

## Lecture 13: Seidel's Randomized Incremental Construction for Linear Programming

Lecturer: Suresh Venkatasubramanian

Scribe: Pankaj Nathani

### 13.1 Randomized Incremental Construction (RIC) for Linear Programming

#### 13.1.1 Comparison with Previous Work

There are two algorithms by Megiddo to solve linear programming with complexities  $O(2^{2^d} n)$  and  $O(n(\log n)^{d^2})$ . Megiddo showed that if the number of variables in a linear program ( $d$ ) is considered a constant then the linear program can be solved in time that is linear in the number of its constraints ( $n$ ). These algorithms were extremely complex and unfortunately the running time of these algorithms depended on  $d$  in a superexponential way.

Clarkson and Dyer improved the complexity to the order of  $O(n3^{d^2})$ . In 1989, Clarkson came up with a randomized algorithm with running time of  $O(d^2) + (\log n)O(d)^{d/2+O(1)} + O(d^4\sqrt{m}\log m)$ . The algorithm is relatively straightforward but the analysis of its expected running time is bit complex.

Seidel came up with an exceedingly simple randomized linear programming algorithm whose expected running time is  $O(d!n)$ . This bound is worse than that of Clarkson but the analysis of its expected running time is very simple and so is the algorithm.

### 13.1.2 Basics of RIC

Lets write a widget for RIC:

#### Iterative View

MakeWidget( $S$ )

1. Randomly permute points
2. Insert one by one
3. Update structure

#### Recursive View

MakeWidget( $S$ )

1. Randomly remove one item  $s \in S$
2. MakeWidget( $S - \{s\}$ )
3. Re-insert  $s$

Effectively all the items are first removed and than added to the list in the reverse order:

$s_1, s_2, \dots, s_{n-1}, s_n$

→ Order of removing

← Order of inserting back

## 13.2 Seidel's RIC algorithm

### 13.2.1 Problem Statement

GIVEN: A set  $\mathcal{H}$  of  $n$  halfspaces and a vector  $c \in \mathbb{R}^d$

FIND: An "optimum vertex"  $v$  of the polyhedron  $P_{\mathcal{H}}$  formed by the intersection of the halfspaces in  $\mathcal{H}$  so that  $v$  maximizes the linear functional specified by  $c$ ; in other words,  $v$  must be contained in the tangent hyperplane of  $P_{\mathcal{H}}$  whose outward normal is  $c$ .

### 13.2.2 Algorithm

SEIDELLP( $d, n$ )

- 1: Randomly choose  $H$  from the  $n$  halfspaces in  $\mathcal{H}$  to obtain the set  $\mathcal{H}'$ .
2. Recursively compute the optimum vertex  $v'$  of  $P_{\mathcal{H}'}$  (with respect to the direction  $c$ ). In other words, recursively call SEIDELLP( $d, n - 1$ ).  $n - 1$  because one halfspace has been removed from the earlier set of  $n$  halfspaces.
3. We compute  $v$  from  $v'$  as follows:

If  $v'$  is contained in the halfspace  $H$  (i.e. halfspace which was removed in step 1)

**then**

$v'$  satisfies all the halfspaces and thus  $v = v'$  and we are done.

**else**

$v$  must be contained in the hyperplane  $h$  of the hyperspace  $H$ . We will compute  $\overline{\mathcal{H}'}$  by projecting  $\mathcal{H}'$  on hyperplane  $h$  (i.e.  $\overline{\mathcal{H}'} = \{G \cap h \mid G \in \mathcal{H}'\}$ ). Now  $v$  will be the optimum vertex of the  $(d - 1)$ -dimensional linear program specified by the halfspaces  $\overline{\mathcal{H}'}$  and the direction  $\bar{c}$  where  $\bar{c}$  is the orthogonal projection of  $c$  into  $h$  and  $\overline{\mathcal{H}'}$ .

Thus,  $v$  can be determined by recursively solving a  $(d - 1)$ -dimensional problem with  $n - 1$  constraints. In other words, recursively call SEIDELLP( $d - 1, n - 1$ ).

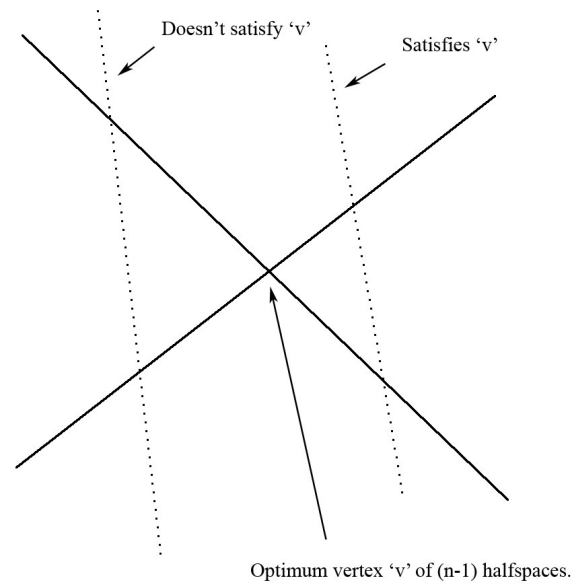


Figure 13.1: Overview of algorithm

### 13.2.3 Analysis of algorithm

**Theorem 1 (Complexity)** *Using the randomized method outlined in above section a linear program with  $n$  constraints in  $d$  variables can be solved in expected time  $O(d!n)$ .*

**Proof:**

Let  $T(d, n)$  be time required to solve Linear programming with  $d$  variables and  $n$  constraints.

Making few assumptions:

1. Testing whether a point is contained in a halfspace takes  $O(d)$  time.
2. Determining the intersection of a halfspace in  $\mathbb{R}^d$  with a hyperplane takes  $O(d)$  time. Calculation of  $\overline{\mathcal{H}'}$  needs determining intersection of  $n - 1$  halfspaces with hyperplane  $h$ . Therefore, calculation of  $\overline{\mathcal{H}'}$  takes  $O(dn)$ .

By now we have several parts of the algorithm except for the probability  $P$  of  $v'$  not satisfying halfspace  $H$ . In other words  $P$  is the probability of  $v$  and  $v'$  being different. Optimal point is defined by  $d$  halfspaces.  $v$  and  $v'$  can be different only if one of the  $d$  halfspaces whose bounding hyperplanes define  $v$  is  $H$ . Since  $H$  was chosen from the  $n$  halfspaces in  $\mathcal{H}$  uniformly at random, it follows that  $v$  is different from  $v'$  with probability  $d/n$ .

Plugging time complexities of sub-tasks in the main algorithm:

$$\begin{aligned}
T(d, n) = & \text{Time to solve a } d\text{-dimensional problem with } n - 1 \text{ constraints} + \\
& + \text{Time to test whether point is contained in halfspace } H + \\
& + \text{Probability } P \text{ of point not contained in halfspace } H * \\
& \left( \text{Time to determine intersection of } n - 1 \text{ halfspaces with hyperplane } h + \right. \\
& \left. + \text{Time to solve a } d - 1\text{-dimensional problem with } n - 1 \text{ constraints} \right) \quad (13.1)
\end{aligned}$$

$$= T(d, n - 1) + O(d) + \frac{d}{n} (dn + T(d - 1, n - 1)) \quad (13.2)$$

$$= T(d, n - 1) + O(d) + d^2 + \frac{d}{n} (T(d - 1, n - 1)) \quad (13.3)$$

$$\begin{aligned}
= & \left( T(d, n - 2) + O(d) + d^2 + \frac{d}{n - 1} (T(d - 1, n - 2)) \right) + \\
& + O(d) + d^2 + \frac{d}{n} (T(d - 1, n - 1)) \quad (13.4)
\end{aligned}$$

$$= nd^2 + d \sum_{i=1}^{n-1} \frac{T(d - 1, i)}{i + 1} \quad (\text{Expanding and eliminating } O(d) \text{ terms}) \quad (13.5)$$

$$= nd^2 + d \sum_{i=1}^{n-1} \frac{c^{d-1}(d-1)!i}{i+1} \quad (\text{Guessing the solution as } c^d d!n) \quad (13.6)$$

$$= nd^2 + c^{d-1} d! \sum_{i=1}^{n-1} \frac{i}{i+1} \quad (13.7)$$

$$\leq nd^2 + c^{d-1} d! \sum_{i=1}^{n-1} 1 \quad (13.8)$$

$$\leq nd^2 + c^{d-1} d!(n - 1) \quad (13.9)$$

$$\leq nd^2 + c^{d-1} d!n \quad (13.10)$$

$$\leq c' c^{d-1} d!n + c^{d-1} d!n \quad (nd^2 \text{ is asymptotically lower than } c^{d-1} d!n) \quad (13.11)$$

$$\leq (c - 1) c^{d-1} d!n + c^{d-1} d!n \quad (\text{Let } c' = c - 1) \quad (13.12)$$

$$\leq c^d d!n \quad (13.13)$$

We initially guessed the solution as  $c^d d!n$  and finally reached the same solution.

Hence, it is proved that Seidel's randomized algorithm takes  $O(d!n)$  ■

## 13.3 Application of Linear Programming

Now we will see a beautiful application of linear programming. Linear programming can be used to formulate an approximation algorithm for Vertex Cover.

### 13.3.1 Vertex Cover

A vertex cover of an undirected graph  $G = (V, E)$  is a subset  $V'$  of the vertices of the graph which contains at least one of the two endpoints of each edge:

$$V' \subseteq V : \forall \{a, b\} \in E : a \in V' \cup b \in V'$$

### 13.3.2 Vertex Cover as an Integer Program

Let us formulate a linear program that is in close correspondence with the Vertex Cover Problem. Linear programming is based on the use of vector of variables. In our case, we will have a *decision variable*  $x_i$  for each vertex  $i \in V$  to model the choice of whether to include vertex  $i$  in the vertex cover.  $x_i = 0$  will indicate that vertex  $i$  is not in the vertex cover, and  $x_i = 1$  will indicate that vertex  $i$  is in the vertex cover.

We will use linear inequalities to encode the requirement that the selected vertices form a vertex cover; for each edge  $(i, j) \in E$ , it must have one end in the vertex cover, and we write this as the inequality  $x_i + x_j \geq 1$ .

We will use an objective function to encode the goal of minimizing the total number of vertices chosen; the minimization problem can be expressed as minimizing the sum of *decision variable*  $x_i$ .

In summary, Vertex Cover Problem can be formulated in linear programming as:

$$\begin{aligned} (\text{VCIP}) \quad & \text{Min} \sum_{i \in V} (x_i) \\ \text{s.t.} \quad & x_i + x_j \geq 1 \quad (i, j) \in E \\ & x_i \in \{0, 1\} \quad i \in V. \end{aligned}$$

Vertex Cover of graph  $G$  is in one-to-one correspondence with the solutions of this system of linear inequalities in which  $x_i$  is equal to 0 or 1.

By above formulation, we have reduced Vertex Cover Problem to Integer Programming Problem in polynomial time:

$$\text{Vertex Cover} \leq_P \text{Integer Programming}$$

Since Vertex Cover is known to be NP-hard problem, Integer Programming is also NP-hard.

### 13.3.3 LP Relaxation of Vertex Cover

Integer programming imposes  $x_i$  to take either 0 or 1. We can relax this requirement by allowing  $x_i$  to be arbitrary real numbers between 0 and 1. Relaxation of NP-hard Integer Programming formulation of Vertex Cover to Linear Programming helps in solving Vertex Cover in polynomial time.

All other parts of above integer programming formulation remain same. Therefore, Vertex Cover Problem can be relaxed to Linear Programming as:

$$\begin{aligned}
 (\text{VC}_{\text{LP}}) \quad & \text{Min} \sum_{i \in V} (x_i) \\
 \text{s.t.} \quad & x_i + x_j \geq 1 \quad (i, j) \in E \\
 & x_i \geq 0 \quad i \in V.
 \end{aligned}$$

### 13.3.4 Using Linear Programming for Vertex Cover

We should not solve Vertex Cover using Integer Programming because Integer Programming has been proved to be NP-hard in the above section. We can exploit the fact that Linear Programming is not as hard as Integer Programming. Above section also shows that initial Integer Programming formulation can be relaxed to Linear Programming. Thus, we can solve Vertex Cover using Linear Programming.

**Theorem 2 (Bounds)**  $\text{VC}_{\text{LP}} \leq \text{VC}_{\text{IP}}$

**Proof:** As we said earlier, Vertex cover of  $G$  corresponds to integer solutions to  $\text{VC}_{\text{IP}}$ , so the minimum of  $\text{Min} \sum (x_i)$  over all integer  $x_i$  is exactly the minimum Vertex Cover. To get the minimum of the linear program  $\text{VC}_{\text{LP}}$ , we allow  $x_i$  to take arbitrary real-number values (i.e, we minimize over many more choices of  $\mathbf{x}$ ) and so the minimum of  $\text{VC}_{\text{LP}}$  is no larger than that of  $\text{VC}_{\text{IP}}$ . ■

With integer programming, it looks natural that if  $x_i = 1$  for some vertex  $i$ , then we should put it in the vertex cover and if  $x_i = 0$ , then we should leave out of vertex cover. With linear programming,  $x_i$  can have any real number between 0 and 1 (for ex- 0.4). We can employ *rounding*. Therefore, given a fractional solution  $\{x_i\}$ , Vertex Cover  $VC = \{i \in V : x_i \geq \frac{1}{2}\}$  (i.e.,  $x_i \geq \frac{1}{2}$  are rounded up to 1 and are included in the Vertex Cover and  $x_i < \frac{1}{2}$  are rounded down to 0 and are not included in the Vertex Cover. Let  $\text{VC}_S$  be vertex cover defined this way

**Theorem 3 (Bounds)**  $\text{VC}_S \leq 2\text{VC}_{\text{LP}}$

**Proof:** Consider an edge  $e = (i, j)$  of graph  $G$ . At least one of either  $i$  or  $j$  must be in the vertex cover. Recall that one of our inequalities is  $x_i + x_j \geq 1$ . So in any solution that satisfies this inequality, either  $x_i \geq \frac{1}{2}$  or  $x_j \geq \frac{1}{2}$ . Thus at least one of these two will be rounded up, and  $i$  or  $j$  will be placed in the vertex cover. Since there is a factor of 2 (jumping from  $\frac{1}{2}$  to 1),  $\text{VC}_S \leq 2\text{VC}_{\text{LP}}$ . ■

Thus, we have produced a vertex cover  $\text{VC}_S$  which is at most twice of  $\text{VC}_{\text{LP}}$ . Theorem 2 also proves that the optimal vertex cover  $\text{VC}_{\text{IP}}$  is at least  $\text{VC}_{\text{LP}}$ . **Therefore, linear programming produces a vertex cover which is at most twice the optimal vertex cover.**

## 13.4 Duality of Linear Programming

Dual of a typical linear program:

$$\begin{array}{ll} \text{Min} & cx \\ \text{s.t.} & Ax \geq b \\ & x \geq 0 \end{array}$$

will be

$$\begin{array}{ll} \text{Max} & y^T A \\ \text{s.t.} & y^T A \leq c \\ & y \geq 0 \end{array}$$

Lets formulate dual of linear programming for Vertex Cover.

We will have a *decision variable*  $y_{e_i}$  for each edge  $i \in E$ .

$$\begin{array}{ll} \text{Max} & y_{e_i} \\ \sum_{v \in e} (y_{e_i}) & \leq 1 \quad \forall v \in V \\ y_{e_i} & = \{0, 1\} \end{array}$$

Above formulation basically solves *Maximum Cardinality Matching* Problem and it is not NP-hard. A *matching* of a graph  $G = (V, E)$  is a set of edges  $M \subseteq E$  such that number of edges incident to any vertex is at most one.

We started with NP-hard problem in primal space and it is no longer a NP-hard problem in dual space. The reason is Integer Programming does not have primal-dual relationship whereas Linear Programming has primal-dual relationship.

[1] R. Seidel, Linear programming and convex hulls made easy, Proc 6th Ann. ACM Symp. Computational Geometry, pages 211-215, 1990.

[2] J. Kleinberg, E. Tardos, Algorithm Design, Pearson Education Inc, 2006.