapacity, \*\*hCapacity, \*\*vLoad, \*\*hLoad, \*\*prevLoad, \*\*prehLoad, \*\*vof, \*\*hof; // record loading
  
 int preCount; // record how many rip-up and re-reoute is preformed
  
};

A class "router" to preform Dijkstra's shortest path algorithm

class Router
  
{
  
public:
  
 Router();
  
 ~Router();
  
   
 void routeAll( ofstream& \_of );
  
 bool preRoute();
  
   
private:
  
 DijstraMap \*dMap;
  
 time\_t startTime, preRouteStopTime; // record the timing limit
  
 multimap<int, Net\*> orderedNets; // reocord nets in decreasing order base on their Manhattan distance of pins in subnets
  
   
 // Dijstra function ...
  
 // Ordering function ...
  
 // util function ...
  
 // print function ...
  
   
};

## 2. Algorithm

To find the path from start to end, I use Dijkstra shortest-path algorithm, which described as below

// weight function = 2^(load/capacity)
  
double DijstraMap::get\_weight(int h, int w, int direction)
  
{
  
 int load = get\_load(h, w, direction);
  
 double ratio = (load)\*1.0 / capacity\*1.0;
  
 double weight = (pow(2, ratio));
  
 return weight;
  
}
  
   
// Dijkstra
  
void router::Dijkstra(int \_x, int \_y, int \_end\_x, int \_end\_y, ofstream& output)
  
{
  
 minHeap \_distanceHeap;
  
 Vertice\* \_v;
  
 short \_targetX = \_sn->GetTargetPinGx(), \_targetY = \_sn->GetTargetPinGy();
  
   
 // initialize single source
  
 dMap->refreshDistance();
  
 \_v = new Vertice( \_sn->GetSourcePinGx(), \_sn->GetSourcePinGy(), 0 );
  
 dMap->setDistance( \_v );
  
 dMap->setParent( \_v, NONE );
  
 relax( \_v , \_distanceHeap );
  
 delete \_v;
  
   
 while(true){
  
 \_v = \_distanceHeap.top();
  
 \_distanceHeap.pop();
  
   
 if ( \_v->x == \_targetX && \_v->y == \_targetY ){
  
 delete \_v;
  
 while ( !\_distanceHeap.empty() )
  
 {
  
 \_v = \_distanceHeap.top();
  
 \_distanceHeap.pop();
  
 delete \_v;
  
 }
  
 break;
  
 }
  
   
 if ( !dMap->isMinDistance( \_v ) ) { // old vertice(substitute for Decrease-Key)
  
 delete \_v;
  
 continue;
  
 }
  
   
 relax( \_v, \_distanceHeap );
  
 delete \_v;
  
 }
  
}
  
   
void Router::relax ( Vertice\* \_p, minHeap& \_mheap )
  
{
  
 double \_newDistance;
  
 Vertice\* \_toVertice;
  
   
 for(int d = UP; d < NONE; ++d)
  
 {
  
 \_toVertice = toVertice( \_p, d );
  
 if ( \_toVertice != NULL ) {
  
 \_newDistance = \_p->distance + getWeight( \_p, \_toVertice );
  
 if ( \_newDistance < dMap->getDistance( \_toVertice ) )
  
 {
  
 dMap->setParent( \_toVertice, d );
  
 update( \_p, \_toVertice, \_newDistance, \_mheap);
  
 }
  
 else delete \_toVertice;
  
 }
  
 }
  
}

The only concept different from the algorithm discussed in class is that because I'm not using the Fibonocci heap, and the priority\_queue in standard library in standard library doesn't have "decrease-key" feature, I substitute it by push a new vertice to the heap whenever an edge is relaxed. Then, we can still choose the "closest" vertice to relax. When the heap pop out a vertice whose distance is further than distance stored in Modle, we can tell that this is a vertice whose key is actually "decreased". Hence, we just ignore it, and the whole algorithm is like the original Dijkstra shortest-path algorithm with "Decrease-Key".

I've tried some optimization to get a better performance. Eventually, I find out that sort the start-end vertice pair with their Manhatten distance in increasing order can get a better result. I also try rip-up and re-route, it can improve the solution quality tremendously.

## 3. Discussion

### a. Adjacency list v.s. Adjacency matrix ?

When modeling a graph, we usually use one of the above. However, I found them both not suitable for this problem because the arrangement of vertice and edge is very regular and all edges are bidirectional. Hence, I store all information in hxw matrix(not adjacency matrix which is hxw x hxw).

### b. Rip-up and Re-Route

My method is simply rip-up all the nets and reroute based on previous loading, i.e: load = previous load + load in this iteration. It can get a better result compare to run 1 iteration of Dijkstra only. However, the "previous load" term can be modified to get a even better solution(I heard from other classmate in PD), but I fail to test this method before deadline.