

Lecture 8: 2.12.05

Lecturer: Christos

Scribe: Tanya Roosta, Sumitra Ganesh

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

8.1 Efficient Algorithms

The standard algorithm for solving LP problems is the simplex algorithm. The problem with this algorithm is that it not known bo run in polynomial time. In fact some implementations can run in exponential time. Ellipsoid algorithm is another algorithm designed for LP which runs in polynomial time.

8.1.1 Seidel's Algorithm

Seidel's algorithm is a simple, randomized algorithm which runs in polynomial time. Given the linear program $\max Cx$

st. $Ax \leq b$

with d variables and n constraints, and $d \leq n$.

1. **if** there is only one variable **then**
2. Treat all inequalities in $Ax \leq b$ as equalities and solve for x in each.
3. Return the point out of these that maximizes Cx
4. **end if**
5. **if** there are as many constraints as variables **then**
6. Solve the linear system $Ax = b$ using Gaussian elimination and return the resulting point.
7. **end if**
8. Pick a constraint $h : a_i x \leq b_i$ uniformly at random
9. $x^* \leftarrow \text{Seidel}(A - a_i, b - b_i, C)$
10. **if** x^* satisfies h **then**
11. Return x^*
12. **else**
13. $A', b' \leftarrow$ substitution of the euqlity $a_i x = b_i$
14. Return $\text{Seidel}(A', b', C)$
15. **end if**

8.1.2 Correctness of Seidel's Algorithm

We want to choose a corner from among the $\binom{n}{d}$ choices. Lines 1 through 7 solve the simple cases when there is only one variable or as many constraints as variables. Otherwise, the code recurses until it reaches one of the following cases.

Line 11 will return correct values which can be proven by induction. We assume that x^* is the best solution without the chosen h . Including h again can only restrict the solution space, but x^* was verified to satisfy the constraint h , so it must be the solution to the original problem. We can see this from figure 8.1.

If the program reaches line 14, then x^* did not satisfy h . This means h is one of the "important" tight constraints that determines the solution. Therefore, the solution must lie on h 's hyperplane, i.e. $a_i = b_i$ for the true solution x . This equation may be simplified to an equation with one variable, and after substitution it will lower the dimensionality of the problem. Given the new system of inequalities is equivalent to the old system, by induction we can prove that the returned value from the recursion in the algorithm is correct. Figure 8.2 illustrates this case.

8.1.3 Running Time of the Algorithm

The running time of this randomized algorithm is defined in terms of the expected running time. Assume $T(n, d)$ is the expected running time of Seidel's algorithm with d variables and n constraints. The algorithm always makes a recursive call that results in a system with smaller dimension. This recursive call happens even when the constraint chosen is "tight". Therefore, with d tight constraints there is a d/n probability that this recursive call is made. The recursive calls give the following formula:

$$T(n, d) = T(n - 1, d) + d/nT(n, d - 1)$$

After solving this equation, we get the running time of the algorithm to be $T(n, d) = O(d!n)$

Therefore, the expected running time of the algorithm is linear if we fix d .

8.2 Recurrences

8.2.1 An Example

In quick sort, we partition a list into two sublists, sort each sublist and then merge the results. The merging takes $O(n)$. The recurrence algorithm for this example is the following:

$$T(n) = 2T(n/2) + n$$

This solves in $O(n \cdot \log(n))$.

Another example of using the recurrence to solve a problem more efficiently is the multiplication of two n bit numbers. Let $A = a2^{n/2} + b$ and $B = c2^{n/2} + d$ be two n bit numbers. Instead of performing four multiplications, we can use the following clever way:

$$w = (a + b)(c + d)$$

$$u = ac$$

$$v = bd$$

Therefore, $ac + bd = w - u - v$ and $T(n) = 3T(n/2) + \theta(n)$, and the multiplication has a complexity of $O(n^{\log_2 3})$.

8.2.2 Divide and Conquer and The Master Theorem

Divide and conquer means breaking the problem down into two or more smaller subproblems. The procedure that we used in the above example can be generalized to the recurrence of the form $T(n) = aT(n/b) + n$ for any a, b . We can get three different forms for divide and conquer as follows:

$a < b$: runs in linear time in n , example median

$a > b$: runs in $n^{\log_b a}$, example integer multiplication

$a = b$: runs in $n \log n$, example FFT

For the general description of the formula, we have a branching factor of a with n/b^d branching at a given level d . Therefore, we get the following formula for divide and conquer:

$$T(n) = n \sum_{i=0}^{\log_b n} (a/b)^i$$

This leads us to the next important result, ie. The Master Theorem.

Theorem 8.2.1 Suppose $T(n)$ satisfies the recurrence relation:

$$T(n) = \begin{cases} aT(n/b) + \Theta(n^\alpha), & \text{if } n > 1; \\ \theta(1), & \text{otherwise.} \end{cases}$$

Then,

$$T(n) = \begin{cases} \theta(n^\alpha), & \text{if } \alpha > \beta; \\ \theta(n^\alpha \log_b n), & \text{if } \alpha = \beta; \\ \theta(n^\beta), & \text{if } \alpha < \beta. \end{cases}$$

where $\beta = \log_b a$.

Let $T(n)$ be the cost of solving a problem with input size n . The problem is divided into a subproblems each of size (n/b) . Each of these subproblems is solved. The solutions to these subproblems are then merged to yield the solution to the initial problem.

Let the cost of dividing the problem and merging the solutions be $\theta(n^\alpha)$. By definition of $T(n)$, the cost of solving the a subproblems is $a * T(n/b)$.

Adding the above two costs, we get the first equation in the recurrence relation for $T(n)$. The base case equation follows from the fact that it takes constant time to solve a problem of unit size input.

For the divide and conquer we can imagine having a recursion tree:

At the root:

- 1 Number of problems = 1
- 2 size of problem = n
- 3 work done at the root level = n^α

In the second level of the tree:

- 1 Number of problems = a
- 2 size of problem = n/b
- 3 work done at this level = $(n/b)^\alpha * a$

In the i^{th} level of the tree:

- 1 Number of problems = a^i
- 2 size of problem = n/b^i
- 3 work done at this level = $(n/b^i)^\alpha * a^i$

Total number of stages in the recursion = $\log_b n$

- 1 Number of problems at the leaf level = $a^{\log_b n} = n^{\log_b a}$
- 2 work done at the leaf level = $\theta(n^{\log_b a})$

If the cost of decomposing the problem and merging the subsolutions dominates the cost of solving the subproblems, the cost is mainly determined by the cost in the first stage.

If the cost of solving the subproblems dominates the cost of dividing the problem and merging the subsolutions, the cost is determined mainly by the cost of the leaf stage.

If the cost of breaking up the problem and merging the subsolutions is comparable to the cost of solving all the subproblems, then the cost is determined by the entire tree.

More formally, summing up the costs for each of the stages,

$$\text{Total Cost } C = N^\alpha + n^\alpha * (a/b^\alpha) + n^\alpha * (a/b^\alpha)^2 + \dots + n^\alpha * (a/b^\alpha)^i + \dots + n^\alpha * (a/b^\alpha)^{\log_b n}$$

Substituting $\beta = \log_b a$, and $b^{\beta-\alpha} = r$,

$$C = n^\alpha * (1 + r + r^2 + \dots + r^i + \dots + r^{\log_b n})$$

This is a geometric progression, and the sum is:

$$C = n^\alpha * \frac{r^{\log_b n + 1} - 1}{r - 1}$$

$$\text{Thus, } C = \begin{cases} n^\alpha, & \text{if } \alpha > \beta; \\ n^\alpha * (a/b^\alpha)^{\log_b n} = n^{\log_b a}, & \text{if } \alpha = \beta; \\ n^\alpha * \log_b n, & \text{if } \alpha < \beta. \end{cases}$$

References

- [1] <http://www.cs.berkeley.edu/~chrisht/teaching/fall2001-cs270/>
- [2] CLR

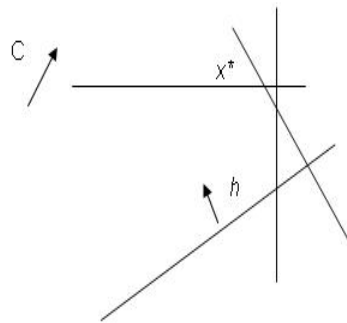


Figure 8.1: Case 1