Graph Partitioning

The Kernighan–Lin Algorithm Approach

Introduction

The Kernighan–Lin algorithm is a heuristic algorithm for finding partitions of graphs[1]. It's the most popular greedy algorithm for two-way partitioning problem. The main idea is to define a function gain for a network partitioning where gain value is the difference between the number of edges in the community. It attempts to find an optimal series of interchange operations between elements of two partitions which maximizes the gain value of the gain function, and then executes the function to produce a partition of the graph to two partitions.

The input of the kernighan-Lin algorithm is an undirected graph G = (V, E) with the vertex set V, the edge set E and weights of the edges in E. The goal of it is to partition V into two disjoint subsets A and B, which A and B has the equal size, in a way that minimizes the sum T of the weights of the subset of edges that cross from A to B. But in the case of that the undirected graph G is unweighted, the goal is to minimize the number of crossing edges.

The specific strategy is to move the vertices from one community structure to the other community structures or to exchange vertices in different community structures. The search starts from the initial solution until no better candidate solution is found from the current solution and then stops. The pseudo code[2](FIG 1) in the below.

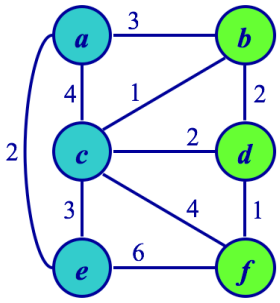
FIG 1: Pseudo Code of KL Algorithm

```
1  function Kernighan-Lin(G(V,E)):
2      determine a balanced initial partition of the nodes into sets A and B
3
4      do
5          compute D values for all a in A and b in B
6          let gv, av, and bv be empty lists
7          for (n := 1 to |V|/2)
8              find a from A and b from B, such that g = D[a] + D[b] - 2*E(a, b) is maximal
9              remove a and b from further consideration in this pass
10             add g to gv, a to av, and b to bv
11             update D values for the elements of A = A \ a and B = B \ b
12         end for
13         find k which maximizes g_max, the sum of gv[1],...,gv[k]
14         if (g_max > 0) then
15             Exchange av[1],av[2],...,av[k] with bv[1],bv[2],...,bv[k]
16     until (g_max <= 0)
17 return G(V,E)
```

Experimental Design

Assume there is a given undirected weighted graph G (FIG 2) with V(G) = { a, b, c, d, e, f }.

FIG 2: An Undirected Weighted Graph G



Step 1 : Start with any partition of V(G) into X = { a, c, e} and Y = { b, d, f } and then compute the gain values of moving vertex set X to vertex set Y. First, get the cut-size equals to 16 (3+ 1+ 2 + 4 + 6) From the definition gain value G = E - I, where E is the cost of edges connecting node with the other group, and, I is the cost of edges connecting node within its own group. Base on this formula, the gain value of every vertex which from the set {a, b, c, d, e, f } are:

Ga = Ea - Ia = 3 - 4 -2 = -3

Gc = Ec - Ic = 1 + 2 + 4 - 4 -3 = 0

Ge = Ee - Ie = 6 -2 -3 = 1

Gb = Eb - Ib = 3 + 1 -2 = 2

Gd = Ed - Id = 2 -2 -1 = -1

Gf = Ef - If = 4 + 6 -1 = 9

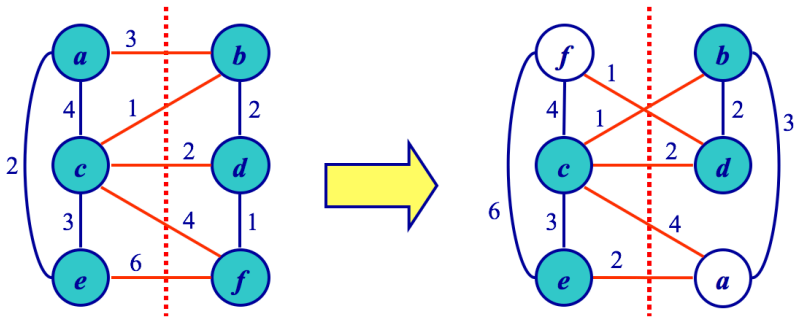
Step 2 : Calculate the real gain value. The cost saving when exchanging vertex a and b is Ga + Gb. However, the cost saving 3 of the direct edge was counted twice. But this edge still connects the two groups, Hence, the real gain will be Gab = Ga + Gb - 2Cab. Where C is the wight of a to b. For example:

Gab = Ga + Gb - 2Cab = -3 + 2 - 2 x 3 = -7

Repeat this method for every connected to the other set vertex, then get the following results: { Gab = -7; Gad = -4, Gaf = 6; Gcb = 0; Gcd = -5; Gcf = 1; Geb = 3; Ged = 0; Gef = -2}.

Step 3 : Get the pair with maximum real gain. Compare the set of real gain, the pair with maximum gain is Gaf. Then Exchange the vertex a to f (FIG 3), and lock them at the same time.

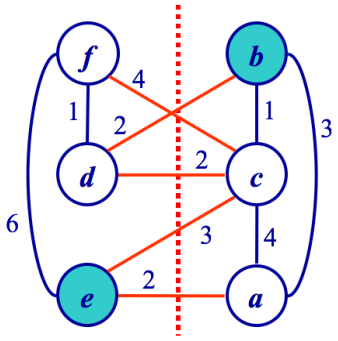
FIG 3 Exchange of Vertex a and f



After the exchange, the partition of G(v) becomes to X' = {c, e } and Y' = {b, d }

Step 4 : Repeat the step 1 to step 3, get the set X'' = {e} and {b}, and get the maximum gain of Gcd with -3 and then repeat agin to get the the maximum gain of Geb with -3. Finally, get the graph (FIG 4） below.

FIG 4 : The Graph after Moving Vertex



After step 1 - 4, it finished the 1 pass. So far the the gain vale of G = 6, G+G' = 3 and G + G' + G'' = 0. So, the maximum gain is g which equals 6. The vertex a and f had been exchanged.

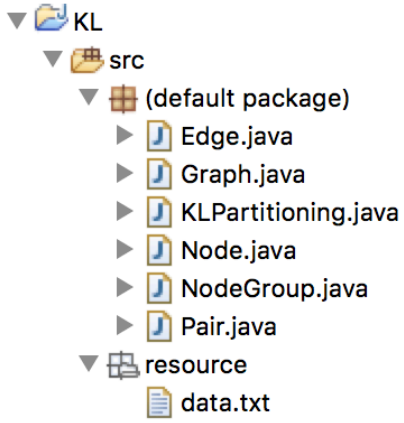
Step 5: Repeat the Kernighan - Lin algorithm until there is no better candidate solution is found from the current solution and then stops.

For each pass, it cost O(n^2) time to find the best pair to exchange, so the complexity of n pairs exchange by using this algorithm is O(n^3) per pass. The best case should be O(n^2lg n).

Implementation and Results

An implementation of the Kernighan - Lin algorithm for splitting a graph into two sets where the weights of the edges between groups. It (FIG 5) contains Edge, Node, NodeGroup, Pair, Graph and the main class KLPartitioning.

FIG 5: An implementation of the Kernighan - Lin algorithm



First, initialized graph, read the nodes, edges, and the weight from the data resource.

The date set:

Vertex set :{ node1, node2, node3, node4, node5, node6, node7, node8 }

Edge set with weight : {node1: node8: 132, node1: node7: 70, node1: node6: 111, node3: node6: 80, node3: node8: 55, node2: node3: 133, node2: node4: 289, node4: node5: 31}

Second, Star to perfoem Kerninghan-Lin algorithm on the given graph.

1, Split nodes into group A and B

```
        int i = 0;
        for (Node v : this.graph.getNodes()) {
            (++i > partitionSize ? B : A).add(v);
        }
        unswappedA = new NodeGroup(A);
        unswappedB = new NodeGroup(B);

        doAllSwaps();
    }

    public NodeGroup getGroupA() {
        return A;
    }

    public NodeGroup getGroupB() {
        return B;
    }

    public Graph getGraph() {
        return graph;
    }
```

2, Perform moving the node swaps from group A to B, and choose the one with least cut cost one

```
    private void doAllSwaps() {

        LinkedList<Pair<Node>> swaps = new LinkedList<Pair<Node>>();
        double minCost = Double.POSITIVE_INFINITY;
        int minId = -1;

        for (int i = 0; i < partitionSize; i++) {
            double cost = doSingleSwap(swaps);
            if (cost < minCost) {
                minCost = cost;
                minId = i;
            }
        }

        // Unwind swaps
        while (swaps.size() - 1 > minId) {
            Pair<Node> pair = swaps.pop();
            // unswap
            swapNodes(A, pair.second, B, pair.first);
        }
    }
```

3, Swap Va and Vb within group A and B

```
private static void swapNodes(NodeGroup a, Node va, NodeGroup b, Node vb) {
    if (!a.contains(va) || a.contains(vb) || !b.contains(vb) || b.contains(va))
        throw new RuntimeException("Invalid swap");
    a.remove(va);
    a.add(vb);
    b.remove(vb);
    b.add(va);
}
```

4, Choose the least cut cost swap and perform it

```
private double doSingleSwap(Deque<Pair<Node>> swaps) {

    Pair<Node> maxPair = null;
    double maxGain = Double.NEGATIVE_INFINITY;

    for (Node v_a : unswappedA) {
        for (Node v_b : unswappedB) {

            Edge e = graph.findEdge(v_a, v_b);
            double edge_cost = (e != null) ? e.weight : 0;
            // Calculate the gain in cost if these nodes were swapped
            // subtract 2*edge_cost because this edge will still be an
            // external edge
            // after swapping
            double gain = getNodeCost(v_a) + getNodeCost(v_b) - 2 * edge_cost;

            if (gain > maxGain) {
                maxPair = new Pair<Node>(v_a, v_b);
                maxGain = gain;
            }

        }
    }

    swapNodes(A, maxPair.first, B, maxPair.second);
    swaps.push(maxPair);
    unswappedA.remove(maxPair.first);
    unswappedB.remove(maxPair.second);

    return getCutCost();
}
```

4, Returns the difference of external cost and internal cost of this node.When moving a node from within group A, all internal edges become external edges and vice versa.

```
    private double getNodeCost(Node v) {

        double cost = 0;

        boolean v1isInA = A.contains(v);

        for (Node v2 : graph.getNeighbors(v)) {

            boolean v2isInA = A.contains(v2);
            Edge edge = graph.findEdge(v, v2);

            if (v1isInA != v2isInA) // external
                cost += edge.weight;
            else
                cost -= edge.weight;
        }
        return cost;
    }
```

5, Returns the sum of the costs of all edges between group A and B

```
private double getCutCost() {
    double cost = 0;

    for (Edge edge : graph.getEdges()) {
        Pair<Node> endpoints = graph.getEndpoints(edge);

        boolean firstInA = A.contains(endpoints.first);
        boolean secondInA = A.contains(endpoints.second);

        if (firstInA != secondInA) // external
            cost += edge.weight;
    }
    return cost;
}
```

6, Main

```
public static void main(String[] args) {
    KLPartitioning partitioning = new KLPartitioning();
    partitioning.process();

    System.out.print("Group A:   ");
    for (Node x : partitioning.getGroupA()) {
        System.out.print(x);
        System.out.print("  ");
    }
    System.out.print("\nGroup B:   ");
    for (Node x : partitioning.getGroupB()) {
        System.out.print(x);
        System.out.print("  ");
    }
    System.out.println("\nCut cost:   " + partitioning.getCutCost());
}
```

Result:

After perform the Kerninghan-Lin algorithm on the given graph, get the result(FIG 6) below:

FIG 6: Output

```
Group A:   node3  node4  node2  node5
Group B:   node7  node6  node1  node8
Cut cost:   135.0
```

Conclusion

The Kernighan-Lin algorithm is probably the most intensely studied algorithm for partitioning. In the future, the main improvements to the algorithm is to improve its running time. The unimproved running time is actually quite bad due to the need to examine n^2/4 pairs of vertex. If we simplistically search through the list every time we wish to swap a new pair of vertex, the total running time of each iteration would be n^3 /4. This could lead to a total running time of O(n^4) or worse. Needless to say, there has been considerable interest in algorithms that deal with individual vertex rather than pairs of vertices.

References

[1] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs" The Bell System Technical Journal, 49(2):291-307, 1970

[2] Ravikumār, Si. Pi; Ravikumar, C.P (1995). Parallel methods for VLSI layout design. Greenwood Publishing Group. p. 73. ISBN 978-0-89391-828-6.