Approximation Algorithms Lecture 21

Contents

1 Recap

Lower bounds on approximation guarantees, f-approximation for set cover.

2 Content

2.1 Primal-dual algorithms

We'll look at some algorithms that use the primal-dual method.

2.2 Steiner forest problem

The steiner tree problem says this: find a minimum cost connected subgraph which includes all terminals (T).

The steiner forest problem says this: Given k pairs of terminals (s_i, t_i), find a minimum cost subgraph in which each pair s_i, t_i is connected.

The Steiner forest problem is a generalization of Steiner tree: if I have to find a Steiner tree over a set of terminals then this can be formulated as a Steiner forest problem by defining pairs $(t_1, t), \ldots, (t_k, t)$ where $\{t, t_1, \ldots, t_k\} = T$.

Let's formulate this as an integer program.

Integer program for Steiner forest

For edge $e \in E$, let x_e be defined as 1 if e is picked in the Steiner forest, and 0 otherwise.

The cost of the Steiner forest becomes $\sum_{e \in E} c_e \cdot x_e$, where c_e is the cost of edge e.

Now we need to capture the connectivity relations. Consider a set S which has s such that $\overline{S}$ has t. Then for s being connected to t, it is necessary that at least one of the edges between S and $\overline{S}$ should be in the Steiner forest. Call these edges $\delta(s)$.

Our condition becomes $\sum_{e \in \delta(S)} x_e \geq 1$, $\forall S : \exists i \ni (s_i, t_i)$ is separated by S.

This is clearly necessary. Suppose E' be the edges which have $x_e = 1$. Suppose E' does not form a Steiner forest. There exists i such that there is no path from s_i to t_i using edegs of E'. Consider the set of vertices S which are reachable from s_i using edges of E'. Then clearly, $\overline{S}$ has t_i, else there is a path from s_i to t_i. So S separates s_i and t_i. Also note that no edge of $\delta(S)$ has $x_e = 1$ (since we could have added another vertex into this set S). So this implies that $\sum_{e \in \delta(S)} x_e = 0$, and this is a violation of constraint, which is a contradiction.

Let $r : 2^V \to \{0, 1\}$ be a requirement function on sets where $r(s) = 1$ iff S separates an (s_i, t_i).

Now we will look at a LP-relaxation.

LP relaxation of IP

$$\min \sum_{e \in E} c_e x_e$$
subject to the constraints
$$\forall S \subseteq V, \sum_{e \in \delta(S)} x_e \geq r(S)$$
$$0 \leq x_e \leq 1$$

The $x_e \leq 1$ constraint is redundant, so we can drop this constraint.

Of course, this LP has exponentially many equations, so we won't actually solve this (but use this as a conceptual tool).

Consider the dual of this LP. Then we have the dual program as:

$$\max \sum_{S \in 2^V} r(S) \cdot y_S$$
subject to the constraints
$$\forall e \in E, \sum_{S : e \in \delta(S)} y_S \leq c_e$$
$$0 \leq y_S$$

Remember that the dual is written so that y_S should be such that for each variable x_e for the primal LP, coefficient of x_e in the linear combination should not exceed its coefficient in the objective function, i.e., c_e.

These constraints are called **packing constraints**. Let's consider the Steiner forest problem when we have only one pair (s_1, t_1). This is exact the shortest path problem.

In this setting, the primal LP would be the same. We can look at how Dijkstra works on this algorithm, and we'll eventually try to assign y_S using consecutive sets.

We are only interested in increasing the dual values of sets for which $r(S) = 1$.

Look at the primal and dual in unison. Then there are complementary constraints, and due to complementary slackness, at least one of the constraints must be tight in an optimal solution.

Definition 1

Tight edge: An edge is tight iff $\sum_{S:e\in\delta(S)} y_S = c_e$.

In the primal-dual algorithm, we increase dual values associated with sets. When an edge e is tight, we set $x(e) = 1$.

Definition 2

A set S is unsatisfied if $r(S) = 1$ and $\sum_{e\in\delta(S)} x(e) < 1$, else it is satisfied.

Definition 3

A set S is a minimal unsatisfied set if S is unsatisfied, and for all $A \subsetneq S$, A is satisfied.

Claim 0.1

The collection of minimal unsatisfied sets is disjoint. That is, if A, B are minimally unsatisfied, then they are disjoint.

Proof. Suppose not. Then $A \cap B \neq \emptyset$. So we have $r(A) = r(B) = 1$. Then we will show that either $r(A \cap B) = 1$ or $r(A \setminus B) = r(B \setminus A) = 1$.

Suppose A separates (s_i, t_i).

1. If s_i is in $A \setminus B$, then $A \setminus B$ separates this pair.

2. If s_i is in $A \cap B$, then $A \cap B$ separates this pair.

Now look at tight edges going across these sets.

1. No tight edge goes across $A \cap B$ (case analysis).
2. No tight edge goes across $A \setminus B$ (case analysis).

This implies that if $r(A \cap B) = 1$, then $A \cap B$ is also an unsatisfied edge, and if $r(A \setminus B) = r(B \setminus A) = 1$ then $A \setminus B$ and $B \setminus A$ are also unsatisfied sets, which is a contradiction to minimality. $\square$

We call minimal unsatisfied sets active sets.

Algorithm:

1. Increase the dual variables of these active sets at the same rate till some edge becomes tight.
2. Include tight edges in primal solution. Recompute minimal unsatisfied sets.
3. Repeat these sets until there are no more unsatisfied sets. [End of phase 1]
4. In phase 2, we have a pruning phase in which some edges included in the primal solution which are redundant are dropped.

Let's see the example of the min spanning tree, and see the algorithm in action there.

The least cost edge gets tight at time t = cost of edge/2.

The k^{th} edge gets tight at a similar time. Note that this is slightly wrong – we won't make anything tight which is inside a "component" since we raise the dual of a set with requirement 1.

So the components are forests, and we pick edges which go across components in increasing order, which is precisely Kruskal's algorithm.

Let's come back to the original problem.

Consider the connected components of the forest at any time in the algorithm. A connected component, if it is unsatisfied, is a minimal unsatisfied set, and we increase the dual corresponding to it.

Minor remark: the half-cost time remark doesn't hold for general examples since there might be sets which don't get dual increased at all.

Let F be the set of tight edges after phase 1.

Now we come to the pruning phase: remove redundant edges. For that, consider the edges in the reverse order in which they get tight. Remove an edge e from F if $F \setminus \{e\}$ is also a steiner forest.

Let F' denote the final set of edges.

Claim 0.2

$\sum_{e \in F'} \leq 2 \sum_{S \subset V} y_S$

Proof.

Note 1

Why we want to prove this: dual feasible sol $\leq$ opt lp sol $\leq$ opt ip sol $\leq$ steiner forest.

Why this is tight: G is a cycle on n vertices and each edge has length 1. Every pair of vertices needs to be connected. Optimal integral solution is $n-1$ - by spanning tree. Giving $1/2$ to each edge gives an optimal LP solution with cost $n/2$, since sum of x_e is 1. This is optimal (by adding all inequalities). Another way is to see the dual. Set the value of any singleton set to $1/2$ and for every other set, use 0.

The chosen set of edges forms a Steiner forest, so it is indeed a feasible solution.

Since $e \in F'$ is tight, $c_e = \sum_{S : e \in \delta(S)} y_S$.

So we need to prove: $\sum_{e \in F'} \sum_{S : e \in \delta(S)} y_S \leq 2 \sum_{S \subset V} y_S$.

Interchange of summation transforms this into $\sum_{S \subset V} y_S \cdot |\delta(S) \cap F'| = \sum_{S \subset V} y_S \cdot \deg_{F'}(S) \leq 2 \sum_{S \subset V} y_S$.

An iteration is the period of time between two consecutive edges going tight.

In the i-th iteration, the increase in LHS is $\Delta \cdot \sum_{S \text{ active in this iteration}} \deg_{F'}(S)$.

Increase in RHS $= 2 \cdot$ number of active sets in this iteration $\cdot \Delta$. So if the increase in LHS is at most the increase in RHS, we'll be done.

Edges of F' form a forest on the connected components (both active and inactive sets) in this iteration.

Consider a tree in the forest. If all vertices (sets) were active, then $\sum_{S \text{ active}} \deg'_F(S) = 2 \cdot (\text{active sets} - 1)$. We are interested in the average degree of only the active vertices. If every inactive vertex has degree ≥ 2 (i.e., all leaves are active), we'll be done.

Claim 0.3

There is no inactive leaf.

Proof. Suppose in iteration i, a leaf in one of the trees (formed by edges of F' over the connected components of this iteration) is inactive.

If the edge joining the inactive set to its parent was not deleted, then its deletion would cause a pair (s_i, t_i) to get disconnected. So the set corresponding to the leaf is separating a pair of terminals (s_i, t_i) and should have requirement 1 which implies that this set is not inactive at iteration i. □

□

Implementation:

We'll implement it in a manner similar to Kruskal's algorithm.

1. Phase 1:

   - Maintain sets S_i in a union-find data structure.
   - In each iteration, find the edge that gets tight first. For this, if next event happens after time t, then reduce the cost of every edge incident to exactly one active set by t, and that incident to two active sets by $2t$. So we maintain reduced costs $rc[e]$.

   - Maintain reduced costs: array which contains reduced cost of each edge, and set $s_1, \ldots, s_k$ corresponding to connected components $C_1, \ldots, C_k$. With each set we have information whether the set is active or not.

   - For every edge $e = (u, v)$, let S_i, S_j be the sets containing u, v respectively. If both are the same, then skip.

   - If both are active, then next event time = min(next event time, $rc[e]/2$), else if one is active, next event time = min(next event time, $rc[e]$).

   - Update reduced costs: $rc[e] \leftarrow rc[e] -$ next event time / c, where $c = 1$ if exactly one is active, and 2 if both active.

   - For the edge whose reduced cost became 0, add it, $e = (u, v)$, to F, and join the two components.

   - Now check if this newly created component is active or not. To do this, check if there is an i such that (s_i, t_i) has exactly one of them in S. If there is, then S is active, otherwise it is inactive.

In an iteration, time taken is $O(|E| + k +$time for union). Excluding time for unions, we can do it in $O(n(m+k))$. Represent each set as a linked list, with every node having a pointer to head (the representative of that set). For find, it's easy (go to head). For union, join smaller set to the larger set. Time for union is $O(\min(|S_1|, |S_2|))$.

Overall this is $O(n \log n)$, since the pointer corresponding to each vertex is modified at most $\log_2 n$ times, since every time it is modified, the size of the set to which the set belongs at least doubles.

So the total time taken is $O((m + k + \log n)n)$.