5th April 2013

Alternative Minimum Spanning Trees - 23.4, CLRS

23-4 Alternative minimum-spanning-tree algorithms
In this problem, we give pseudocode for three different algorithms. Each one takes a connected graph and a weight function as input and returns a set of edges T. For each algorithm, either prove that T is a minimum spanning tree or prove that T is not a minimum spanning tree. Also describe the most efficient implementation of each algorithm, whether or not it computes a minimum spanning tree.

a. **MAYBE-MST-A (G, w)**

```
1. sort the edges into non increasing order of edge weights w
2. T = E
3. for each edge e, taken in non increasing order by weight
4.    if T - {e} is a connected graph
5.       T = T - {e}
6. return T
```

Proof of Correctness:

- In order for T to be an MST, we must show that a) It is a spanning tree, and b) It is minimal.
- a) T starts as the entire set of edges, E. We only remove an edge e if T-{e} is a connected graph. A connected graph is by definition a spanning tree, since there is a path from every vertex to every other vertex in G. Thus upon completion of this algorithm, T is indeed a spanning tree.
- b) To show that T is minimal:
  - Consider an arbitrary sub-problem in the algorithm, where there are two or more edges that connect the components in some arbitrary cut of the graph. Call this set of edges E*. Clearly the MST needs only one of these edges, and the rest may be deleted.
  - Until E* is reduced to having only one edge, any edges e* in E* that are encountered in the loop of line (3) will be removed, since T - e* will still be connected. Since we check edges in non-increasing order of weight, the last edge remaining will be the lightest-weight edge that connects the two components of the cut.
  - Thus for each sub-problem / cut of the graph, we have included only one edge (the minimal number of edges), and this edge has the minimum weight of all edges that connect that particular cut. Thus the output T is composed of a minimum number of edges of minimum weight, and is valid MST.

Most Efficient Implementation: /* Consolidation of Relevant Lecture Material */

1. A max priority queue will be used to hold the edges; this can be built in linear time, and insertions and deletions can be done in logarithmic time. Of course finding the max takes constant time in a max PQ.

2. We recall that the union find UF data structure, when using weighted unions (when performing a union, put the smaller tree as a subset of the larger tree) along with path compression (after going down a path to find a node, simply alter all the nodes along that path to become direct descendants of the root), we in practice achieve an essentially linear union operation and check for connection operation, since the lg*V is essentially constant in the practical world. (iterated log, lg*V <= 5 in practice; up to |V| = 2^65536).
3. The algorithm builds the priority queue in O(E), deletes a node in a max of E times, each in O(lg(E)) time, and makes a union O(V) times / does a 'connected' check O(E) times, both in O(lg(V)) time.
4. So let's consider the running time of this algorithm:
    - Line 1 builds a max priority queue / heap in O(E) time.
    - Line 3 runs O(E) times
        - Line 4, within the Line 3 loop, checks connectivity in O(lg(V)) time.
        - Line 5, within the Line 3 loop, deletes an edge in O(lg(E)) time.
    - So Line 4 has running time of O(E*lg(V)), and Line 5 has running time of O(E*lg(E)).
5. **Thus the running time of the algorithm is bounded by O(E*lg(E)) time.**

b. **MAYBE-MST-B (G, w)**

```
1. T = null
2. for each edge e, taken in arbitrary order
3.   if T U {e} has no cycles
4.     T = T U {e}
5. return T
```

Proof of Incorrectness (by Counter-Example):

- First an intuitive explanation for why this is incorrect: the loop considers edges in arbitrary order, and adds any edge e to the MST if T U e contains no cycles. The algorithm never adjusts edges or removes edges. Because of the arbitrary order, clearly we could add a heavy edge that did not make a cycle, when lighter edges would be more optimal.
- Example of Algorithm Failure:

Graph: MST:

```
                   (B)--1--(D)                        (B)---1---(D)
                   /\      /                                     /
                  /  \    /                                     /
                 /    1  /                                     /
```

```
      5      \/                                  /
     /       /\                                 1
    /       /  \                               /
   /       1    \                             /
  /       /      \                           /
(A)--1--(C)--1--(E)               (A)--1--(C)--1--(E)
```

- Consider the graph shown above on the left, along with a sample MST on the right, of weight 4.
- No let us iterate through the *MAYBE-MST-B ( )* algorithm. Since the edges are taken in arbitrary order, let's begin with the left-most ed ge, (A)----(B) of weight 5. T is empty, and thus T U (A, B) has no cycles. Thus T = T U (A, B). The weight of T that will be returned i s already greater than 4, and thus this algorithm is not guaranteed to produce an MST.
- Of course if we were lucky, and simply chose optimal edges to be considered first, by coincidence, we could still produce an MST wit h this algorithm.

Most Efficient Implementation:

1. An array of boolean values will be kept to determine if some vertex is covered by the MST.
2. A queue will contain the edges in the MST

3. Since in Line 3, we check for cycles as we add each edge, the check is simply done by looking in constant time, if the vertices u, v in t he added edge (u, v), are already covered by the MST. If they are covered, then we have a cycle.
4. Inserting {e} to the queue takes constant time. It doesn't matter where this is, since there's no priority.
5. So all we do is initialize the data structures then process the edges:
    - Line 2 runs O(E) times.
        - Line 3, within the Line 2 loop, checks for cycle existence in constant time.
        - Line 4, within the Line 2 loop, appends a node to the queue in constant time.
6. **Thus the running time of the algorithm is bounded by O(E)**

c. **MAYBE-MST-C (G, w)**

```
1. T = null
2. for each edge e, taken in arbitrary order
3.    T = T U {e}
4.    if T has a cycle c
5.       let e' be a maximum-weight edge on c
6.       T = T - {e'}
7. return T
```

Proof of Correctness:

- Again, for T to be an MST, we must show that a) It is a spanning tree, and b) It is minimal.
- a) T will at some point add every edge in G. An edge will only be deleted if it is part of a cycle. Starting with a spanning tree, the removal of any edge that is part of a cycle results in, still, a spanning tree. This is because in a cycle, every vertex is touched by at least two edges; thus removing one edge still leaves every vertex still reachable by one edge. So since T adds every edge at some point, and only deletes an edge that is not necessary for spanning, T must be a spanning tree.
- b) To show that T is minimal:
    - First we note that for a spanning tree to not have the minimum number of edges, there must be a cycle. This is because after an MST is created, any additional edge will be between two already-connected vertices. Thus a cycle will be created.
    - Since if at any point T contains a cycle, we remove a cycle edge, T must have a a minimum number of edges when it is returned.
    - Now we show that those minimum edges are indeed also minimum-weighted.
    - Any time a cycle is created, one edge is deleted. The deleted edge is the heaviest one in the cycle.

- So since: (i), any time we add an unnecessary edge, we delete an edge in that cycle, and (ii) the edge we delete is always the heaviest one, the the output T must indeed be an MST.

Most Efficient Implementation:

1. An array of boolean values will be kept to determine if some vertex is covered by the MST.
2. A max priority queue will contain the edges in the MST for logarithmic insertion/deletion.
3. Line 4's cycle check is actually easier than it seems; since we check for cycles as we add each edge, this check is simply done by looking in constant time, if the vertices u, v in the added edge (u, v), are already covered by the MST. If they are covered, then we have a cycle.
4. We can find the maximum-weight edge of C in constant time with the priority queue.
5. Running time analysis:
    - Line 2 runs O(E) times.
        - Line 3, within the Line 2 loop, inserts a new edge to the priority queue in O(lg(V)) time.
        - Line 4, within the Line 2 loop, checks for cycle existence in constant time.
        - Line 5, within the Line 2 loop, finds the max-weight edge in constant time.
        - Line 6, within the Line 2 loop, deletes an edge in O(lg(V)) time.
    - So the two parts that define the complexity are Lines 3 and 6, which run in O(E(lg(V)) time.
6. **Thus the running time of the algorithm is bounded by O(E*lg(V))**

**** Note that the insertion and deletions are done in O(lg(V)) because we will only have a O(V) edges in our priority queue for the MST at any given point.***

Posted 5th April 2013 by mattislearning

Labels: Algorithms

2 View comments

Lee Jia May 7, 2014 at 8:49 AM

This comment has been removed by the author.

Reply

Lee Jia May 7, 2014 at 8:50 AM

"A connected graph is by definition a spanning tree" -- I don't think so. To be a spanning tree, this graph also needs to be acyclic.

Reply

Enter your comment...

Comment as: lichaoyu1997@ Sign out

Publish Preview Notify me