```
template <class TWeight> class GCGraph
{
public:
	GCGraph();
	GCGraph( unsigned int vtxCount, unsigned int edgeCount );
	~GCGraph();
	void create( unsigned int vtxCount, unsigned int edgeCount );
	int addVtx();
	void addEdges( int i, int j, TWeight w, TWeight revw );
	void addTermWeights( int i, TWeight sourceW, TWeight sinkW );
	TWeight maxFlow();
	bool inSourceSegment( int i );
private:
	class Vtx
	{
	public:
		Vtx *next; // initialized and used in maxFlow() only
		int parent;
		int first;
		int ts;
		int dist;
		TWeight weight;
		uchar t;
	};
	class Edge
	{
	public:
		int dst;
		int next;
		TWeight weight;
	};

	std::vector<Vtx> vtcs;
	std::vector<Edge> edges;
	TWeight flow;
};

template <class TWeight>
GCGraph<TWeight>::GCGraph()
{
	flow = 0;
```

```
}
template <class TWeight>
GCGraph<TWeight>::GCGraph( unsigned int vtxCount, unsigned int edgeCount )
{
	create( vtxCount, edgeCount );
}
template <class TWeight>
GCGraph<TWeight>::~GCGraph()
{
}
template <class TWeight>
void GCGraph<TWeight>::create( unsigned int vtxCount, unsigned int edgeCount )
{
	vtcs.reserve( vtxCount );
	edges.reserve( edgeCount + 2 );
	flow = 0;
}

template <class TWeight>
int GCGraph<TWeight>::addVtx()
{
	Vtx v;
	memset( &v, 0, sizeof(Vtx));
	vtcs.push_back(v);
	return (int)vtcs.size() - 1;
}

template <class TWeight>
void GCGraph<TWeight>::addEdges( int i, int j, TWeight w, TWeight revw )
{
	CV_Assert( i>=0 && i<(int)vtcs.size() );
	CV_Assert( j>=0 && j<(int)vtcs.size() );
	CV_Assert( w>=0 && revw>=0 );
	CV_Assert( i != j );

	if( !edges.size() )
		edges.resize( 2 );

	Edge fromI, toI;
	fromI.dst = j;
	fromI.next = vtcs[i].first;
```

```
    fromI.weight = w;
    vtcs[i].first = (int)edges.size();
    edges.push_back( fromI );

    toI.dst = i;
    toI.next = vtcs[j].first;
    toI.weight = revw;
    vtcs[j].first = (int)edges.size();
    edges.push_back( toI );
}

template <class TWeight>
void GCGraph<TWeight>::addTermWeights( int i, TWeight sourceW, TWeight sinkW )
{
    CV_Assert( i>=0 && i<(int)vtcs.size() );

    TWeight dw = vtcs[i].weight;
    if( dw > 0 )
        sourceW += dw;
    else
        sinkW -= dw;
    flow += (sourceW < sinkW) ? sourceW : sinkW;
    vtcs[i].weight = sourceW - sinkW;
}

template <class TWeight>
TWeight GCGraph<TWeight>::maxFlow()
{
    CV_Assert(!vtcs.empty());
    CV_Assert(!edges.empty());
    const int TERMINAL = -1, ORPHAN = -2;
    Vtx stub, *nilNode = &stub, *first = nilNode, *last = nilNode;
    int curr_ts = 0;
    stub.next = nilNode;
    Vtx *vtxPtr = &vtcs[0];
    Edge *edgePtr = &edges[0];

    std::vector<Vtx*> orphans;

    // initialize the active queue and the graph vertices
    for( int i = 0; i < (int)vtcs.size(); i++ )
```

```
{
        Vtx* v = vtxPtr + i;
        v->ts = 0;
        if( v->weight != 0 )
        {
                last = last->next = v;
                v->dist = 1;
                v->parent = TERMINAL;
                v->t = v->weight < 0;
        }
        else
                v->parent = 0;
}
first = first->next;
last->next = nilNode;
nilNode->next = 0;

// run the search-path -> augment-graph -> restore-trees loop
for(;;)
{
        Vtx* v, *u;
        int e0 = -1, ei = 0, ej = 0;
        TWeight minWeight, weight;
        uchar vt;

        // grow S & T search trees, find an edge connecting them
        while( first != nilNode )
        {
                v = first;
                if( v->parent )
                {
                        vt = v->t;
                        for( ei = v->first; ei != 0; ei = edgePtr[ei].next )
                        {
                                if( edgePtr[ei^vt].weight == 0 )
                                        continue;
                                u = vtxPtr+edgePtr[ei].dst;
                                if( !u->parent )
                                {
                                        u->t = vt;
                                        u->parent = ei ^ 1;
```

```
                        u->ts = v->ts;
                        u->dist = v->dist + 1;
                        if( !u->next )
                        {
                            u->next = nilNode;
                            last = last->next = u;
                        }
                        continue;
                    }

                    if( u->t != vt )
                    {
                        e0 = ei ^ vt;
                        break;
                    }

                    if( u->dist > v->dist+1 && u->ts <= v->ts )
                    {
                        // reassign the parent
                        u->parent = ei ^ 1;
                        u->ts = v->ts;
                        u->dist = v->dist + 1;
                    }
                }
                if( e0 > 0 )
                    break;
            }
            // exclude the vertex from the active list
            first = first->next;
            v->next = 0;
        }

        if( e0 <= 0 )
            break;

        // find the minimum edge weight along the path
        minWeight = edgePtr[e0].weight;
        CV_Assert( minWeight > 0 );
        // k = 1: source tree, k = 0: destination tree
        for( int k = 1; k >= 0; k-- )
        {
```

```
        for( v = vtxPtr+edgePtr[e0^k].dst;; v = vtxPtr+edgePtr[ei].dst )
        {
            if( (ei = v->parent) < 0 )
                break;
            weight = edgePtr[ei^k].weight;
            minWeight = MIN(minWeight, weight);
            CV_Assert( minWeight > 0 );
        }
        weight = fabs(v->weight);
        minWeight = MIN(minWeight, weight);
        CV_Assert( minWeight > 0 );
    }

    // modify weights of the edges along the path and collect orphans
    edgePtr[e0].weight -= minWeight;
    edgePtr[e0^1].weight += minWeight;
    flow += minWeight;

    // k = 1: source tree, k = 0: destination tree
    for( int k = 1; k >= 0; k-- )
    {
        for( v = vtxPtr+edgePtr[e0^k].dst;; v = vtxPtr+edgePtr[ei].dst )
        {
            if( (ei = v->parent) < 0 )
                break;
            edgePtr[ei^(k^1)].weight += minWeight;
            if( (edgePtr[ei^k].weight -= minWeight) == 0 )
            {
                orphans.push_back(v);
                v->parent = ORPHAN;
            }
        }

        v->weight = v->weight + minWeight*(1-k*2);
        if( v->weight == 0 )
        {
            orphans.push_back(v);
            v->parent = ORPHAN;
        }
    }
```

```
// restore the search trees by finding new parents for the orphans
curr_ts++;
while( !orphans.empty() )
{
	Vtx* v2 = orphans.back();
	orphans.pop_back();

	int d, minDist = INT_MAX;
	e0 = 0;
	vt = v2->t;

	for( ei = v2->first; ei != 0; ei = edgePtr[ei].next )
	{
		if( edgePtr[ei^(vt^1)].weight == 0 )
			continue;
		u = vtxPtr+edgePtr[ei].dst;
		if( u->t != vt || u->parent == 0 )
			continue;
		// compute the distance to the tree root
		for( d = 0;; )
		{
			if( u->ts == curr_ts )
			{
				d += u->dist;
				break;
			}
			ej = u->parent;
			d++;
			if( ej < 0 )
			{
				if( ej == ORPHAN )
					d = INT_MAX-1;
				else
				{
					u->ts = curr_ts;
					u->dist = 1;
				}
				break;
			}
			u = vtxPtr+edgePtr[ej].dst;
		}
```

```
                    // update the distance
                    if( ++d < INT_MAX )
                    {
                        if( d < minDist )
                        {
                            minDist = d;
                            e0 = ei;
                        }
                        for( u = vtxPtr+edgePtr[ei].dst; u->ts != curr_ts; u = vtxPtr+edgePtr[u->parent].dst )
                        {
                            u->ts = curr_ts;
                            u->dist = --d;
                        }
                    }
                }

                if( (v2->parent = e0) > 0 )
                {
                    v2->ts = curr_ts;
                    v2->dist = minDist;
                    continue;
                }

                /* no parent is found */
                v2->ts = 0;
                for( ei = v2->first; ei != 0; ei = edgePtr[ei].next )
                {
                    u = vtxPtr+edgePtr[ei].dst;
                    ej = u->parent;
                    if( u->t != vt || !ej )
                        continue;
                    if( edgePtr[ei^(vt^1)].weight && !u->next )
                    {
                        u->next = nilNode;
                        last = last->next = u;
                    }
                    if( ej > 0 && vtxPtr+edgePtr[ej].dst == v2 )
                    {
                        orphans.push_back(u);
```

```
                                u->parent = ORPHAN;
                            }
                        }
                    }
                }
                return flow;
}

template <class TWeight>
bool GCGraph<TWeight>::inSourceSegment( int i )
{
        CV_Assert( i>=0 && i<(int)vtcs.size() );
        return vtcs[i].t == 0;
}
```