Reverse-delete algorithm

The **reverse-delete algorithm** is an algorithm in graph theory used to obtain a minimum spanning tree from a given connected, edge-weighted graph. It first appeared in Kruskal (1956), but it should not be confused with Kruskal's algorithm which appears in the same paper. If the graph is disconnected, this algorithm will find a minimum spanning tree for each disconnected part of the graph. The set of these minimum spanning trees is called a minimum spanning forest, which contains every vertex in the graph.

This algorithm is a greedy algorithm, choosing the best choice given any situation. It is the reverse of Kruskal's algorithm, which is another greedy algorithm to find a minimum spanning tree. Kruskal's algorithm starts with an empty graph and adds edges while the Reverse-Delete algorithm starts with the original graph and deletes edges from it. The algorithm works as follows:

- Start with graph G, which contains a list of edges E.
- Go through E in decreasing order of edge weights.
- For each edge, check if deleting the edge will further disconnect the graph.
- Perform any deletion that does not lead to additional disconnection.

1 Pseudocode

```
1 **function** ReverseDelete(edges[] $E$)
2   **sort** $E$ in decreasing order
3   Define an index $i \leftarrow 0$
4   **while** $i <$ **size**($E$)
5     Define $edge \leftarrow E[i]$
6     **delete** $E[i]$
7     **if** graph is not connected
8       $E[i] \leftarrow edge$
9     $i \leftarrow i + 1$
10  **return** edges[] $E$
```

In the above the graph is the set of edges E with each edge containing a weight and connected vertices *v1* and *v2*.

2 Example

In the following example green edges are being evaluated by the algorithm and red edges have been deleted.

3 Running time

The algorithm can be shown to run in $O(E \log V (\log \log V)^3)$ time (using big-O notation), where E is the number of edges and V is the number of vertices. This bound is achieved as follows:

- Sorting the edges by weight using a comparison sort takes $O(E \log E)$ time, which can be simplified to $O(E \log V)$ using the fact that the largest E can be is V^2.
- There are E iterations of the loop.
- Deleting an edge, checking the connectivity of the resulting graph, and (if it is disconnected) re-inserting the edge can be done in $O(\log V (\log \log V)^3)$ time per operation (Thorup 2000).

4 Proof of correctness

It is recommended to read the proof of the Kruskal's algorithm first.

The proof consists of two parts. First, it is proved that the edges that remain after the algorithm is applied form a spanning tree. Second, it is proved that the spanning tree is of minimal weight.

4.1 Spanning tree

The remaining sub-graph (g) produced by the algorithm is not disconnected since the algorithm checks for that in line 7. the result sub-graph cannot contain a cycle since if it does then when moving along the edges we would encounter the max edge in the cycle and we would delete that edge.thus g must be a spanning tree of the main graph G.

4.2 Minimality

We show that the following proposition ***P*** is true by induction: If F is the set of edges remained at the end of the while loop, then there is some minimum spanning tree that (its edges) are a subset of *F*.

1. Clearly **P** holds before the start of the while loop . since a weighted connected graph always has a minimum spanning tree and since F contains all the edges of the graph then this minimum spanning tree must be a subset of F.

2. Now assume ***P*** is true for some non-final edge set *F* and let *T* be a minimum spanning tree that is contained in *F* . we must show that after deleting edge e in the algorithm there exist some (possibly other) spanning tree T' that is a subset of F.
   (a) if the next deleted edge e doesn't belong to T then T=T' is a subset of F and ***P*** holds. .
   (b) otherwise, if e belongs to T : first note that the algorithm only removes the edges that do not cause a disconnectedness in the F . so e does not cause a disconnectedness . But deleting e causes a disconnectedness in tree T (since it is a member of T) . assume e separates T into sub-graphs t1 and t2 . Since the whole graph is connected after deleting e then there must exists a path between t1 and t2 (other than e) so there must exist a cycle C in the F (before removing e) . now we must have another edge in this cycle (call it f) that is not in T but it is in F (since if all the cycle edges were in tree T then it would not be a tree anymore) . we now claim that T' = T - e + f is the minimum spanning tree that is a subset of F.
   (c) firstly we prove that T' is a ***spanning tree*** . we know by deleting an edge in a tree and adding another edge that does not cause a cycle we get another tree with the same vertices. since T was a spanning tree so T' must be a ***spanning tree*** too. since adding " f " does not cause any cycles since "e" is removed.(note that tree T contains all the vertices of the graph).
   (d) secondly we prove T' is a ***minimum*** spanning tree . we have three cases for the edges "e" and " f ". wt is the weight function.
      i. wt(f) < wt(e) this is impossible since this causes the weight of tree T' to be strictly less than T . since T is the minimum spanning tree, this is simply impossible.
      ii. wt(f) > wt(e) this is also impossible. since then when we are going through edges in decreasing order of edge weights we must see " f " first . since we have a cycle C so removing " f " would not cause any disconnectedness in the F. so the algorithm would have removed it from F earlier . so " f " does not exist in F which is impossible(we have proved f exists in step 4 .
      iii. so wt(f) = wt(e) so T' is also a ***minimum*** spanning tree. so again ***P*** holds.
3. so ***P*** holds when the while loop is done (which is when we have seen all the edges) and we proved at the end F becomes a ***spanning tree*** and we know F has a ***minimum*** spanning tree as its subset . so F must be the ***minimum spanning tree*** itself .

5 See also

- Kruskal's algorithm
- Prim's algorithm
- Borůvka's algorithm
- Dijkstra's algorithm

6 References

- Kleinberg, Jon; Tardos, Éva (2006), *Algorithm Design*, New York: Pearson Education, Inc..
- Kruskal, Joseph B. (1956), "On the shortest spanning subtree of a graph and the traveling salesman problem", *Proceedings of the American Mathematical Society*, **7** (1): 48–50, doi:10.2307/2033241, JSTOR 2033241.
- Thorup, Mikkel (2000), "Near-optimal fully-dynamic graph connectivity", *Proc. 32nd ACM Symposium on Theory of Computing*, pp. 343–350, doi:10.1145/335305.335345.

7 Text and image sources, contributors, and licenses

7.1 Text

7.2 Images

7.3 Content license