# Graph

## Definition

## About Algorithm

### DijkstraAlgorithm

算法描述：

  给定开始节点，返回开始节点到图中所有节点的距离，要求：图是无向图，同时图中没有和为负值的环路的存在

经典实现：

```
// 定义一个set，用来存储固定的节点距离记录，不能再进行更新记录-locked;
set<GNode*> lock;
//定义一个map ，用来存储从开始节点到图中其他节点的距离记录-res;
map<GNode*,int> res;
//从res中寻找最短的距离记录minNode，并且这个节点不能被锁定，也就是这个节点不在locked中
while(minNode!=nullptr)
{
  //用距离记录最小的节点来更新没有被锁定节点的记录
  int preDist = res.find(minNode)->second; // 这是minNode节点到开始节点的距离
  //解锁minNode节点的所有边；
  //如果此边的end节点没有在res，就添加此end节点到res中，距离记录为：preDist+thisEdge.weight
  //如果此边的end节点在res中了，那就看更新距离是否能更小化这个距离记录，如果能，则更新，如果不能，就不更新
          for(auto & edge:minNode->edges)
        {
            // get the this node's edge
            // edge's end node is recodered in res

            if(res.find(edge->end)==res.end())
            {
                // no this node recoder,insert

                res.insert(make_pair(edge->end,edge->weight+preDist));  //
            }
            else
            {
                // all recoders need to update

                // this node not be locked, so need to update
                // int preDist = res.find(minNode)->second;
                if((res.find(edge->end)->second)>(edge->weight+preDist)) res.find(edge->end)->second=edge->weight;
            }
        }
  // 按照minNode节点更新了所有能更新的节点，minNode也就完成了使命，可以被锁定了，不能更新了
  lock.insert(minNode);
  // 按照同样的逻辑在res中找到下一个minNode继续
}

```

一种改写堆的优化：

思路：首先在上述经典实现过程中，每次需要在为被锁定的记录中找到最小的记录，并且更新未被锁定的所有的节点的记录，如果res使用小根堆的方式来实现，只能找到最小的记录，并不能实现更新的操作，所以可以将上述代码实现中用来负责记录的res，使用改写的堆来实现。

```
//改写的小根堆
// 需要的功能：  实现能弹出最小的记录，而且能按照最小记录更新，然后还能调整成小根堆

class MinHeap{
public:
    MinHeap()
    {
        heapSize=0;
        vector<GNode*> list(100);
        heap=list;
//        cout<<heap.size()<<endl;
    }
    void swap(int index1,int index2)
    {
        GNode * tmp=heap[index1];
        heap[index1]=heap[index2];
        heap[index2]=tmp;
    }

    void heapFiy(int index,int heapSize)
    {
        int l =index*2+1;
        while(l<heapSize)
        {
            int minest_index= (l+1)<heapSize && (distanceRecorder.find(heap[l+1])->second < distanceRecorder.find(heap[l])->second) ? l+1:l;
            minest_index = distanceRecorder.find(heap[minest_index])->second < distanceRecorder.find(heap[index])->second ? minest_index:index;
            // swap
            if(minest_index==index) break; // 不需要调整
            swap(index,minest_index);
            index=minest_index;
            l=2*index+1;
        }
    }

    GNode * pop()
    {
        // pop the smallest distance node recorder.
        GNode * p=heap[0];
        // heapfiy
        swap(0,heapSize-1); //bottom -> top
        heapFiy(0,--heapSize);
        //locked
        locked.insert(p);
        return p;
    };
    void addOrupdate(GNode * node,float distance)
    {
        // if locker?
        if(locked.find(node)==locked.end())
        {
            if(distanceRecorder.find(node)!=distanceRecorder.end())
            {
                // already in, using distance to update
                if(distanceRecorder.find(node)->second>distance)
                    distanceRecorder.find(node)->second= distance;
                // restore heap
                heapFiy(0,heapSize);

            }
            else{
                // no this recorder, register this node and distance info.
//                  cout<<node->value<<endl;
                distanceRecorder.insert(make_pair(node, distance));
                // add this node into heap and heapSize ++
                heapInsert(node, heapSize);
            }
        }
//
//          for(int i=0;i<heapSize;i++)
//          {
//              cout<<heap[i]->value<<",";
```

```
//
//        }
//        cout<<endl;

    }
    void heapInsert(GNode * node,int index)
    {

        heap[index]=node; //先插入，然后在从index向上调整
        // get the distance.
        while(distanceRecorder.find(heap[index])->second < distanceRecorder.find(heap[index/2])->second)
        {
            // less than root
            swap(index,index/2);
            index=index/2;

        }
        heapSize++;
    }
    bool isEmpty()
    {
        if(heapSize==0) return true;
        return false;
    }
    int  search(GNode* node)
    {
        return distanceRecorder.find(node)->second;
    }
    int heapsize()
    {
        return heapSize;
    }
private:
    vector<GNode*> heap;
    map<GNode*,float> distanceRecorder;
    set<GNode*> locked;
    int heapSize;

};
```

```
// dijkstra
void DijkstraAlgorithmV2(Graph *G ,GNode* beginNode,map<GNode*,float> & res)
```

```
{
    if(G->nodes.size()<1) return;
    //
    MinHeap * mheap=new MinHeap();
    // add begin node into small root heap
    mheap->addOrupdate(beginNode,0);



    while(!mheap->isEmpty())
    {
        GNode * minNode=mheap->pop();
        float  preDistance=mheap->search(minNode);
        cout<<minNode->value<<endl;
        for(auto &item: minNode->edges)
        {
            mheap->addOrupdate(item->end,item->weight+preDistance);
        }
        res.insert(make_pair(minNode,preDistance));

    }


}
```

test

```
    map<GNode*,float> res,res1;
    DijkstraAlgorithm(graph,graph->nodes.begin()->second,res);
    cout<<res.size()<<endl;
    for(auto & item:res)
    {
        cout<<item.first->value<<"--"<<item.second<<endl;
    }
    cout<<"--------"<<endl;
    DijkstraAlgorithmV2(graph,graph->nodes.begin()->second,res1);

    cout<<res1.size()<<endl;
    for(auto & item:res1)
    {
        cout<<item.first->value<<"--"<<item.second<<endl;
    }

/*
test graph
```

```
  1
a-----b
  \  5| \ 4
  2\  | 3 \
     c-------e
*/

/*output:
4
1--0
2--1
3--2
4--5
-------
4
1--0
2--1
3--2
4--5
*/

```