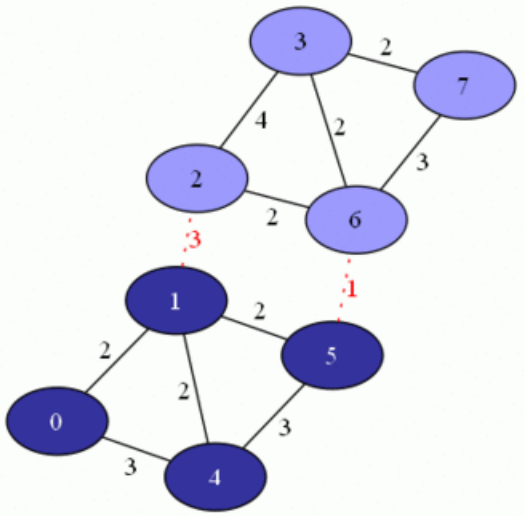
WIKIPEDIA

Stoer–Wagner algorithm

In graph theory, the **Stoer–Wagner algorithm** is a recursive algorithm to solve the minimum cut problem in undirected weighted graphs with non-negative weights. It was proposed by Mechthild Stoer and Frank Wagner in 1995. The essential idea of this algorithm is to shrink the graph by merging the most intensive vertices, until the graph only contains two combined vertex sets.[2] At each phase, the algorithm finds the minimum s-t cut for two vertices s and t chosen at its will. Then the algorithm shrinks the edge between s and t to search for non s-t cuts. The minimum cut found in all phases will be the minimum weighted cut of the graph.



A min-cut of a weighted graph having min-cut weight 4[1]

A **cut** is a partition of the vertices of a graph into two non-empty, disjoint subsets. A minimum cut is a cut for which the size or weight of the cut is not larger than the size of any other cut. For an unweighted graph, the minimum cut would simply be the cut with the least edges. For a weighted graph, the sum of all edges' weight on the cut determines whether it is a minimum cut. In practice, the minimum cut problem is always discussed with the maximum flow problem, to explore the maximum capacity of a network, since the minimum cut is a bottleneck in a graph or network.

Contents

Stoer–Wagner minimum cut algorithm

Let $G = (V, E, w)$ be a weighted undirected graph. Suppose that $s, t \in V$. The cut is called an s-t cut if exactly one of s or t is in S. The minimal cut of G that is also an s-t cut is called the s-t min-cut of G.[3]

This algorithm starts by finding an s and a t in V, and an s-t min-cut (S, T) of G. For any pair $\{s, t\}$, there are two possible situations: either (S, T) is a global min-cut of G, or s and t belong to the same side of the global min-cut of G. Therefore, the global min-cut can be found by checking the graph $G/\{s, t\}$, which is the graph after the merging of vertices s and t. During the merging, if s and t are connected by an edge then this edge disappears. If s and t both have edges to some vertex v, then the weight of the edge from the new vertex st to v is $w(s, v) + w(t, v)$.[3] The algorithm is described as:[2]

MinimumCutPhase(G, w, a)
  $A \leftarrow \{a\}$
  while $A \neq V$
    add to A the most tightly connected vertex
  store the cut in which the last remaining vertex is by itself (the "cut-of-the-phase") and shrink G by merging the two vertices added last

MinimumCut(G, w, a)
  while $|V| > 1$
    MinimumCutPhase(G, w, a)
    if the cut-of-the-phase is lighter than the current minimum cut
      then store the cut-of-the-phase as the current minimum cut

The algorithm works in phases. In the MinimumCutPhase, the subset A of the graphs vertices grows starting with an arbitrary single vertex until A is equal to V. In each step, the vertex which is outside of A, but most tightly connected with A is added to the set A. This procedure can be formally shown as:[2] add vertex $z \notin A$ such that $w(A, z) = \max\{w(A, y) \mid y \notin A\}$, where $w(A, y)$ is the sum of the weights of all the edges between A and y. So, in a single phase, a pair of vertices s and t , and a min s-t cut C is determined.[4] After one phase of the MinimumCutPhase, the two vertices are merged as a new vertex, and edges from the two vertices to a remaining vertex are replaced by an edge weighted by the sum of the weights of the previous two edges. Edges joining the merged nodes are removed. If there is a minimum cut of G separating s and t, the C is a minimum cut of G. If not, then the minimum cut of G must have s and t on a same side. Therefore, the algorithm would merge them as one node. In addition, the MinimumCut would record and update the global minimum cut after each MinimumCutPhase. After $n - 1$ phases, the minimum cut can be determined.[4]

Example

This section refers to Figs. 1–6 in the original paper[2]

The graph in step 1 shows the original graph G and randomly selects node 2 as the starting node for this algorithm. In the MinimumCutPhase, set A only has node 2, the heaviest edge is edge (2,3), so node 3 is added into set A. Next, set A contains node 2 and node 3, the heaviest edge is (3,4), thus node 4 is added to set A. By following this procedure, the last two nodes are node 5 and node 1, which are s and t in this phase. By merging them, the new graph is as shown in step 2. In this phase, the weight of cut is 5, which is the summation of edges (1,2) and (1,5). Right now, the first loop of MinimumCut is completed.

In step 2, starting from node 2, the heaviest edge is (2,15), thus node 15 is put in set A. The next heaviest edges is (2,3) or (15,6), we choose (15,6) thus node 6 is added to the set. Then we compare edge (2,3) and (6,7) and choose node 3 to put in set A. The last two nodes are node 7 and node 8. Therefore, merge edge (7,8). The minimum cut is 5, so remain the minimum as 5.

The following steps repeat the same operations on the merged graph, until there is only one edge in the graph, as shown in step 7. The global minimum cut has edge (2,3) and edge (6,7), which is detected in step 5.

Proof of correctness

To prove the correctness of this algorithm, we need to prove that the cut given by MinimumCutPhase is in fact a minimum s-t cut of the graph, where s and t are the two vertices last added in the phase. Therefore, a lemma is shown below:

Lemma 1: MinimumCutPhase returns a minimum s-t-cut of G.

Let $C = (X, \overline{X})$ be an arbitrary s-t cut, and CP be the cut given by the phase. We must show that $W(C) \geq W(CP)$. Observe that a single run of MinimumCutPhase gives us an ordering of all the vertices in the graph (where a is the first and s and t are the two vertices added last in the phase). We say the vertex v is active if v and the vertex added just before v are in opposite sides of the cut. We prove the lemma by induction on the set of active vertices. We define A_v as the set of vertices added to A before v, and C_v to be the set of edges in C with both of their ends in $A_v \cup \{v\}$, i.e. $C_v \subseteq C$ is the cut induced by $A_v \cup \{v\}$. We prove, for each active vertex v,

$$w(A_v, v) \leq w(C_v)$$

Let v_0 be the first active vertex. By the definition of these two quantities, $w(A_{v_0}, v_0)$ and $w(C_{v_0})$ are equivalent. A_{v_0} is simply all vertices added to A before v_0, and the edges between these vertices and v_0 are the edges that cross the cut C. Therefore, as shown above, for active vertices v and u, with v added to A before u:

$$w(A_u, u) = w(A_v, u) + w(A_u - A_v, u)$$

$w(A_u, u) \leq w(C_v) + w(A_u - A_v, u)$ by induction, $w(A_v, u) \leq w(A_v, v) \leq w(C_v)$

$w(A_u, u) \leq w(C_u)$ since $w(A_u - A_v, u)$ contributes to $w(C_u)$ but not to $w(C_v)$ (and other edges are of non-negative weights)

Thus, since t is always an active vertex since the last cut of the phase separates s from t by definition, for any active vertex t:

$$w(A_t, t) \leq w(C_t) = w(C)$$

Therefore, the cut of the phase is at most as heavy as C.

Time complexity

The running time of the algorithm **MinimumCut** is equal to the added running time of the $|V| - 1$ runs of **MinimumCutPhase**, which is called on graphs with decreasing number of vertices and edges.

For the **MinimumCutPhase**, a single run of it needs at most $O(|E| + |V| \log |V|)$ time.

Therefore, the overall running time should be the product of two phase complexity, which is $O(|V||E| + |V|^2 \log |V|)$[2].[2]

For the further improvement, the key is to make it easy to select the next vertex to be added to the set A, the most tightly connected vertex. During execution of a phase, all vertices that are not in A reside in a priority queue based on a key field. The key of a vertex V is the sum of the weights of the edges connecting it to the current A, that is, $w(A, v)$. Whenever a vertex v is added to A we have to perform an update of the queue. v has to be deleted from the queue, and the key of every vertex w not in A, connected to v has to be increased by the weight of the edge vw, if it exists. As this is done exactly once for every edge, overall we have to perform $|V|$ ExtractMax and $|E|$ IncreaseKey

operations. By using the Fibonacci heap we can perform an ExtractMax operation in $O(\log |V|)$ amortized time and an IncreaseKey operation in $O(1)$ amortized time. Thus, the time we need for this key step that dominates the rest of the phase, is $O(|E| + |V| \log |V|)$.[2]

Example code[5]

```
// Adjacency matrix implementation of Stoer–Wagner min cut algorithm.
//
// Running time:
//     O(|V|^3)
//
// INPUT:
//     - graph, constructed using AddEdge()
//
// OUTPUT:
//     - (min cut value, nodes in half of min cut)

#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> VVI;

const int INF = 1000000000;

pair<int, VI> GetMinCut(VVI &weights)
{
    int N = weights.size();
    VI used(N), cut, best_cut;
    int best_weight = -1;

    for (int phase = N-1; phase >= 0; phase--)
    {
        VI w = weights[0];
        VI added = used;
        int prev, last = 0;
```

```
        for (int i = 0; i < phase; i++)
        {
            prev = last;
            last = -1;
            for (int j = 1; j < N; j++)
            {
                if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
            }
            if (i == phase-1)
            {
                for (int j = 0; j < N; j++) weights[prev][j] += weights[last][j];
                for (int j = 0; j < N; j++) weights[j][prev] = weights[prev][j];
                used[last] = true;
                cut.push_back(last);  // this part gives wrong answer.
                                      // EX) n=4, 1st step: prev=1, last=2 / 2nd step: prev=3, last=4
                                      // if 2nd step gives mincut, the cut is {1,2,3},{4} but this code gives wrong answer - {1,3},{2,4}
                if (best_weight == -1 || w[last] < best_weight)
                {
                    best_cut = cut;
                    best_weight = w[last];
                }
            }
            else
            {
                for (int j = 0; j < N; j++)
                {
                    w[j] += weights[last][j];
                    added[last] = true;
                }
            }
        }
    }
    return make_pair(best_weight, best_cut);
}
```

```
const int maxn = 550;
const int inf = 1000000000;
int n, r;
int edge[maxn][maxn], dist[maxn];
bool vis[maxn], bin[maxn];
void init()
```

```
{
    memset(edge, 0, sizeof(edge));
    memset(bin, false, sizeof(bin));
}
int contract( int &s, int &t )          // Find s,t
{
    memset(dist, 0, sizeof(dist));
    memset(vis, false, sizeof(vis));
    int i, j, k, mincut, maxc;
    for (i = 1; i <= n; i++)
    {
        k = -1; maxc = -1;
        for (j = 1; j <= n; j++)if (!bin[j] && !vis[j] && dist[j] > maxc)
        {
            k = j;  maxc = dist[j];
        }
        if (k == -1) return mincut;
        s = t;  t = k;
        mincut = maxc;
        vis[k] = true;
        for (j = 1; j <= n; j++) if (!bin[j] && !vis[j])
            dist[j] += edge[k][j];
    }
    return mincut;
}


int Stoer_Wagner()
{
    int mincut, i, j, s, t, ans;
    for (mincut = inf, i = 1; i < n; i++)
    {
        ans = contract( s, t );
        bin[t] = true;
        if (mincut > ans) mincut = ans;
        if (mincut == 0)return 0;
        for (j = 1; j <= n; j++) if (!bin[j])
            edge[s][j] = (edge[j][s] += edge[j][t]);
    }
    return mincut;
}
```

References

1. "Boost Graph Library: Stoer–Wagner Min-Cut - 1.46.1" (http://www.boost.org/doc/libs/1_46_1/libs/graph/doc/stoer_wagner_min_cut.html). www.boost.org. Retrieved 2015-12-07.
2. "A Simple Min-Cut Algorithm" (https://scholar.google.com/scholar?cluster=10111487970680388034).
3. "Lecture notes for Analysis of Algorithms": Global minimum cuts" (http://www.cse.iitd.ernet.in/~ssen/csl356/root.pdf) (PDF).
4. "The minimum cut algorithm of Stoer and Wagner" (http://e-maxx.ru/bookz/files/mehlhorn_mincut_stoer_wagner.pdf) (PDF).
5. "Stanford University ACM Team Notebook (2014–15)" (https://web.stanford.edu/~liszt90/acm/notebook.html). web.stanford.edu. Retrieved 2015-12-07.

External links

- StoerWagnerMinCut.java (https://github.com/thomasjungblut/tjungblut-graph/blob/master/src/de/jungblut/graph/partition/StoerWagnerMinCut.java) - a Java library that implements the Stoer-Wagner algorithm

Retrieved from "https://en.wikipedia.org/w/index.php?title=Stoer–Wagner_algorithm&oldid=988194166"

This page was last edited on 11 November 2020, at 17:59 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.