Algorithms and Data Structures

Minimum Spanning Tree

Marcin Sydow

Web Mining Lab
PJWSTK

Topics covered by this lecture:

- Minimum Spanning Tree (MST) Problem
- Cut Property and Cycle Property of MST
- Prim's Algorithm
- Kruskal's Algorithm
- Union-Find Abstract Data Structure
- (*) Fast Implementation of Union-Find

Minimum Spanning Tree (MST)

Given graph $G = (V, E)$ with weights on edges

Find a tree $T = (V, E')$ (spanned on the vertices of G where $E' \subseteq E$) such that the sum of weights on E' is minimum possible.

INPUT: an undirected connected graph $G = (V, E)$ with positive weights on edges, given by a weight-function: $w : E \rightarrow R^+$

OUTPUT: a graph G' such that:

1. $G' = (V, E')$ is a connected subgraph of G (it connects all the nodes of the original graph G)
2. the sum of weights of its edges $\sum_{e \in E'} w(e)$ is minimum possible

Exercise: Is the MST the same thing as the tree of shortest paths from a source vertex to other vertices?

The "Cut Property" of MST

Definition

For a connected graph $G = (V, E)$ and a subset S of V, a *cut* is a set $E' \subseteq E$ of all edges having one end in S and another in $V \setminus S$.

Lemma

(cut property) Assume E' is a cut and e is a minimum-cost edge in E'. Then, there exists a MST of G that contains e. In addition, if T' is a set of edges contained in some MST and T' does not contain any edge from E' then $T' \cup \{e\}$ is also contained in some MST.

Proof[1] (of a second claim): Let T be a MST of G with $T' \subseteq T$ and $e = (u, v)$. T is a spanning tree, so it contains a path p from u to v. E' separates u and v, since it is a cut, and p must contain an edge e' from E'. Thus, $T'' = (T \setminus \{e'\}) \cup \{e\}$ is also a spanning tree since deleting e' partitions T into two subtrees, which are subsequently joined back by e. But $c(e) \leq c(e')$ implies that $c(T'') \leq c(T)$ so that T'' is also a MST.

The first claim is implied by taking $T' = \emptyset$.

[1] After K.Mehlhorn et al., "Algorithms and Data Structures"

The "Cycle Property" of MST

Lemma

Assume S is a subset of edges of some MST of G and $C \subseteq E$ is a cycle in a connected graph $G = (V, E)$. If $e = (u, v) \in C$ is an edge with maximum cost in C such that u is incident with S, and v not, there exists a MST T' of G that contains S and that does not contain e (i.e. e is "not needed" in any MST of G).

Proof: (*"reductio ad absurdum"*) Assume every extension T of S to MST must contain e. e partitions T into two subtrees T_u, T_v. There must exist another edge $e' = (u', v')$ from C with $u' \in T_u$ and $v' \in T_v$. Now, $T' = (T \setminus \{e\}) \cup \{e'\}$ is a spanning tree that does not contain e. But T' is a MST since $c(e') \leq c(e)$. Contradiction.

From Cut Property to General MST Solution

The cut property can be exploited to design a simple greedy algorithm for finding MST.
The general scheme of such algorithm is as follows:

1. set $T = \emptyset$
2. until T is not a spanning tree, add a minimum-cost edge e from any cut E' disjoint with T

(the cut property guarantees the correctness of the above general approach)

Different choices of the selection of the cut E' in each iteration lead to different algorithms for finding MST.
We will now discuss two different algorithms: Prim's and Kruskal's

Prim's Algorithm - the idea

(similar to Dijkstra Algorithm for finding shortest paths)

Start with any "source" node s and grow the tree as follows. S (initially containing only s) denotes the set of nodes in each iteration. The cut E' is the set of edges having exactly one end in S. In each iteration a minimum-cost edge from E' is added to S.

To efficiently find such a minimum-cost e a priority queue is used, that, for each node outside of S, keeps its current shortest connection to S (actually, such a shortest connection is via the edge e that is sought). After selecting e, all the edges incident to it are relaxed similarly as in the algorithm of Dijkstra.

Prim's Algorithm

($G = (V, E)$ in a form of adjLists, $w(u, v)$ denotes the weight of the edge (u, v) and s denotes the (arbitrary) starting node):

```
MSTPrim(V,w,s){
  PriorityQueue pq
  s.dist = 0
  s.parent = null
  pq.insert(s)
  for each u in V\{s}:
    u.dist = INFINITY

  while(!pq.isEmpty()):
    u = pq.deleteMin()
    u.dist = 0
    for each v in u.adjList:
      if (w(u,v) < v.dist):
        v.dist = w(u,v)
        v.parent = u
        if (pq.contains(v)): pq.decreaseKey(v)
        else pq.insert(v)
}
```

(the resulting MST is encoded in the parent attributes)

Analysis

data size: n=|V|, m=|E|

dominating operation: assignment (initialisation) and comparison (both explicit and hidden inside `deleteMin`, `decreaseKey` operations)

initialisation: $O(n)$, loop: (n times delMin() + m times decreaseKey())

If we implement the priority queue on a binary heap:
loop: $O(nlog(n)) + O(mlog(n)) =$ **O((n+m)log(n))**

If we use Fibonacci heap (amortised constant cost of decreaseKey()):
$O(nlog(n) + m)$

Kruskal's Algorithm - the idea

1. initially $T = \emptyset$
2. add a minimum-cost edge e that does not form a cycle in T until T is a spanning tree

Thus, the edges are considered in the order of non-decreasing weight and each edge is considered only once and is either:

- rejected: due to the cycle property, it is a maximum-cost, cycle-making edge in T in the moment of rejection
- accepted: due to the cut property, it is a minimum-cost member of a cut

The main issue in the algorithm is to efficiently check whether a considered edge is forming a cycle.

A helper *union-find* data structure can be used here. Notice that after each iteration T forms a forest. Thus, forming a cycle for an edge (u, v) is equivalent to u and v belonging to the same subtree of T.

Union-Find (Abstract Data Structure)

Union-Find is an abstract data structure for representing a family of disjoint subsets of some set U, that supports the following operations:

- find(element)
- union(set1,set2)

The first operation answers which set of a family an element belongs to. The second operation joins two (disjoint) sets of the family into one set. The data structure invariant: the subsets are always disjoint.

Kruskal's Algorithm Implemented with Union-Find

A union-find data structure can be applied to implement the Kruskal's algorithm as follows:

```
kruskalMST(V,E,w){
  T = 0
  UnionFind uf
  foreach edge (u,v) in non-decreasing order of weight:
    if (uf.find(u) != uf.find(v)):
      T = T + (u,v)
      uf.union(uf.find(u),uf.find(v))
  return T
}
```

There exists an extremely fast implementation of union-find that has constant complexity of the union operation and "almost"[2] constant amortised complexity. With this implementation, the Kruskal's algorithm will have $O(mlog(m))$ complexity (since sorting m edges will dominate the work).

[2] The word "almost" will be explained soon

Simple Implementation of Union-Find

Assume, the universe set $U = \{1, .., m\}$ and assume that, for each subset of the family, its label is some of its elements. Initially, each element of U forms a separate subset (i.e. $find(e) == e$ for any $e \in U$)

(simple implementation) Union-Find can be simply implemented as an array **uf**, where $uf[e]$ is just the label of the set S containing it (e.g. the minimum element in S). `find` will have constant time complexity, but `union` will have $\Theta(m)$ complexity (as the whole array uf needs to be scanned to update the labels of the joined sets)

There is a much faster implementation.

Fast Implementation of Union-Find - idea

- Each subset of a family is represented as a rooted tree (with elements in nodes).
- The root of each tree contains the representative (label) of each subset.
- Each element keeps a pointer to its parent in the tree.
- find(e): follow a path from e to the root and return the root's element
- union(set1,set2): make a root of one tree a parent of the root of the other one

Fast Implementation of Union-Find - Refinements

There are two possible improvements possible:

- ("union by rank") to keep the height of each tree small, each root contains an integer (called "rank") that is an upper-bound of the height. The tree with the higher rank is made the root while joining two trees.
- ("path compression") to accelerate the find operation. During any find operation, any examined parent attribute is set directly to the root

Lemma

The height of any tree with union by rank is $O(log(n))$

(Proof draft: by induction, each tree of rank k contains at least 2^k nodes.)

(*) Time Complexity of Fast Implementation of Union-Find (The proof is is not simple and is omitted here)

Theorem

The tree-based implementation of Union-Find data structure with "union by rank" and "path compression" achieves $O(m\alpha(m,n))$ time complexity for any sequence of m find and $n-1$ union operations, where:
$\alpha(m,n) = min\{i \geq 1 : A(i, \lceil m/n \rceil) \geq log(n)\}$
and A is defined as follows[3]*:*

- $A(1,j) = 2^j$ *for* $j \geq 1$
- $A(i,1) = A(i-1,2)$ *for* $i \geq 2$
- $A(i,j) = A(i-1, A(i,j-1))$ *for* $i,j \geq 2$

The Ackermann's function grows so extremely fast that $\alpha(m,n)$ is usually less than 5 for any values of m, n found in any current practical applications.

Thus, the amortised time complexity is "almost" constant.

[3] A is called Ackermann's function

Summary of Prim's and Kruskal's Algorithms

- Prim's algorithm is quite similar to the Dijkstra's algorithm for shortest paths (edge weights are used as priorities instead distances to source). The partial solution is always connected (it is a tree)
- the idea of the Kruskal's algorithm is very simple. The partial solution is not necessarily connected (it is a forest not a tree)
- Prim's Algorithm is a good choice in general case
- However, Kruskal's Algorithm (with fast implementation of union-find) can be faster on *sparse* graphs, where $m = O(n)$
- Kruskal's Algorithm can be used in a "streaming mode": i.e. the edges come on-line through a network connection, etc. Even if they are not sorted by weight it is possible to device a quite fast version of the algorithm. (e.g. with $O(mlog(n))$ time complexity and $O(n)$ space complexity)

Summary

- Minimum Spanning Tree (MST) Problem
- Cut Property and Cycle Property of MST
- Prim's Algorithm
- Kruskal's Algorithm
- Union-Find Abstract Data Structure
- (*) Fast Implementation of Union-Find

Thank you for attention