Programming homework 3

Wan-Cyuan Fan

- Data structure

```
struct tile{
   float d;
   pair<int,int> coord_now;
   pair<int,int> coord_from;
   bool visit;
};

struct coor_pair{
   pair<int,int> t1;
   pair<int,int> t2;
};
```

```
//graph vari.
vector<vector<tile>> grid;   //store the vertex
vector<vector<int>> edge_y;   //store the vertical edge weight
vector<vector<int>> edge_x;   //store the horizontal edge weight
//parameter vari.
int capacity;
int netNum;
int count = 0;
int row, col;
//Dijkstra's Alogrithm parameter
vector<tile*> Q;
//path storage
vector<int> gr_pair;
vector<coor_pair> gr_path;
//file
fstream myfile;
```

利用sturcture儲存tile,coor_pair兩個基本的元素：

1. tile代表每一個vertex，在做Dijkstra's Alg時，需要紀錄coor_now目前vertex座標位置，coord_from前一個vertex的座標位置，d為vertex目前累積的最小path weight。
2. coor_pair則是最後用來儲存，整條可行的path的基本元素，包含t1,t2，表示這兩個vertex有path。

利用vertex儲存整個graph,edge,edge weight等資訊兩個基本的元素：

1. grid：存取整個graph的vertex資訊。
2. edge_x,edge_y：存取edge的weight，分成水平和垂直方便讀值。
3. Q：為Dijkstra's alg中的，Q值。
4. Gr_pair：用來讀取每一次的start,end point以接著做接下來的Dijkstra's alg
5. Gr_path：儲存一條start,end point連線line的vertex配對。

- Algorithm

  Part 1: setting

```
cout << "@ Start routing data..." << endl;
cout << "@ Start setting data..." << endl;
router_graph.set_row_col(parser.gNumHTiles(),parser.gNumVTiles());
router_graph.set_capacity(parser.gCapacity());
router_graph.set_netNum(parser.gNumNets());
cout << "@ Setting data successed..." << endl;
cout << "@ Start graph..." << endl;
router_graph.set_graph(parser.gNumHTiles(),parser.gNumVTiles());
cout << "@ Setting graph successed..." << endl;
router_graph.setfile(argv[2]);
```

1. 將所有parser的資料存到graph的存取空間中，用來後面製作graph。
2. set_graph：建造對應的vertex數量，以及將edge接連上去，並將edge的weight初始為0。

Part 2: routing

```
for (int idNet = 0; idNet < parser.gNumNets(); ++idNet){
   pair<int, int> posS = parser.gNetStart( idNet );
   pair<int, int> posE = parser.gNetEnd( idNet );
   router_graph.set_gr_pair(idNet,posS.first,posS.second,posE.first,posE.second);
   router_graph.Dijkstra();
   router_graph.store_path();
   router_graph.writefile();
}
```

1. 在main中，我們分gNumNets次(# of the nets)，將資料丟入graph中反覆進行Dijkstra’s alg。
2. 因此set_gr_pair用來存取每次對應的vertex net pair。
3. 然後進行Dijkstra演算法。(下面補充說明)。
4. 再將每次net連線完的line利用store_path存取到edge_x,edge_y中每次累加。
5. 最後writefile寫入.out中。
6. 如此不斷重複，計算可能的path。

Part 3: Dijkstra's Alg

```
void
graph::Dijkstra(){
   initialize_single_source(); // include Q = G.V
   while (!Q.empty()) {
      tile* u = Extract_Min(Q);
      u->visit = true;
      if (u->coord_now.first - 1 >= 0) {
         relax(&grid[u->coord_now.first][u->coord_now.second], &grid[u->coord_now.first-1][u->coord_now.second]);
      }

      if (u->coord_now.second + 1 < row) {
         relax(&grid[u->coord_now.first][u->coord_now.second], &grid[u->coord_now.first][u->coord_now.second+1]);
      }

      if (u->coord_now.second - 1 >= 0) {
         relax(&grid[u->coord_now.first][u->coord_now.second], &grid[u->coord_now.first][u->coord_now.second-1]);
      }

      if (u->coord_now.first + 1 < col) {
         relax(&grid[u->coord_now.first][u->coord_now.second], &grid[u->coord_now.first +1][u->coord_now.second]);
      }
   }

}
```

```
void
graph::initialize_single_source(){
   for (int i = 0; i < row; ++i) {
      for (int j = 0; j < col; ++j) {
         grid[i][j].d = INT_MAX;
         Q.push_back(&grid[i][j]);
         grid[i][j].coord_from = make_pair(NAN,NAN);
      }
   }
   grid[gr_pair[1]][gr_pair[2]].d = 0;
}

tile*
graph::Extract_Min(vector<tile*>){
   tile* min;
   std::sort(Q.begin(), Q.end(), comp_dis);
   min = Q.front();
   Q.erase(Q.begin());
   return min;
}

void
graph::relax(tile* u,tile* v){
   float weight = get_edge(u, v);
   if (v->d > u->d + weight) {
      v->d = u->d + weight;
      int x = u->coord_now.first;
      int y = u->coord_now.second;
      v->coord_from = make_pair(x,y);
   }
}
```

1. 在Dijkstra's Alg中，首先先將所有vertex初始化，即initialize_single_source()將vertex weight設成無限大，再將source point設定成0，另外將所有的vertex放到Q中。
2. 第二部分，我們需要不斷的找到Q中的最小值，然後做relax，將更小的vertex weight做代換。
3. Extract_Min()：使用sort先將Q由小到大排好，再直接取第一個拿到最小值。
4. Relax部分，weight是利用router.h中的，edge_cost()，將對應到的demand,capacity作轉換找到的。(轉換公式在下面討論)

- Discussion

計算演算法複雜度：

演算法的兩個部分，

part 1 :setting，是將不同的vertex,edge寫上初始值。

因此複雜度為O(V + E)

part 2 : Dijkstra's alg

1. initialize_single_source(): O(V)
2. 在while 迴圈中的Extract_Min(),Relax()，只要影響是Extract_Min()，因為Relax()基本上為周圍最多4的vertex相連*O(1)
3. Extract_Min()：使用sort()，因此平均為O(VlgV)。但這部分因為不管第幾次sort，Q在每一次relax的改變量最多4個，因此Q幾乎是sort好的，如果用insertion sort可能可以到達O(V)的複雜度。這裡我們使用O(VlgV)估計。
4. 因此total: $O(V+V^2lgV)$。
5. 因為基本上作業檢查主要為connection,overflow程度，故這邊不在最佳化。

效能考慮： (edge weight考慮)

1. weight比較討論：

Edge cost	$2^{(demand/capacity)}-1$				
size	4x4	5x5	10x10	20x20	60x60
Overflow	0	0	1	0	52277
wirelength	15	44	310	20256	319446
Error	0	0	0	0	0

Edge cost	$10^{(demand/capacity)}$-2* demand/capacity				
size	4x4	5x5	10x10	20x20	60x60
Overflow	0	0	2	0	50946
wirelength	15	36	258	19902	315684
Error	0	0	0	0	0

Edge cost	$20^{(demand/capacity)}$-1				
size	4x4	5x5	10x10	20x20	60x60
Overflow	0	0	0	0	52327
wirelength	15	44	316	20370	322616
Error	0	0	0	0	0

2. 我們可以發現， overflow,wirelength，如果我們將edge cost設的和edge demand比例越大，也就是demand增加，edge cost增加越快，則overflow雖然減少了，但是反而wirelength變得多了一些。反之則會增加overflow減少wirelength。以目標先考量overflow,後考量wirelength的前提，我們採用$10^{(demand/capacity)}$-2* demand/capacity當作cost較好。