The minimum cut algorithm of Stoer and Wagner

Kurt Mehlhorn and Christian Uhrig

June 27, 1995

1. Min-cuts in undirected graphs.

Let $G = (V, E)$ be an undirected graph (self-loops and parallel edges are allowed) and let $w : E \to \Re_{\geq 0}$ be a non-negative weight function on the edges of G. A cut C of G is any subset of V with $\emptyset \neq C \neq V$. The weight of a cut is the total weight of the edges crossing the cut, i.e.,

$$w(E) = \sum_{e \in E; |e \cap C| = 1} w(e).$$

A minimum cut is a cut of minimum weight. For a pair $\{s,t\}$ of distinct vertices of G a cut is called an s-t cut if C contains exactly one of s and t. We describe a particularly simple and efficient min-cut algorithm due to Stoer and Wagner ([SW94]). The algorithm runs in time $O(nm + n^2 \log n)$.

The algorithm works in phases. In each phase it determines a pair of vertices s and t and a minimum s-t cut C. If there is a minimum cut of G separating s and t then C is a minimum cut of G. If not then any minimum cut of G has s and t on the same side and therefore the graph obtained from G by *combining* s and t has the same minimum cut as G. So a phase determines vertices s and t and a minimum s-t cut C and then combines s and t into one node. After $n - 1$ phases the graph is shrunk to a single node and one of the phases must have determined a minimum cut of G.

```
⟨ mincut.c 1 ⟩ ≡
#include <LEDA/graph_alg.h>
#include <LEDA/ugraph.h>
#include <LEDA/stream.h>
#include <LEDA/node_pq.h>
  list⟨node⟩ minimum_cut(const graph &G0, edge_array⟨int⟩ &weight)
  {
    ⟨ initialization 2 ⟩
    /* n is now the number of nodes of the current graph and a is a fixed
    vertex */
    while (n ≥ 2) {⟨ a phase 5 ⟩}
    ⟨ output best cut 4 ⟩
  }
```

2. We call our input graph *G0* and our current Graph G. G is of type **UGRAPH**⟨**list**⟨**node**⟩ $*$, **int**⟩. Every node of G represents a set of nodes of *G0*. This set is stored in a linear list pointed to by $G[v]$. Every edge $e = \{v, w\}$ of G represents a set of edges of *G0*, namely $\{\{x, y\}; x \in G[v]$ and $y \in G[w]\}$. The total weight of these edges is stored in $G[e]$.

It is easy to initialize G. We simply make G a copy of *G0* (except for self-loops) and initialize $G[v]$ to the appropriate singleton set for every vertex v of G.

```
⟨initialization 2⟩ ≡
  typedef list⟨node⟩ nodelist;
  UGRAPH⟨nodelist *, int⟩ G;
  node v, x;
  edge e;
  node_array⟨node⟩ partner(G0);
  forall_nodes (x, G0) {
    partner[x] = G.new_node(new nodelist);
    G[partner[x]]¬append(x);
  }
  forall_edges (e, G0)
    if (source(e) ≠ target(e))
      G.new_edge(partner[source(e)], partner[target(e)], weight[e]);
```

See also section 3.

This code is used in section 1.

3. We also need to fix a particular node a of G, define n as the number of nodes of G, and introduce variables to store the currently best cut.

```
⟨initialization 2⟩ +≡
  node a = G.first_node( );
  int n = G.number_of_nodes( );
  list⟨node⟩ best_cut;
  int best_value = MAXINT;
  int cut_weight;
```

4. Outputting the best cut is easy.

```
⟨output best cut 4⟩ ≡
  return best_cut;
```

This code is used in section 1.

5. We now come to the heart of the matter, a phase. A phase initializes a set A to the singleton set $\{a\}$ and then successively merges all the other nodes of G into A. In each stage the node $v \notin A$ which maximizes

$$w(v, A) = \sum\{w(e); e = \{v, y\} \text{ for some } y \in A\}$$

is merged into A. Let s and t be the last two vertices added to A in a phase. The cut C computed by the phase is the cut consisting of node t only; in the graph *G0* this corresponds to the cut $G[t]$.

Lemma 1 *Let s and t be the last two nodes merged into A during a phase. Then $\{t\}$ is a minimum s-t cut.*

Proof:

Let C' be any s-t cut. We show that $w(C') \geq w(\{t\})$. Let $v_1, \ldots, v_n$ be the order in which the nodes are added to A. Then $v_1 = a, v_{n-1} = s$, and $v_n = t$.

Call a vertex $v = v_i$ critical if $i \geq 2$ and v_i and v_{i-1} belong to different sides of C'. Note that t is critical. Let k be the number of critical nodes and let i_1, i_2,..., i_k be the indices of the critical nodes. Then $i_k = n$. For integer i use A_i to denote the set $\{v_1, \ldots, v_i\}$. Then $w(\{t\}) = w(v_{i_k}, A_{i_k - 1})$ and $w(C') \geq \sum_{j=1}^{k} w(v_{i_j}, A_{i_j - 1} \setminus A_{i_{j-1}-1})$ since any edge counted on the right side is also counted on the left and edge costs are non-negative. We now show for all integer l, $1 \leq l \leq k$, that

$$w(v_{i_l}, A_{i_l - 1}) \leq \sum_{j=1}^{l} w(v_{i_j}, A_{i_j - 1} \setminus A_{i_{j-1}-1}).$$

For $l = 1$ we have equality. So assume $l \geq 2$. We have

$$\begin{aligned} w(v_{i_l}, A_{i_l - 1}) &= w(v_{i_l}, A_{i_{l-1}-1}) + w(v_{i_l}, A_{i_l - 1} \setminus A_{i_{l-1}-1}) \\ &\leq w(v_{i_{l-1}}, A_{i_{l-1}-1}) + w(v_{i_l}, A_{i_l - 1} \setminus A_{i_{l-1}-1}) \\ &\leq \sum_{j=1}^{l-1} w(v_{i_j}, A_{i_j - 1} \setminus A_{i_{j-1}-1}) + w(v_{i_l}, A_{i_l - 1} \setminus A_{i_{l-1}-1}) \\ &\leq \sum_{j=1}^{l} w(v_{i_j}, A_{i_j - 1} \setminus A_{i_{j-1}-1}). \end{aligned}$$

Here the first inequality follows from the fact that $v_{i_{l-1}}$ is added to $A_{i_{l-1}-1}$ and not v_{i_l} and the second inequality uses the induction hypothesis.

∎

```
⟨ a phase 5 ⟩ ≡
  ⟨ determine s and t and the value cut_weight of the cut {t} 6 ⟩;
  if (cut_weight < best_value) {
    best_cut = *(G[t]);
    best_value = cut_weight;
  }
  ⟨ combine s and t 7 ⟩;
  n--;
```

This code is used in section 1.

6. How can we determine the order in which the vertices are merged into A? This can be done in a manner akin to Prim's minimum spanning tree algorithm. We keep the vertices v, $v \notin A$, in a priority queue ordered according to $w(v, A)$. In each stage we select the node, say u, with maximal $w(u, A)$ and add it to A. This increases $w(v, A)$ by $w(\{v, u\})$ for any vertex $v \notin A$ and $v \neq u$. Since LEDA priority queues select minimal values we store $-w(v, A)$ in the queue. The node added last to A is the vertex t. The value *cut_weight* is $w(t, A_t)$.

```
⟨ determine s and t and the value cut_weight of the cut {t} 6 ⟩ ≡
  node t = a;
  node s;
  node_array⟨bool⟩ in_PQ(G);
  node_pq⟨int⟩ PQ(G);
  forall_nodes (v, G)
    if (v ≠ a) {
      PQ.insert(v, 0);
          // w(v, A) = 0 if there is no edge connecting v to A
      in_PQ[v] = true;
    }
  forall_adj_edges (e, a)
    PQ.decrease_inf(G.opposite(a, e), PQ.inf(G.opposite(a, e)) − G[e]);
  while (¬PQ.empty( )) {
    s = t;
    cut_weight = −PQ.inf(PQ.find_min( ));
    t = PQ.del_min( );
    in_PQ[t] = false;
    forall_adj_edges (e, t) {
      if (in_PQ[v = G.opposite(t, e)]) PQ.decrease_inf(v, PQ.inf(v) − G[e]);
    }
  }
```

This code is used in section 5.

7. It remains to combine s and t. We do so by deleting t from G and moving all edges incident to t to s. More precisley, we need to do three things:

- Add $G[t]$ to $G[s]$ ($G[s]$→$conc(*(G[t]))$).
- Increase $G[\{s, v\}]$ by $G[\{t, v\}]$ for all vertices v with $\{t, v\} \in E$ and $v \neq s$.
- Delete t and all its incident edges from G ($G.del\_node(t)$).

The second step rises two difficulties: The edge $\{s, v\}$ might not exist and there is no simple way to go from the edge $\{t, v\}$ to the edge $\{s, v\}$. We overcome these problems by first recording the edge $\{s, v\}$ in $s\_edge[v]$ for every neighbor

v of s. We then go through the neighbors v of t: If v is connected to s then we simply increase $G[\{s,v\}]$ by $G[\{t,v\}]$, if v is not connected to s and different from s then we add a new edge $\{s,v\}$ with weight $G[\{t,v\}]$.

```
⟨ combine s and t 7 ⟩ ≡
  G[s]→conc(*(G[t]));
  node_array⟨edge⟩ s_edge(G, nil);
  forall_adj_edges (e, s) s_edge[G.opposite(s, e)] = e;
  forall_adj_edges (e, t) {
    if (s_edge[v = G.opposite(t, e)] ≡ nil) G.new_edge(s, v, G[e]);
    else G[s_edge[v]] += G[e];
  }
  delete G[t];
  G.del_node(t);
```

This code is used in section 5.

8. The running time of our algorithm is clearly at most n times the running time of a phase. A phase takes time $O(m + n \log n)$ to merge all nodes into the set A (the argument is the same as for Prim's algorithm) and time $O(n)$ to record the cut computed and to merge s and t. The total running time is therefore $O(nm + n^2 \log n)$.

Table 1 lists the running times (in seconds) for some experiments with random graphs on a SPARCstation 10/52. The first column gives the number of nodes and the first row the number of edges. If there are more than 400 nodes the running time is about $9nm + 8.5n^2 \log n$ μsec.

References

[SW94] M. Stoer, and F. Wagner. A Simple Min Cut Algorithm. *Algorithms - ESA '94, LNCS 855*, 141–147, 1994.

	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
100	0.52	0.96	1.31	1.51	1.76	2.03	2.56	3.02	2.75	3.54
200	1.72	2.91	4.01	4.62	5.62	6.43	7.26	7.74	8.58	9.29
300	3.19	5.17	7.13	8.79	10.57	12.37	13.33	14.66	16.62	17.86
400	4.59	8.12	11.31	14.02	16.40	18.82	20.61	22.80	26.04	28.12
500	6.26	11.04	15.43	18.93	22.47	26.05	29.14	32.17	35.79	40.12
600	8.42	14.50	20.09	24.81	29.89	33.75	38.84	42.38	46.71	50.54
700	10.91	18.08	24.19	31.03	36.55	43.24	48.47	53.72	61.18	68.01
800	13.74	22.81	30.92	39.65	46.75	52.78	58.95	64.46	71.39	79.24
900	15.71	25.38	35.80	42.83	52.73	60.99	68.59	76.98	85.08	92.24
1000	18.70	31.75	41.38	50.95	60.13	70.62	79.04	87.96	96.38	107.13

Table 1: Some experimental results

Index

List of Refinements

⟨ a phase 5 ⟩ Used in section 1.
⟨ combine s and t 7 ⟩ Used in section 5.
⟨ determine s and t and the value *cut_weight* of the cut $\{t\}$ 6 ⟩ Used in section 5.
⟨ initialization 2, 3 ⟩ Used in section 1.
⟨ `mincut.c` 1 ⟩
⟨ output best cut 4 ⟩ Used in section 1.