# Advanced Algorithms
# Lecture Notes

Periklis A. Papakonstantinou

Fall 2011

# Preface

This is a set of lecture notes for the 3rd year course "Advanced Algorithms" which I gave in Tsinghua University for Yao-class students in the Spring and Fall terms in 2011. This set of notes is partially based on the notes scribed by students taking the course the first time it was given. Many thanks to Xiahong Hao and Nathan Hobbs for their hard work in critically revising, editing, and adding figures to last year's material. I'd also like to give credit to the students who already took the course and scribed notes. These lecture notes, together with students' names can be found in the Spring 2011 course website.

This version of the course has a few topics different than its predecessor. In particular, I removed the material on algorithmic applications of Finite Metric Embeddings and SDP relaxations and its likes, and I added elements of Fourier Analysis on the boolean cube with applications to property testing. I also added a couple of lectures on algorithmic applications of Szemerédi's Regularity Lemma.

Except from property testing and the regularity lemma the remaining topics are: a brief revision of the greedy, dynamic programming and network-flows paradigms; a rather in-depth introduction to geometry of polytopes, Linear Programming, applications of duality theory, algorithms for solving linear programs; standard topics on approximation algorithms; and an introduction to Streaming Algorithms.

I would be most grateful if you bring to my attention any typos, or send me your recommendations, corrections, and suggestions to papakons@tsinghua.edu.cn.

*Beijing,*
*Fall 2011*

PERIKLIS A. PAPAKONSTANTINOU

*– This is a growing set of lecture notes. It will not be complete until mid January 2012 –*

# Contents


**1 Big-O Notation** **4**
1.1 Asymptotic Upper Bounds: big-O notation . . . 4

**2 Interval Scheduling** **6**
2.1 Facts about Greedy Algorithms . . . 6
2.2 Unweighted Interval Scheduling . . . 7
2.3 Weighted Interval Scheduling . . . 10

**3 Sequence Alignment: can we do Dynamic Programming in small space?** **12**
3.1 Sequence Alignment Problem . . . 12
3.2 Dynamic Algorithm . . . 13
3.3 Reduction . . . 14

**4 Matchings and Flows** **17**
4.1 Introduction . . . 17
4.2 Maximum Flow Problem . . . 18
4.3 Max-Flow Min-Cut Theorem . . . 19
4.4 Ford-Fulkerson Algorithm . . . 20
4.4.1 Edmonds-Karp Algorithm . . . 21
4.5 Bipartite Matching via Network Flows . . . 24
4.6 Application of Maximum Matchings: Approximating Vertex Cover . . . 24

**5 Optimization Problems, Online and Approximation Algorithms** **26**
5.1 Optimization Problems . . . 26
5.2 Approximation Algorithms . . . 27
5.3 Makespan Problem . . . 27
5.4 Approximation Algorithm for Makespan Problem . . . 28

**6 Introduction to Convex Polytopes** **31**
6.1 Linear Space . . . 31
6.2 Affine Space . . . 31
6.3 Convex Polytope . . . 33
6.4 Polytope . . . 34
6.5 Linear Programming . . . 35

**7 Forms of Linear Programming** 36
7.1 Forms of Linear Programming . . . 36
7.2 Linear Programming Form Transformation . . . 37

**8 Linear Programming Duality** 38
8.1 Primal and Dual Linear Program . . . 38
8.1.1 Primal Linear Program . . . 38
8.1.2 Dual Linear Program . . . 39
8.2 Weak Duality . . . 40
8.3 Farkas' Lemma . . . 41
8.3.1 Projection Theorem . . . 41
8.3.2 Farkas' Lemma . . . 41
8.3.3 Geometric Interpretation . . . 42
8.3.4 More on Farkas Lemma . . . 43
8.4 Strong Duality . . . 43
8.5 Complementary Slackness . . . 44

**9 Simplex Algorithm and Ellipsoid Algorithm** 47
9.1 More on Basic Feasible Solutions . . . 47
9.1.1 Assumptions and conventions . . . 47
9.1.2 Definitions . . . 47
9.1.3 Equivalence of the definitions . . . 48
9.1.4 Existence of Basic Feasible Solution . . . 49
9.1.5 Existence of optimal Basic Feasible Solution . . . 50
9.2 Simplex algorithm . . . 50
9.2.1 The algorithm . . . 50
9.2.2 Efficiency . . . 51
9.3 Ellipsoid algorithm . . . 51
9.3.1 History . . . 51
9.3.2 Mathematical background . . . 52
9.3.3 LP,LI and LSI . . . 53
9.3.4 Ellipsoid algorithm . . . 53

**10 Max-Flow Min-Cut Through Linear Programming** 55
10.1 Flow and Cut . . . 55
10.1.1 Flow . . . 55
10.1.2 An alternative Definition of Flow . . . 56
10.1.3 Cut . . . 56
10.2 Max-flow Min-Cut Theorem . . . 56

**11 Rounding Technique for Approximation Algorithms(I)** 60
11.1 General Method to Use Linear Programming . . . 60
11.2 Set Cover Problem . . . 60
11.2.1 Problem Description . . . 60
11.2.2 Complexity . . . 61

11.2.3 Greedy Set Cover . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
11.2.4 LP-rounding Set Cover . . . . . . . . . . . . . . . . . . . . . . . . . . 65
11.3 Vertex Cover . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67

**12 Rounding Technique for Approximation Algorithms(II) 68**
12.1 Randomized Algorithm for Integer Program of Set Cover . . . . . . . . . . . . . 68
12.1.1 Algorithm Description . . . . . . . . . . . . . . . . . . . . . . . . . . 68
12.1.2 The correctness of the Algorithm . . . . . . . . . . . . . . . . . . . . 68
12.2 Method of Computation Expectations . . . . . . . . . . . . . . . . . . . . . . 70
12.2.1 MAX-K-SAT problem . . . . . . . . . . . . . . . . . . . . . . . . . . 70
12.2.2 Derandomized Algorithm . . . . . . . . . . . . . . . . . . . . . . . . 71
12.2.3 The proof of correctness . . . . . . . . . . . . . . . . . . . . . . . . . 71

**13 Primal Dual Method 73**
13.1 Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
13.2 Set Cover Problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74

# Chapter 1

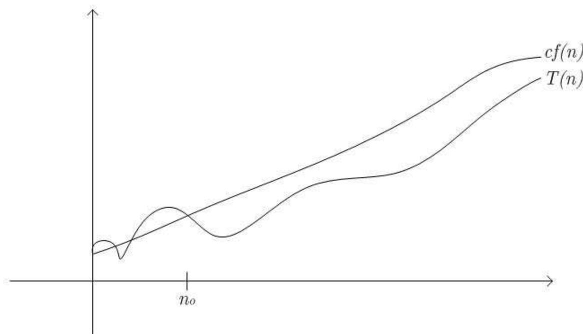
# Big-O Notation

*When talking about algorithms (or functions or programs), it would be nice to have a way of quantifying or classifying how fast or slow that algorithm is, how much space it uses and so on. It is useful to present this quantification in relation to the size of the algorithm's input, so that it is independent of the peculiarities of the problem. In this chapter we introduce the big-O notation which among other things it can be used to quantify the* asymptotic *behavior of an algorithm as a function of its input length $n$ measured in bits.*

## 1.1 Asymptotic Upper Bounds: big-O notation

Let $T(n)$ be a function, say, the worst-case running time of a certain algorithm with respect to its input length $n$. What we would like to point out here is that worst-case running time is a well-defined function. Contrast this to the running time of an algorithm which is in general not a function of the input length (why?). Given a function $f(n)$, we say that $T(n) \in O(f(n))$ if, for sufficiently large $n$, the function $T(n)$ is bounded from above by a constant multiple of $f(n)$. Sometimes this is written as $T(n) = O(f(n))$.

To be precise, $T(n) = O(f(n))$, if there exists a constant $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, we have $T(n) \leq cf(n)$. In this case, we will say that $T$ is $asymptoticly\ upper\ bounded$ by $f$.




For a particular algorithm, different inputs result in different running times. Notice that the

running time of a *particular* input is a constant, not a function. Running times of interest are the best, the worst, and the average expressing that the running time is at least, at most and on average for each input length $n$. For us the worst-case execution time is often of particular concern since it is important to know how much time might be needed in the worst case to guarantee that the algorithm will always finish on time. Similarly, for the worst-case space used by an algorithm. From this point on unless stated otherwise the term "running time" is used to refer to "worst-case running time".

**Example.** Take *Bubble Sort* as an example, shown in Algorithm 1. This is an algorithm correct for the SORTING problem, which has the obvious definition. We see that the number of outer loops and the number of inner loops together determine the running time of the algorithm. Notice that we have:

$$the\ number\ of\ outer\ loops \leq n$$

$$the\ number\ of\ inner\ loops \leq n$$

Therefore, a rough estimation on the worst-case running time is $O(n^2)$. This bound can be proved to be asymptotically tight.

**Algorithm 1:** Bubble Sort

```
input  : L: list of sortable items
output: L': list of sorted items in increasing order
repeat
    swapped = false;
    for i ← 1 to length(L) - 1 do
        if L[i − 1] > L[i] then
            swap( L[i-1], L[i] )
            swapped = true
        end
    end
until not swapped;
return L
```

An important skill required in the design of combinatorial algorithms is constructing examples. In this case, infinite families of examples. The more interesting direction in bounding the running time is what we did before, i.e. to *upper bound* it. We showed that $T(n) = O(n^2)$. How do we know we have not overestimated things? One way is to show that $T(n) = \Omega(n^2)$. To do that, we construct an *infinite family of examples parametrized by* $n$ where the algorithm takes $\Omega(n^2)$. In this case, given $n$ we can consider as input $\langle n, n-1, \ldots, 1\rangle$ and conclude that on this input Algorithm 1 makes $\Omega(n^2)$ many steps[1].

[1] To be very precise we should have treated $n$ as the input length in the number of bits in the input (and not the number of integers). For example, the above input for a given $n$ has length $\approx n \log n$ (each integer requires about $\log n$ many bits to be represented). However, it is an easy exercise to do the (annoying) convention between $m$ integers represented in binary that altogether have length $n$ bits.

# Chapter 2

# Interval Scheduling

*In this lecture, we begin by describing what a greedy algorithm is.* UNWEIGHTED INTERVAL SCHEDULING *is one problem that can be solved using such an algorithm. We then present* WEIGHTED INTERVAL SCHEDULING*, where greedy algorithms don't seem to work, and we must employ a new technique, dynamic programming, which in some sense generalizes the concept of a greedy algorithm.*

## 2.1 Facts about Greedy Algorithms

Most greedy algorithms make decisions by iterating the following 2 steps.

1. Order the input elements
2. Make an irrevocable decision based on a local optimization.

Here, **irrevocable** means that once a decision is made about the partially constructed output, it is never able to be changed. The greedy approach relies on the hope that repeatedly making locally optimal choices will lead to a global optimum.

Most greedy algorithms order the input elements just once, and then make decisions one by one afterwards. Kruskal's algorithm on minimum spanning trees is a well-known algorithm that uses the greedy approach. More sophisticated algorithms will reorder the elements in every round. For example, Dijkstra's shortest path algorithm will update and reorder the vertices to choose the one who's distance to the vertex is minimized.

Greedy algorithms are often the first choice when one tries to solve a problem for two reasons:

1. The greedy concept is simple.
2. In many common cases, a greedy approach leads to an optimal answer.

We present a simple algorithm to demonstrate the greedy approach.

## 2.2 Unweighted Interval Scheduling

Suppose we have a set of jobs, each with a given starting and finishing time. However, we are restricted to only one machine to deal with these jobs, that is, no two jobs can overlap with each other. The task is to come up with an arrangement or a "scheduling" that finishes the maximum number of jobs within a given time interval. The formal definition of the problem can be described as follows:

**Definition 2.2.1** (Intervals and Schedules)**.** Let $S$ be a finite set of intervals. An interval $I$ is a pair of two integers, the first smaller than the second; i.e. $I = (a, b) \in (\mathbb{Z}^+)^2$, $a < b$. We say that $S' \subseteq S$ is a *feasible schedule* if no two intervals $I, I' \in S$ *overlap* with each other. Let $I = (a_1, b_1), I' = (a_2, b_2)$; overlap means $a_2 < b_1 \leq b_2$ or $a_1 < b_2 \leq b_1$.

**Problem:** UNWEIGTHED INTERVAL SCHEDULING
**Input:** a finite set fo intervals $S$
**Output:** a feasible schedule $S' \subseteq S$, such that $|S'|$ is maximum (the maximum is taken over the size of all feasible schedules of $S$).

Figure 2.1 gives an example instance of the Unweighted Interval Scheduling Problem. Here interval $(1, 7)$ and interval $(6, 10)$ overlap with each other.

**(Failed) attempt 1:** Greedy rule: Order the intervals in ascending order by length, then choose the smaller intervals first. The intuition behind this solution is that shorter jobs may be less likely to overlap with other jobs. So we order the intervals in ascending order, then we choose jobs which do not overlap with the ones we have already chosen (choosing the shortest job which fulfils this requirement). However, this solution is not correct.

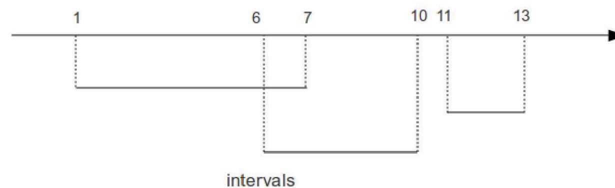



Figure 2.1: An example of the time line and intervals

Figure 2.2 gives a counterexample in which this greedy rule fails. Giving a counterexample is a standard technique to show that an algorithm is not correct[1]. In this example, the optimal choice is job 1 and job 3. But by adhering to a greedy rule that applies to the length of a job, we can only choose job 2. Solution 1 gives an answer only $\frac{1}{2}$ of the optimal. We know that this algorithm is wrong. But there is still one thing we can ask about this greedy algorithm: How bad can it be? Can

[1] The definition of a *correct* algorithm for a given problem, is that *for every input* (i) the algorithm terminates and (ii) if it terminates and the input is in the correct form then the output is in the correct form. Therefore, to prove that an algorithm is not correct for a given problem it is suffcient to show that *there exists an input* where the output is not correct (in this case, by definition of the problem, correct is a schedule of optimal size).

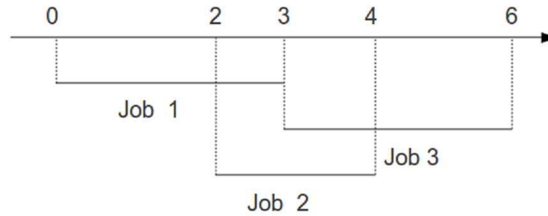



Figure 2.2: Counterexample of Solution 1

it give an result much worse than $\frac{1}{2}$? The answer is no. This greedy algorithm can always give a result no worse than $\frac{1}{2}$ of the optimal (see Exercise 1).

**Attempt 2:** Greedy rule: Sort by the finishing time in non-decreasing order, and choose jobs which do not overlap with jobs already chosen. A formal algorithm is described below.

**Algorithm 2:** A greedy algorithm for interval scheduling

**input** : $S$ : the set of all intervals
**output**: $S' \subseteq S$ s.t. $S'$ is a scheduling with maximum size
Order $S$ as $I_1 < I_2 < ... < I_n$ in non-decreasing finishing time;
$S' \leftarrow \emptyset$;
**for** $i \leftarrow 1$ *to* $n$ **do**
  **if** $I_i$ *does not overlap an interval in* $S'$ **then**
    $S' \leftarrow \cup \{I_i\}$;
  **end**
**end**
**return** $S'$;

We would like to show that this algorithm is "correct". But, we must first make sure that we define what is meant by the "correctness" of an algorithm.

The correctness of an algorithm is defined as follows:

1. The algorithm always terminates.

2. If it can be proved that the precondition is true, then the postcondition must also be true. In other words, every legal input will always generate the right output.

It is clear that Algorithm 8 will terminate after n iterations. For some algorithms though, termination can be a problem and might require more justification to show. We will now shift our focus to the more difficult question of how to prove Algorithm 8 always gives the correct maximum scheduling.

*Proof.* We begin by introducing some basic concepts used in the proof. Let $S'_i$ be the content of $S'$ at the end of the i'th iteration of the for loop, and let $OPT_i$ be an optimal extension of $S'_i (S'_i \subseteq OPT_i)$. There is one important thing about the second part of definition. We have to show the existence of such an optimal extension, *otherwise the problem is proved by the definition itself.*

**Remark 2.2.2.** You may wonder why we are concerned about a different $OPT_i$ (one for each iteration $i$). This is because after the base case is established, we find $OPT_n$ by assuming it has been proved for all values between the base case and $OPT_{n-1}$, inclusive. This is the way induction works.

We will show by induction that

$$P(i) : \forall i, \exists OPT_i \; s.t. \; S'_i \subseteq OPT_i$$

For the base case, $P(0)$, it is trivial to get $S'_0 = \emptyset$ and $S'_0 \subseteq OPT_0$. For the inductive step, suppose we have $P(k)$ and we want to solve $P(k+1)$. At time $k+1$, we consider interval $I_{k+1}$:

- **Case I:** $I_{k+1} \notin S'_{k+1}$. We can construct $OPT_{k+1} = OPT_k$. It is obvious that $S'_{k+1} = S'_k \subseteq OPT_{k+1}$.
- **Case II:** $I_{k+1} \in S'_{k+1}$
  - **Subcase a:** $I_{k+1}$ does not overlap $OPT_k$.
    This case cannot happen since it contradicts with the definition of $OPT_k$.
  - **Subcase b:** $I_{k+1}$ overlaps exactly one interval $I^*$ from $OPT_k$.
    Since $I_{k+1} \in S'_{k+1}$, $I_{k+1}$ will not overlap any interval in $S'_k$, which implies $I^* \in OPT_k \setminus S'_k$. So $S'_k \subseteq OPT_k \setminus \{I^*\}$. Then we can construct $OPT_{k+1} = (OPT_k \setminus \{I^*\}) \cup \{I_{k+1}\}$. $S'_{k+1} = S'_k \cup \{I_{k+1}\}$, where $S'_k \subseteq OPT_k \setminus \{I^*\}$. So $S'_{k+1} \subseteq OPT_{k+1}$.
  - **Subcase c:** $I_{k+1}$ overlaps more than one intervals from $OPT_i$.
    We wish to show this case can not happen. Since $I_{k+1} \in S'_{k+1}$, $I_{k+1}$ will not overlap any interval in $S'_k$, which implies $I_{k+1}$ only overlaps intervals from $OPT_k \setminus S'_k$. By way of contradiction, suppose $I_{k+1}$ overlaps with more than one interval $I_a, I_b ... I_t \in OPT_k \setminus S'_k$, for some $t$. Here $I_a, I_b ... I_t$ are sorted in non-decreasing order by finishing time. Because $I_a, I_b ... I_t \in OPT_k$, we have the property that earlier intervals always end before later ones start, i.e. all intervals in $OPT_k$ are sequential and non-overlapping. Since we have more than one interval that overlaps with $I_k$, there must be at least one interval from $I_a, I_b ... I_t \in OPT_k \setminus S'_k$ that ends before $I_{k+1}$ finishes (c.f. Figure 2.3). By the way the algorithm works, $I_{k+1}$ should end before all $I_a, I_b ... I_t$ end. So there is a contradiction.

By induction, for any $k > 0$, we can always construct $OPT_{k+1}$ from $OPT_k$ such that $S_{k+1} \subseteq OPT_{k+1}$. Thus

$$P(i) : \forall i, \exists OPT_i \; s.t. \; S'_i \subseteq OPT_i$$

□

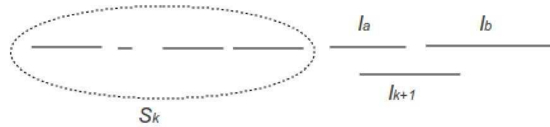



Figure 2.3: An example of Case II(b): There are two intervals $I_a$ and $I_b$ from $OPT_k$ overlaps with $I_k$

## 2.3 Weighted Interval Scheduling

This interval scheduling problem is identical to the previous problem, but with the difference that all intervals are assigned weights; i.e. a weighted interval $I$ can be identified with $I = (a, b)$ and an integer $w(I)$ which is its weight .
**Input:** S : a set of weighted intervals $S$ and $w : S \to \mathbb{N}$
**Output:** A schedule $S' \subseteq S$ of maximum total weight.

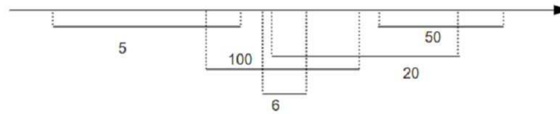



Figure 2.4: An example of Weighted Interval Scheduling

Unlike the Unweighted Interval Scheduling problem, no natural greedy algorithm is known for solving its weighted counterpart. It can be solved, however, via a different technique if we break the problem into sub-problems: dynamic programming. Now instead of just greedily building up one solution, we create a table and in each round we "greedily" compute an optimal solution to a *subproblem* by using the values of the optimal solutions of smaller subproblems we have constructed before.

Of course, not every problem can be solved via Dynamic Programming (DP). Roughly speaking, DP implicitly explores the entire solution space by decomposing the problem into *sub-problems*, and then using the solutions to those sub-problems to assist in solutions for larger and larger sub-problems.

After ordering intervals in non-decreasing order by finishing time, an example sub-problem in WEIGHTED INTERVAL SCHEDULING is to give the scheduling with maximum total weight up to a *specified* finishing time. For example, say that the input consists of the intervals $I_1, I_2, I_3, I_4$, and that the finishing time of $I_1$ is the smallest then it's $I_2$, then $I_3$ and then $I_4$. The DP algorithm computes the optimal schedule if the input consists only of $I_1$, then it computes the optimal schedule if the input consists of $\{I_1, I_2\}$, then it computes the optimal schedule if the input consists of $\{I_1, I_2, I_3\}$ and finally it computes the optimal schedule if the input consists of the entire list $\{I_1, I_2, I_3, I_4\}$. The reader should understand the following: (1) what would have happened if we had defined the subproblems without ordering the intervals in non-decreasing finishing time order and (2) how do we use the fact that in order to compute the optimal solution to input

$\{I_1, \ldots, I_k\}$ we should have recorded in ur table *all* previous solutions to subproblems defined on inputs $\{I_1, \ldots, I_{k-1}\}$, $\{I_1, \ldots, I_{k-2}\}$, $\{I_1, \ldots, I_{k-3}\}$, $\ldots$.

**Algorithm 3:** A dynamic programming algorithm for weighted interval scheduling

**input** : $S$ : the set of all weighted intervals
**output**: $S' \subseteq S$ s.t. $S'$ is a scheduling of maximum total weight
Order $S$ by $I_1 < I_2 < ... < I_n$ in non-decreasing finishing time;
$S' \leftarrow \emptyset$;
**for** $i \leftarrow 1$ *to* $n$ **do**
  Let $j$ be the largest integer s.t. S'[j] contains intervals not overlapping $I_i$;
  $S'[i] \leftarrow \max\{S'[i-1], S'[j] + w[I_i]\}$
**end**
**return** $S'$;

We will use the example illustrated in Figure 2.4 as the input to Algorithm 3. At the end of Algorithm 3's execution, the $S'$ array should look like the following:

| 0 | 5 | 11 | 100 | 100 | 150 |
|---|---|---|---|---|---|

We remark that unlike greedy algorithms, in Dynamic Programming algorithms the correctness is transparent in the description of the algorithm. Usually, the difficulty in a DP algorithm is coming up with a recurrence relation which relates optimal solutions of smaller subproblems to bigger ones (this recurrence relation is usually trivial to implement as an algorithm).

**Exercise 1.** Show why Attempt 1 given for the Unweighted Interval Scheduling problem cannot be worse than half of the optimal.

# Chapter 3

# Sequence Alignment: can we do Dynamic Programming in small space?

*In this course problems come in two forms: search and optimization. For example, the* SEARCH SHORTEST PATH *problem is for a given input to find an* actual *shortest path, whereas the optimization version of the problem is to find the* value *(i.e how long) is such a path. In this chapter, we define the* SEQUENCE ALIGNMENT *problem, and give a dynamic programming algorithm which solves it. It is very interesting that there is a conceptual difference in the algorithms that solve the SEARCH and the OPTIMIZATION version of problems. In particular, although it is easy to find an optimal Dynamic Programming algorithm for the optimization problem which uses little space, it is more involved to come up with a time-efficient algorithm that uses little space for the search version of the problem. To that end, we will combine Dynamic Programming and Divide and Conquer and we will give a solution which uses little time and simultaneously little space.*

## 3.1 Sequence Alignment Problem

Sequence alignment has a very common every day application: spell check. When we type "Algorthsm" when we mean to type "Algorithm", the computer has to find similar words to the one we typed and give us suggestions. In this situation, computers have to find the "nearest" word to the one we typed. This begs the question, how can we define the "nearest" word for a sequence? We will give a way to score the difference (or similarity) between strings. Apart from spell checkers this finds application to algorithms on DNA sequences. In fact, such DNA sequences have huge sizes, so an algorithm that works using space $n$ and algorithm that works using space $n^2$ make a big difference in practice!

Usually, alignments add gaps or insert/delete characters in order to make two sequences the same. Let $\delta$ be a gap cost and $\alpha_{a,b}$ be the cost of a mismatch between $a$ and $b$. We define the distance between a pair of sequences as the minimal alignment cost.

In the example above we have:

$$algor_thsm$$

$$algorith_m$$

In the above alignment, we add a total of two gaps to the sequences. We can say that the alignment cost in this case is $2\delta$. Notice that for any sequence pair, there is more than one way to make them same. Different alignments may lead to different costs.

For example, the sequence pair: "abbba" and "abaa", two sequences over the alphabet $\Sigma = \{a, b\}$. We could assign an alignment cost of $\delta + \alpha_{a,b}$:

$$abbba$$

$$ab_aa$$

Or we could assign an alignment cost of $3\delta$:

$$abbba_$$

$$ab__aa$$

Notice that the relationship between $\delta$ and $\alpha$ may lead to different minimal alignments.

Now we give a formal description of the Sequence Alignment Problem:

- **INPUT:** Two sequences $X, Y$ over alphabet $\Sigma$.
- **OUTPUT:** An alignment of minimum cost.

**Definition 3.1.1** (Alignment)**.** Let $X = x_1x_2...x_n$ and $Y = y_1y_2...y_m$. An alignment between $X$ and $Y$ is a set $M \subseteq \{1, 2, ..., n\} \times \{1, 2, ..., m\}$, such that $\nexists (i, j), (i', j') \in M$, where $i \leq i', j' \leq j$.

## 3.2 Dynamic Algorithm

**Claim 3.2.1.** *Let $X, Y$ be two sequences and $M$ be an alignment. Then one of the following is true:*

1. $(n, m) \in M$
2. *the $n^{th}$ position of $X$ is unmatched.*
3. *the $m^{th}$ position of $Y$ is unmatched.*

*Proof.* When we want to make an alignment for sequences $X$ and $Y$, for any $(n, m) \in \{1, 2, ..., n\} \times \{1, 2, ..., m\}$, one of three situations occur: $x_n$ and $y_m$ form a mismatch, $x_n$ should be deleted from $X$, or $y_m$ should be inserted into $X$. Notice that inserting a blank into the same position for both $X$ and $Y$ can never reach a best alignment, so omit this case. □

We denote by $Opt(i, j)$ the optimal solution to the subproblem of aligning sequences $X^i = x_1x_2...x_i$ and $Y^j = y_1y_2...y_j$. We denote by $\alpha_{x_i,y_j}$ the mismatch cost between $x_i$ and $y_j$ and by $\delta$ the gap cost. From the claim above, we have:

$$Opt(i, j) = \min \left\{ \begin{array}{c} \alpha_{x_i,y_j} + Opt(i-1, j-1), \\ \delta + Opt(i-1, j) \\ \delta + Opt(i, j-1) \end{array} \right\}$$

We can now present a formal dynamic programming algorithm for Sequence Alignment:

**Algorithm 4:** Sequence-Alignment-Algorithm

```
input  : Two sequences X, Y over alphabet Σ
output: Minimum alignment cost
for i ← 0 to n do
    for j ← 0 to m do
        if i = 0 or j = 0 then
            Opt(i, j) = (i + j) × δ;
        end
        else
            Opt(i, j) = min{α_{x_i,y_j} + Opt(i − 1, j − 1), δ + Opt(i − 1, j), δ + Opt(i, j − 1)};
        end
    end
end
return Opt(n, m)
```

It is clear that the running time and the space cost are both $O(nm)$ (the $n$x$m$ array $Opt(i,j)$ for space and the nested for loops give us the running time). However, we could improve the Algorithm 4 by reducing the space required while at the same time not increasing the running time! We can make an algorithm with space $O(n+m)$ using another technique, Divide-And-Conquer.

**Remark 3.2.2.** In fact, we can compute the value of the optimal alignment easily in space only $O(m)$. When computing $Opt(k+1,j)$, we only need the value of $Opt(k,j)$. Thus we can release the space $Opt(0,j)...Opt(k-1,j) \forall j = 0...m$. We will call this the **Space-efficient-Alignment-Algorithm**.

## 3.3 Reduction

We will introduce the Shortest Path on Grids with Diagonals problem and then reduce the Sequence Alignment problem to it. An example of the Shortest Path on Grids with Diagonals problem is given in Figure 3.1.

We are given two sequences $X = x_1x_2...x_n$ and $Y = y_1y_2...y_m$. Let the $i^{th}$ row of a grid (with diagonals) represent $x_i$ in $X$, and the $j^{th}$ column of a grid represent $y_j$ in $Y$. Let the weight of every vertical/horizontal edge be $\delta$ and the weight of a diagonal be $\alpha_{x_iy_j}$ . The shortest path on this graph from $u$ to $v$ is the same as the minimum alignment cost of two sequence.

Figure 3.1 gives an example of a grid with diagonals, $G_{XY}$. $x_1$ is matched to the first column, $x_2$ is matched to the second column. If we move horizontally from left to right on the first row starting from $u$, then this means $x_1$ is matched to a gap. Similarly, $y_2$ is matched to the bottom row, $y_1$ is matched to the upper row. If we move vertically from bottom to top, this means $y_1$ is matched to a gap. When we use diagonals, some letter of the two sequence are matched together, for example, if we move along the diagonal closest to $u$, then $x_1$ is matched to $y_1$. No matter the route we take (only allowing moves up, right and north-east diagonal), we can always find a path from $u$ to $v$ on $G_{XY}$. Moreover, this path always corresponds to an alignment of $X$ and $Y$ and vice versa.

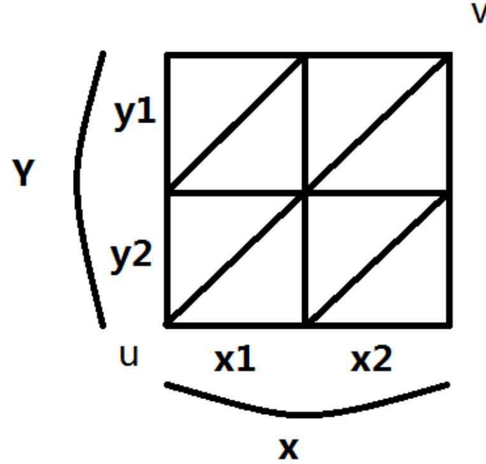



Figure 3.1: An example of grids with diagonals, $G_{XY}$

**Claim 3.3.1.** *The minimum value for the Shortest Path on Grids with Diagonals is the same as the minimum value alignment cost for Sequence Alignment.*

*Proof.* (*sketch*) For any two input sequences, a feasible solution for the Sequence Alignment problem can always have a one to one mapping to a feasible solution for the Shortest Path on Grids with Diagonals problem. □

In the Sequence-Alignment algorithm, if we know the value of the last row, we can run the algorithm in reverse. We call this algorithm the Reverse-Sequence-Alignment algorithm. Let $f[i][j]$ store the value of the shortest path from $u$ to location $(i, j)$, and $g[i][j]$ store the value of the Reverse-Sequence-Alignment algorithm at location $(i, j)$. In other words, $g[i][j]$ is the optimal alignment of subsequences $x_i x_{i+1} ... x_n$ and $y_j y_{j+1} ... y_m$.

**Claim 3.3.2.** *The shortest path in $G_{XY}$ that passes through $(i, j)$ has length $f[i][j] + g[i][j]$.*

*Proof.* Any path that passes through $(i, j)$ can be divided into two shorter paths, one from $u$ to $(i, j)$ and the other from $(i, j)$ to $v$. The shortest path from $u$ to $(i, j)$ is $f[i][j]$. Since the weights along the edges are fixed in $G_{XY}$ no matter which way they are traversed, the shortest path from $(i, j)$ to $v$ is equal to the value of the shortest path from $v$ to $(i, j)$, which is $g[i][j]$. So the minimum cost from $u$ to $v$ through $(i, j)$ is $f[i][j] + g[i][j]$. □

**Claim 3.3.3.** *For a fixed number $q$, let $k$ be the number that minimizes $f[q][k] + g[q][k]$. Then there exists a shortest path from $u$ to $v$ passing through $(q, k)$.*

*Proof.* Suppose there does not exist a shortest path from $u$ to $v$ passing through $(q, k)$. For a fixed number $q$, all the pathes from $u$ to $v$ must pass through some point $(q, t)$, $t \in 1...m$. Thus, a shortest path $L$ must pass through some point $(q, k')$. According to our assumption, $L$ cannot pass through $(q, k)$, which means that in particular, $f[q][k'] + f[q][k'] < f[q][k] + g[q][k]$. This is a contradiction. □

We give the final algorithm based on the Divide and Conquer technique below:

**Algorithm 5:** Divide and Conquer Alignment Algorithm$(X, Y)$

**input** : Two sequences $X = x_1 x_2 ... x_n, Y = y_1 y_2 ... y_n$
**output**: The node on the $u - v$ shortest path on $G_{XY}$
**if** $n \leq 2$ *or* $m \leq 2$ **then**
| exhaustively compute the optimal and quit;
**end**
Call Sequence-Alignment algorithm($X, Y[1, m/2]$)
Call Reverse-Sequence-Alignment algorithm($X, Y[m/2 + 1, m]$)
Find $q$ minimize $f[q][m/2] + g[q][m/2]$
Add $[q, m/2]$ to the output.
Call Divide-and-Conquer-Algorithm($X[1, q], Y[1, m/2]$)
Call Divide-and-Conquer-Algorithm($X[q, n], Y[m/2 + 1, m]$)

The space required in Algorithm 5 is $O(n+m)$. This is because we divide $G_{XY}$ along its center column and compute the value of $f[i][m/2]$ and $g[i][n/2]$ for each value of i. Since we apply the recursive calls sequentially and reuse the working space from one call to the next, we end up with space $O(n + m)$.

We claim the running time of the algorithm is $O(nm)$: Let $size = nm$, we find that $f(size) = O(size) + f(\frac{size}{2})$. This is because the first calls to Sequence-Alignment and Reverse-Sequence-Alignment each need $O(size)$ time, and the latter two calls need $f(\frac{size}{2})$ time. Solving the recurrence, we have the running time is $O(nm)$.

# Chapter 4

# Matchings and Flows

*In this chapter, we discuss matchings and flows from a combinatorial perspective. In a subsequent lecture we will discuss the same issues using Linear Programming. We introduce the problem of maximum flow, and algorithms for solving solve it, including the Ford-Fulkerson method and one polynomial time instantiation of it, the Edmonds-Karp algorithm. We prove the maximum flow - minimum cut theorem, and we show the polynomial time bound for Edmonds-Karp. Then, we apply network flows to solve the maximum bipartite matching problem. Finally, we apply non-bipartite matchings to give a 2-approximation algorithm for Vertex Cover.*

## 4.1 Introduction

Matching and flow are basic concepts in graph theory. And they have many applications in practice. Basically , in a graph a matching is a set of edges of which any two edges does not share a common end node. And a network flow in a directed weighed graph is like a distribution of workloads according to the capacity of each edge.

In this lecture,we are going to talk about maximum bipartite matchings. Formally, we have the following definitions :

**Definition 4.1.1** (Bipartite Matching)**.** For a graph $G = (X, Y, E)$,we say that $M \subseteq E$ is a (bipartite) matching if every vertex $u \in (X \cup Y)$ appears at most once in M.

Now we are aware of the formal definition of a matching, we can introduce the main problem we are gonna talk about today and the day following – Maximum Matching Problem.

**Definition 4.1.2** (Maximum Matching)**.** In a bipartite graph $G = (X, Y, E)$, $M \subseteq E$ is a matching. Then M is a maximum matching if for every matching $M'$ in $G$, $|M'| \leq |M|$. j.

**Definition 4.1.3** (Maximal Matching)**.** In a bipartite graph $G = (X, Y, E)$, $M \subseteq E$ is a matching. Then M is a maximal matching if there does not exist a matching $M' \supset M$.

**Remark 4.1.4.** Notice that we use the term "maximum" and not "maximal". These two are different. A "maximal" matching is a matching that we can not add more edges into and keep it still a matching; while a "maximum" matching is the maximal matching with largest size. Here we give an example in Figure 4.1.

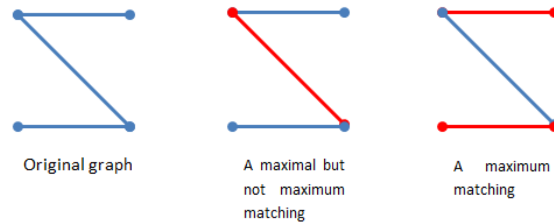



Figure 4.1: *This figure shows the difference between a maximum matching and a maximal matching.*

Here we define the Maximum Bipartite Matching problem.

**Definition 4.1.5.** Maximum Bipartite Matching

- *Input:* A bipartite graph $G = (X, Y, E)$.
- *Output:* A maximum matching $M \subseteq E$.

The maximum bipartite matching problem is fundamental in graph theory and theoretical computer science. It can be solved in polynomial time. BUT notice that there exists NO dynamic programming algorithm that can solve this problem. In order to solve this problem, we introduce another problem called the Maximum Flow Problem. Later we'll see how can we solve maximum matching using maximum flow.

## 4.2 Maximum Flow Problem

What is a FLOW? The word "flow" reminds us of water flow in a first impression. Yes, network flow is just like pushing water from the source to the sink, with the limitation of the capacity of each pipe(here it's edge in a graph). And a maximum flow problem is just pushing as much water in as we can. Formally the definition of a network flow is as follows:

**Definition 4.2.1** (Network Flow)**.** Given $(G, s, t, c)$, where G is a graph $(V, E)$,$s, t \in V$,c is a function which maps an edge to a value, a flow $f : E \to \mathbb{R}$ is a function where :

1. $f(e) \leq c(e), \forall e \in E$
2. $\sum_{\forall e=(u,v)} f(e) = \sum_{\forall e'=(v,w)} f(e^{'}), \forall v \in V$

We define the value of a flow $v(f)$ as $v(f) = \sum_{\forall e=(s,u)} f(e)$, which means the sum of all the edges leading out of the source s. And the maximum flow problem is defined as follows:

**Definition 4.2.2.** Maximum Flow Problem

- *Input:* (G,s,t,c).
- *Output:* a flow in the graph with maximum value.

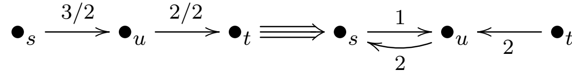



Figure 4.2: the transform from a network to its residual network.

In order to solve the maximum flow problem, we introduce another concept in the graph theory – residual graph:

**Definition 4.2.3** (Residual Graph)**.** For a net work $N : (G, s, t, c)$, and $f$ is a flow for $N$. Then $G_f$ the residual network is a graph where:

1. $V_f = V$
2. $\forall e \in E$, if $c(e) > f(e)$, then e is in $E_f$ and $c_f(e) = c(e) - f(e)$.(Here e is called a forward edge)
3. $\forall e = (u, v) \in E$, if $f(e) > 0$,then $e^{'} = (v, u)$ is in $E_f$ and $c(e^{'}) = f(e)$ (Here $e^{'}$ is called a backward edge)

Here is an example of the transform from a network to its residual network in Figure 4.2.

# 4.3 Max-Flow Min-Cut Theorem

This part we are going to talk about the relation between CUT and FLOW. Firstly we introduce the definition of CUT.

**Definition 4.3.1** (CUT)**.** $G = (V, E)$,$V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$ then we call $(V_1, V_2)$ a cut of the graph G.

In a graph $G = (V, E)$, for every cut $S, T$ such that $s \in S$ and $t \in T$, we call it a s-t cut.

**Claim 4.3.2.** *Let $f$ be any flow in G, for every s-t cut $(S, T)$,$v(f) = f^{out}(S) - f^{in}(S)$.*

*Proof.* By definition, $f^{in}(s) = 0$, so we have $v(f) = \sum_{\forall e=(s,u)} f(e) = f^{out}(s) - f^{in}(s)$. Since every node $v$ in $S$ other than $s$ is internal, we have $f^{out}(v) - f^{in}(v) = 0$. Thus, $v(f) = f^{out}(s) - f^{in}(s) = \sum_{v \in S}(f^{out}(v) - f^{in}(v)) = f^{out}(S) - f^{in}(S)$. □

Then we have a Max-Flow Min-Cut theorem:

**Theorem 4.3.3** (Max-Flow Min-Cut)**.** *If $f$ is an $s - t$ flow such that there is no $s - t$ path in the residual graph $G_f$, then there is an s-t cut $(A^*, B^*)$ in $G$ for which $v(f) = c(A^*, B^*)$. Consequently, $f$ has the maximum value of any flow in $G$, and $(A^*, B^*)$ has the minimum capacity of any $s - t$ cut in $G$.*

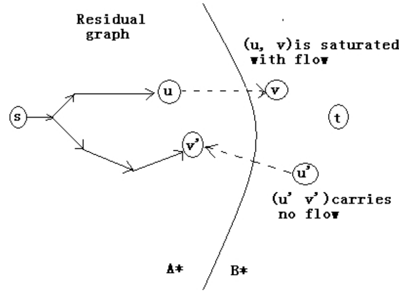



Figure 4.3: the construction of $A^*$ in a graph

*Proof.* Let $A^*$ denote the set of all nodes $v$ in $G$ for which there is an $s-v$ path in $G_f$. Let $B^*$ denote the set of all other nodes: $B^* = V - A^*$. First, we establish that $(A^*, B^*)$ is indeed an $s-t$ cut. The source s belongs to $A^*$ since there is always a path from s to s. Moreover, t doesn't belong to $A^*$ by the assumption that there is no $s-t$ path in the residual graph, so $t \in B^*$.

As shown in Figure 4.3, suppose that $e = (u, v)$ is an edge in G for which $u \in A^*$ and $v \in B^*$. We claim that $f(e) = c(e)$. If not, e would be a forward edge in the residual graph $G_f$, so we would obtain an $s-v$ path in $G_f$, contradicting our assumption that $v \in B^*$. Now we suppose that $e^{'} = (u^{'}, v^{'})$ is an edge in G for which $u^{'} \in B^*$ and $v^{'} \in A^*$. We claim that $f(e^{'}) = 0$. If not, e would be a backward edge $(v^{'}, u^{'})$ in the residual graph $G_f$, so we would obtain an $s-u^{'}$ path in $G_f$, contradicting our assumption that $u^{'} \in B^*$. So we have an conclusion:

$$\begin{aligned} v(f) &= f^{out}(A^*) - f^{in}(A^*) \\ &= \sum_{e=(u,v),u\in A^*,v\in B^*} f(e) - \sum_{e=(u,v),u\in B^*,v\in A^*} f(e) \\ &= \sum_{e=(u,v),u\in A^*,v\in B^*} c(e) - 0 \\ &= c(A^*, B^*) \end{aligned}$$

□

## 4.4 Ford-Fulkerson Algorithm

From the Max-flow Min-cut Theorem, we know that if the residual graph contains a path from $s$ to $t$, then we can increase the flow by the minimum capacity of the edges on this path, so we must not have the maximum flow. Otherwise, we can define a cut $(S, T)$ whose capacity is the same as the flow $f$ , such that every edge from $S$ to $T$ is saturated and every edge from $T$ to $S$ is empty,

which implies that $f$ is a maximum flow and $(S, T)$ is a minimum cut. Based on the theorem, we can have the Ford-Fulkerson Algorithm: Starting with the zero flow, repeatedly augment the flow along any path s-t in the residual graph, until there is no such path.

**Algorithm 6:** Ford-Fulkerson Algorithm

**input** : A network (G,s,t,c)
**output**: a flow in the graph with maximum value
$\forall e, f(e) \leftarrow 0$
**while** $\exists s - t$ *path* **do**
  $f' \leftarrow augment(f, p)$
  $f \leftarrow f'$
  $G_f \leftarrow G_{f'}$
**end**
**return** $f$

Notice that the $augment(f, p)$ in the algorithm.This is an updating procedure of the graph.It means to find randomly a $s - t$ path and push through a flow of minimum capacity of this path.

Also there are several things we need to know about the Ford-Fulkerson Algorithm:

1. Ford-Fulkerson Algorithm does not always terminate when we have real numbers assignment. Consider the flow network shown in Figure 4.4, with source s, sink t,capacities of edges $e_1$,$e_2$ and $e_3$ respectively 1, $r = \frac{\sqrt{5}-1}{2}$ and the capacity of all other edges some integer $M \geq 2$. Here the constant r was chosen so $r^2 = 1-r$. We use augmenting paths according to the table, where $p_1 = \{s, v_4, v_3, v_2, v_1, t\}$,$p_2 = \{s, v_2, v_3, v_4, t\}$,$p_3 = \{s, v_1, v_2, v_3, t\}$. Note that after step 1 as well as after step 5, the residual capacities of edges $e_1$, $e_2$ and $e_3$ are in the form $r^n$, $r^{n+1}$ and 0, respectively, for some $n \in \mathbf{N}$. This means that we can use augmenting paths $p_1$, $p_2$, $p_1$ and $p_3$ infinitely many times and residual capacities of these edges will always be in the same form. Total flow in the network after step 5 is $1+2(r^1+r^2)$. If we continue to use augmenting paths as above, the total flow converges to $1+2\sum_{i=1} \inf r^i = 3+2r$ , while the maximum flow is $2M + 1$. In this case, the algorithm never terminates and the flow doesn't even converge to the maximum flow.

2. if function c maps the edges to integer, then FF outputs integer value.

This algorithm seems quite natural to us, but in fact the worse case running time of Ford-Fulkerson Algorithm is exponential. Here is a famous example of Ford-Fulkerson Algorithm running in exponential time in Figure 4.5.

In this example, if the algorithm choose an $s - t$ path which contains the edge $u - v$ or $v - u$ (this is possible,recalling that the existence of the backword edge), each time we can only push 1 unit flow. And obviously the max flow for the network is $2^{n+1}$, so we need to choose $2^{n+1}$ times augmenting. Thus in this case the running time is exponential of input length (which is n).

### 4.4.1 Edmonds-Karp Algorithm

Now we introduce a faster algorithm called the Edmonds-Karp algorithm which was published by Jack Edmonds and Karp in 1972, and solves the maximum flow problem in $O(m^2 n)$ running

| Step | Augmenting path | Sent flow | Residual capacities | | |
|---|---|---|---|---|---|
| | | | $e_1$ | $e_2$ | $e_3$ |
| 0 | | | $r^0 = 1$ | $r$ | 1 |
| 1 | $\{s, v_2, v_3, t\}$ | 1 | $r^0$ | $r^1$ | 0 |
| 2 | $p_1$ | $r^1$ | $r^2$ | 0 | $r^1$ |
| 3 | $p_2$ | $r^1$ | $r^2$ | $r^1$ | 0 |
| 4 | $p_1$ | $r^2$ | 0 | $r^3$ | $r^2$ |
| 5 | $p_3$ | $r^2$ | $r^2$ | $r^3$ | 0 |

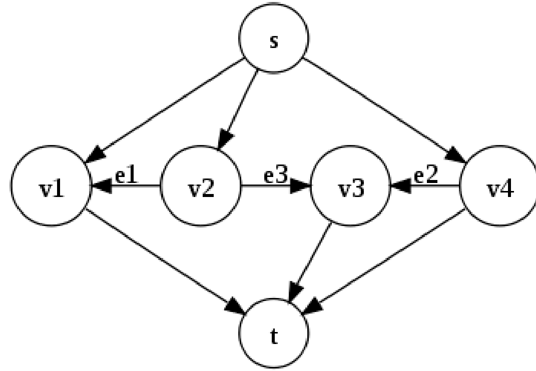



Figure 4.4: an example of Ford-Fulkerson algorithm never terminates

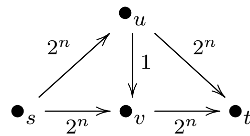



Figure 4.5: An example that Ford-Fulkerson terminates in exponential time

time.This algorithm only differs from the Ford-Fulkerson algorithm in one rule: Ford-Fulkerson algorithm chooses the augmenting path arbitrarily, but *Edmonds-Karp algorithm chooses the augmenting path of the shortest traversal length.*

This rule sounds amazing: what does the traversal length do with the flow that we can push through this path? But deeper analysis shows that they indeed have relations to each other.

**Theorem 4.4.1.** *E-K terminates in* $O(|E|^2|V|)$

*Proof.*

**Lemma 4.4.2.** *If we use Edmonds-Karp algorithm on the flow network* $(G = (V, E), s, t, c)$*, for any* $v \in V - \{s, t\}$*, the shortest-path distance* $\delta_f(u, v)$ *will monotone increasing with each flow augmentation.*

*Proof.* We will suppose that for some vertex $v \in V - \{s, t\}$, there is a flow augmentation that causes the shortest-path distance from s to v to decrease, and then we will derive a contradiction. Let f be the flow just before the first augmentation that decreases some shortest-path distance, and let $f'$ be the flow just afterward. Let v be the vertex with the minimum $\delta_{f'}(s, v)$ whose distance was decreased by the augmentation, so that $\delta_{f'}(s, v) < \delta_f(s, v)$. Let $p = s \to \ldots u \to v$ be the shortest path from s to v in $G'_f$, so that $(u, v) \in E_{f'}$, and $\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$, $\delta_{f'}(s, u) \geqslant \delta_f(s, u)$. Then we will get (u, v) doesn't belong to $E_f$(the proof is easy). As $(u, v) \in E_f$, the augmentation must have increased the flow from v to u, and therefore the shortest path from s to u in $G_f$ have(v, u) as its last edge. Then

$$\delta_f(s, v) = \delta_f(s, u) - 1 \leqslant \delta_{f'}(s, u) - 1 = \delta_{f'}(s, v) - 2$$

. It contradicts our assumption that $\delta_{f'}(s, v) < \delta_f(s, v)$ □

**Theorem 4.4.3.** *If the Edmonds-Karp algorithm runs on a flow network* $G = (V, E)$ *with source s and sink t, then the total number of flow augmentations performed by the algorithm is* $O(VE)$*.*

*Proof.* We say that an edge $(u, v)$ in a residual network $G_f$ is critical on an augmenting path p if the residual capacity of p is the residual capacity of $(u, v)$, that is , if $c_f(p) = c_f(u, v)$. After we have augmented flow along an augmenting path, any critical edge on the path disappears from the residual network. Moreover, at least one edge on any augmenting path must be critical. We will show that each of the $|E|$ edges can become critical at most $\frac{|V|}{2} - 1$ times.
Let u and v be vertices in V that are connected by an edge in E. Since augmenting paths are shortest paths, when (u, v) is critical for the first time, we have $\delta_f(s, v) = \delta_f(s, u) + 1$. Once the flow is augmented, the edge$(u, v)$ disappears from the residual network. It cannot reappear later on another augmenting path until after the flow u to v is decreased, which occurs only if (u, v) appears on an augmenting path. If $f'$ is the flow in G when this event occurs, then we have $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$. As

$$\delta_f(s, v) \leqslant \delta_{f'}(s, v)$$

we have

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geqslant \delta_f(s, v) + 1 = \delta_f(s, u) + 2$$

Consequently, from the time (u, v) becomes critical to the time when it next becomes critical, the distance of u from the source increases by at least 2. The distance of u from the source is initially at

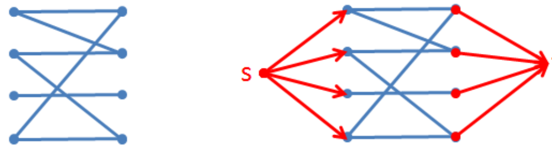



Figure 4.6: constructing a new network in order to run Ford-Fulkerson on it

least 0. The intermediate vertices on a shortest path from s to u can't contain s, u, or t. Therefore, until u becomes unreachable from the source, if ever, its distance is at most $\mid V \mid$-2. Thus, (u, v) can become critical at most $\frac{|V|-2}{2} = \frac{|V|}{2} - 1$ times. Since there are O(E) pairs of vertices that can have an edge between them in a residual graph, the total number of critical edges during the entire execution of the Edmonds-Karp algorithm is O(VE). Each augmenting path has at least one critical edge, and hence the theorem follows. □

Since each iteration of Ford-Fulkerson can be implemented in O(E) time when the augmenting path is found by breadth-first search, the total running time of the total running time of the Edmonds-Karp algorithm is $O(\mid E \mid^2 \mid V \mid)$.

□

## 4.5 Bipartite Matching via Network Flows

In this section we'll solve the maximum matching problem using the network flow problem. In fact this is rather straight forward. Let $G = (X, Y, E)$ be a bipartite graph. We add two nodes ,$s$,$t$,to the graph and connect from s to all nodes in X,and connect all nodes in Y to t,as in Fig 6. Run Ford-Fulkerson on the new network N, we get a maximum flow f.Define a matching $M = \{e \mid f(e) = 1\}$,then m is a maximum matching. The proof is simple: we just need to prove that we can build a one-on-one correspondence from matchings to flows.And this is obvious: for a matching, add two nodes s and t and do similar things as above, we get a flow.And for a flow f, define a matching $M = \{e \mid f(e) = 1\}$,we get a matching. Thus the matching we find using Ford-Fulkerson is a maximum matching.

## 4.6 Application of Maximum Matchings: Approximating Vertex Cover

We introduce a new problem: vertex cover:

**Definition 4.6.1.** Vertex Cover

- *Input:* $G = (V, E)$
- *Output:* a cover of minimum size.

Here is the definition for COVER:

**Definition 4.6.2.** $\mathbf{Cover}$ : $G = (V, E)$,$U \in V$ is a cover if $\forall e \in E$,$\exists u \in U$ which is a end node of e.

Despite the long time of research, the problem remains open.

1. $\mathbf{Negative}$: VC can not be approximated with a factor smaller than 1.363, unless $P = NP$.
   ,

2. $\mathbf{Positive}$ : We have a polynomial algorithm for VC with approximation ratio $2-\Omega(\frac{1}{\sqrt{log(n)}})$.

3. $\mathbf{Belief}$ : VC cannot be approximated within $2 - \epsilon$, $\forall \epsilon > 0$. **Notice:** in fact under something which is called the Unique Game Conjecture this is true.

Here is a simple algorithm that approximates VC with ratio 2:

1. compute the maximum matching M in the graph.

2. output the vertices of M.

*Proof.* Observe that $\mid M \mid \leq OPT$,where M is the max matching. Also notice that vertices in M cover G. Suppose that there is a node u which is not covered by M, then it must be connected to a node outside M,say v.Then $(u, v)$ is an edge that does not belong to M,which contradicts that M is a max matching.And the algorithm outputs $2 \mid M \mid$ nodes, and $2 \mid M \mid \leq 2OPT$ □

# Chapter 5

# Optimization Problems, Online and Approximation Algorithms

*In this chapter, we begin by introducing the definition of an optimization problem followed by the definition of an approximation algorithm. We then present the Makespan problem as a simple example of when an approximation algorithm is useful. Because the Makespan problem is NP-hard, there does not exist a polynomial algorithm to solve it unless $P = NP$. If we can not find an exact answer to an arbitrary instance of Makespan in polynomial time, then perhaps we can give an approximate solution. To this end, we formulate an approximation algorithm for the Makespan problem: Greedy-Makespan. We go on to improve this algorithm to one with a lower approximation ratio (i.e. an algorithm that is guaranteed to give an even closer approximation to an optimal answer).*

## 5.1 Optimization Problems

**Definition 5.1.1.** An optimization problem is finding an optimal (best) solution from a set of feasible solutions.

Formally, an optimization problem $\Pi$ consists of:

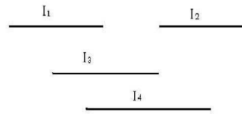
1. A set of inputs, or instances, $\mathcal{I}$.

2. A set $f(I)$ of feasible solutions, for each given instance $I \in \mathcal{I}$.

3. An optimization objective function: $c : \mathcal{F} \to \mathbb{R}$ where $\mathcal{F} = \bigcup_{I \in \mathcal{I}} f(I)$. That is, given an instance $I$ and a feasible solution $y$, the objective function denotes a "measure" of $y$ with a real number.

The goal of an optimization problem $\Pi$ is to either give a minimum or maximum: the shortest or longest path on a graph, for example. For maximization problems, the goal of problem $\Pi$ is to find an output $\forall I$, $O \in f(I)$, s.t. $c(O) = \max_{O' \in f(I)} c(O')$.

For example, in the Unweighted Interval Scheduling Problem, we have 4 intervals, as shown in the following Figure 5.1




Here feasible solutions are $\{I_1, I_2\}, \{I_1\}, \{I_2\}, \{I_3\}, \{I_4\}$, and $\{I_1, I_2\}$ is optimal, as it maximizes the number of intervals that can be scheduled.

**Notice**: In the following section, we will only discuss maximization problems for brevity; minimization problems are in no way fundamentally different.

## 5.2 Approximation Algorithms

Sometimes we may not know the optimal solution for the problem, if there is no known efficient algorithm to produce it, for example. In these cases, it might be useful to find an answer that is "close" to optimal. Approximation algorithms do exactly this, and we present them formally presently.

**Definition 5.2.1** (Approximation Algorithm)**.** Let $\Pi$ be an optimization problem and $A$ be a polynomial time algorithm for $\Pi$. For all instances $I$, denote by $OPT(I)$ the value of the optimal solution, and by $A(I)$ the solution computed by $A$, for that instance. We say that $A$ is a c-approximation algorithm for $\Pi$ if

$$\forall I, OPT(I) \leq cA(I)$$

## 5.3 Makespan Problem

In this section, we will discuss the Makespan problem followed by an approximation algorithm which solves it. Makespan is a basic problem in scheduling theory: how to schedule a set of jobs on a set of machines.

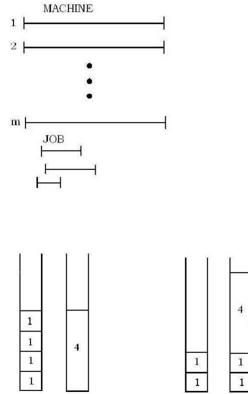
**Definition 5.3.1** (Makespan Problem)**.** We are given $n$ jobs with different processing times, $t_1, t_2, ..., t_n$. The goal is to design a assignment of the jobs to $m$ separate, but identical machines such that the last finishing time for given jobs (also called makespan) is minimized.

We firstly introduce some basic notations to analysis the Makespan Problem. $m \in \mathbb{N}$ and $t_1, t_2, ..., t_n$ are given as input. Here, $m$ is the number of machines and $t_1, t_2, ..., t_n$ are the processing length of jobs. Figure 5.3 gives an example input. We define $A_i$ to be the set jobs assigned to machine $i$ and $T_i$ to be the total processing time for machine $i$: $T_i = \sum_{t \in A_i} t$. The goal of the problem is to find a scheduling of the jobs that minimize $max_i T_i$.

Suppose we are given as input: m = 2 ,$< t_1, t_2, t_3, t_4, t_5 > = < 1, 1, 1, 1, 4 >$.

Of these two possible solutions, the left one is better, in fact it is optimal.

**Fact 5.3.2.** *MAKESPAN is NP-hard.*




Thus, a polynomial time algorithm to solve an arbitrary instance of Makespan does not exist unless $P = NP$. However, we can find an approximation algorithm that gives a good, albeit possibly suboptimal, solution that *can* run in polynomial time.

## 5.4 Approximation Algorithm for Makespan Problem

A greedy approach is often one of the first ideas to try, as it is so intuitive. The basic idea is as follows: we schedule the jobs from 1 to $n$, assigning the $i^{th}$ job to the machine with the shortest total processing time.

**Algorithm 7:** Greedy Makespan

**input** : $m \in \mathbb{N}$: number of machines and $t_1, t_2, ..., t_n$: the processing length of jobs
**output**: $T$: the last finishing time for the scheduling
**for** $i \leftarrow 1$ *to* $n$ **do**
  choose the smallest $T[k]$ over $T[i](i = 1, 2, ..., m)$
  $A[k] = A[k] + T[i]$
  $T[k] = T[k] + t[i]$
**end**
$T = \min_{i \in [n]}\{T[i]\}$
**return** $T$

**Theorem 5.4.1.** *Greedy Makespan is a 2-approximation algorithm for Makespan*

*Proof.* Our proof will first show that Greedy Makespan gives an approximation ratio with a lower bound of 2 (the solution is at least 2 times the optimal). Then we will show that Greedy Makespan has an upper bound of 2 (the solution is no worse than 2 times the optimal). In this way we will be able to show that Greedy Makespan is indeed a 2-approximation for Makespan.
Here is an example to prove that the lower bound approximation ratio of Greedy Makespan Algorithm is at least 2: Consider the case in which there are m machines and n jobs, where $n = m(m-1)+1$. Among the n jobs, $m(m-1)$ of them have processing length 1 and the last one has processing time $m$. Greedy-Makespan will schedule jobs as shown in Figures 5.1 and 5.2. The optimal solution for is shown in Figure 5.3.

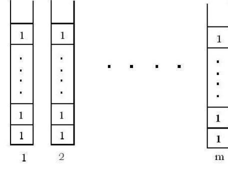


Figure 5.1: Before placing the last job.

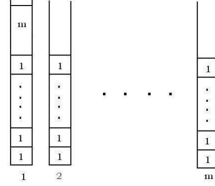


Figure 5.2: After placing the last job

Now we have an approximation ration of Greedy Makespan $= \frac{2m-1}{m} = 2 - \frac{1}{m}$. Thus, by example, we have shown that Greedy Makespan has an approximation ratio lower bound of 2.

We now show that the output of the greedy algorithm is at *most* twice that of the optimal solution, i.e. has an upper bound of 2. Firstly we bring in two conspicuously tenable facts:

1. $\frac{1}{m}\sum t_i \leq OPT(I)$.
2. $t_{max} \leq OPT(I)$ where $t_{max}$ is the most wasteful job.

Let $T'_k = T_k - t_j$ where $t_j$ is the last job assigned to machine $k$. It is easy to see that $T'_k$ is at most the average process time. From property (1), we have: $T'_k \leq \frac{1}{m}\sum_i t_i \leq OPT$. From property (2) we have: $T'_k = T_k - t_j \geq T_k - OPT$. Combining these facts, we get: $T_k - OPT \leq OPT$. So $T_k \leq 2OPT$. □

Now we have an 2-approximation algorithm for Makespan Problem. But can we do better? The answer is yes. Next we will improve this bound with Improved Greedy Makespan.

The only difference between the Improved Greedy Makespan and Greedy Makespan is that the improved algorithm first sorts $t_i$ by non-increasing length before scheduling.

**Theorem 5.4.2.** *Improved Greedy Makespan is a $\frac{3}{2}$-approximation algorithm for Makespan.*

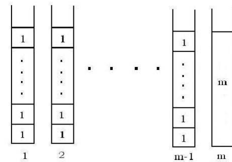


Figure 5.3: Optimal solution

*Proof.* Since processing time are sorted in non-increasing order at the beginning of the algorithm, we have $t_1 \geq t_2 \geq ... \geq t_n$.

In this proof we will make use of the following claim:

**Claim 5.4.3.** *When $n \leq m$, Improved Greedy Makespan will obtain an optimal solution. When $n > m$, we have $OPT \geq 2t_{m+1}$.*

*Proof.* When $n \leq m$, every job is assigned to unique machine, thus the output of the algorithm is equal to the finishing time of the largest job. This is obviously optimal.

When $n > m$, then by the pigeonhole principle there must be two jobs $t_i, t_j$ that, in the optimal case, are assigned to the same machine where $i, j \leq m + 1$ and $t_i, t_j \geq t_{m+1}$. We conclude that $OPT \geq 2t_{m+1}$. ☐

When we define: $T'_k = T_k - t_j$, $t_j$ the last job assigned to machine $k$, then we have:

1. If $t_j$ is such that $j \leq m$, then there is only one job scheduled on machine $k$. This means that $T_k \leq t_1 \leq OPT$.

2. If $t_j$ is such that $j > m$, then machine $k$ should be the one with the shortest total length before scheduling the $j^{th}$ job. This implies $T'_k \leq \frac{1}{j-1} \sum_i t_i \leq \frac{1}{m} \sum_i t_i \leq OPT$.

   Furthermore, we have $T'_k = T_k - t_j \geq T_k - \frac{1}{2}OPT$. Thus, $T_k - \frac{1}{2}OPT \leq OPT$, which implies $T_k \leq \frac{3}{2}OPT$

So for every $k \in [m]$, $T_k \leq \frac{3}{2}OPT$. Therefore $\frac{3}{2}$ is an upper bound on the approximation ratio of Improved Greedy Makespan. ☐

Although we have shown that Improved Greedy Makespan has an approximation ratio of $\frac{3}{2}$, this bound is not tight. That is, we can do better.

**Remark 5.4.4.** Improved Greedy Makespan is in fact a $\frac{4}{3}$-approximation algorithm. (Proof omitted)

# Chapter 6

# Introduction to Convex Polytopes

*In this lecture, we cover some basic material on the structure of polytopes and linear programming.*

## 6.1 Linear Space

**Definition 6.1.1** (Linear space)**.** $V$ is a **linear space**, if $V \subseteq \mathbb{R}^d, d \in \mathbb{Z}^+$, $V$ = *span*$\{\vec{u_1}, \vec{u_2}, \ldots, \vec{u_k}\}$,$u_i \in \mathbb{R}^d$. Equally, we can define linear space as $V = \{\vec{u} | \forall c_1, c_2, \ldots, c_k \in \mathbb{R}, \vec{u} = c_1\vec{u_1} + \cdots + c_k\vec{u_k}\}$.

**Definition 6.1.2** (Dimension)**.** Let *dim*$(V)$ be the **dimension** of a linear space $V$. The dimension of a linear space $V$ is the cardinality of a basis of $V$. A basis is a set of linearly independent vectors that, in a linear combination, can represent every vector in a given linear space.

**Example 6.1.3** (Linear space)**.** Line $V \subseteq \mathbb{R}^2$ in Figure 1 is a linear space since it contains the origin vector **0**. It's obvious that *dim*(V) = 1.

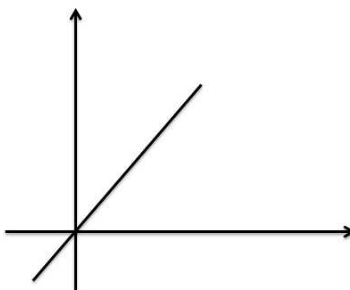


Figure 6.1: An example of linear space

## 6.2 Affine Space

In this section, we propose the notion of *affine maps*: $x \mapsto Ax + x_0$, which represent an affine change of coordinates if $A$ is a nonsingular $n \times n$ matrix, $x, x_0 \in \mathbb{R}^n$.

Then we will introduce an *affine space*. An affine space is a linear space that has "forgotten" its origin. It can be seen as a shift of a linear space.

**Definition 6.2.1** (Affine Space)**.** Let $V$ be a linear space. $V'$ is an **affine space** if $V' = V + \vec{\beta}$, where vector $\vec{v'} \in V'$ if and only if $\forall \vec{v'} \in V'$, $\vec{v'} = \vec{v} + \vec{\beta}$, where $\vec{v}$ is a vector in a linear space $V$, and $\vec{\beta}$ is a fixed scalar for an affine space.

**Example 6.2.2.** In Figure 6.2, line $V$ is a linear space while line $V'$ is not since it does not contain the origin vector **0**. $V'$ is an affine space with the property that $V' = V + \vec{\beta} = \{\vec{v} + \vec{\beta} | \vec{v} \in V\}$ where $V \subseteq \mathbb{R}^2$ is a linear space and $\vec{\beta} \in \mathbb{R}^2$ is a scalar. Note that $dim(V') = dim(V) = 1$.

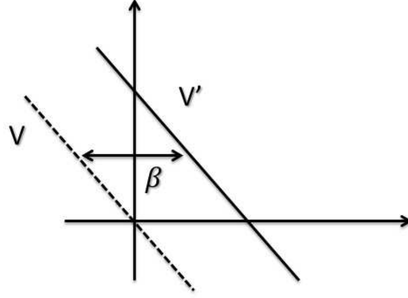



Figure 6.2: An example of affine space

**Definition 6.2.3** (Affine Hull)**.** The affine hull $aff(S)$ of $S$ is the set of all affine combinations of elements of $S$, that is,

$$aff(S) = \{\vec{u'} | \vec{u'} = \sum_{i=0}^{k} \lambda_i \vec{u_i}, \vec{u_i} \in S, \lambda_i \in \mathbb{R}, i = 1...k; \sum_{i=0}^{k} \lambda_i = 1 \; k = 1, 2, ...\}$$

**Claim 6.2.4.** *$V'$ is an affine space if and only if there exist a linear $S = \{\vec{u_0}, \vec{u_1}, ..., \vec{u_k}\}$ such that $V' = aff(S)$*

*Proof.* Let $V' = V + \vec{\beta}$. We define $V = span\{\vec{u_1}, \vec{u_2}, ..., \vec{u_k}\}$, and we let $B = \{\vec{u_1}, \vec{u_2}, ..., \vec{u_k}\}$, and finally $B' = (B \cap \vec{0}) + \vec{\beta} = \{\vec{\beta}, \vec{\beta} + \vec{u_1}, \vec{\beta} + \vec{u_2}, ..., \vec{\beta} + \vec{u_k}\}$.

- For every $\vec{y} \in aff(B')$, we show that $\vec{y} \in V'$:

$$\begin{aligned} \vec{y} &= \lambda_0 \vec{\beta} + \lambda_1(\vec{u_1} + \vec{\beta}) + \lambda_2(\vec{u_2} + \vec{\beta}) + ... + \lambda_k(\vec{v_k} + \vec{\beta}) \\ &= (\lambda_0 + \lambda_1 + ... + \lambda_k)\vec{\beta} + \lambda_1\vec{u_1} + \lambda_2\vec{u_2} + ... + \lambda_k\vec{u_k} \\ &= \vec{\beta} + \lambda_1\vec{u_1} + \lambda_2\vec{u_2} + ... + \lambda_k\vec{u_k} \end{aligned}$$

  Since $\lambda_1\vec{u_1} + \lambda_2\vec{u_2} + ... + \lambda_k\vec{u_k} \in V$, $\vec{y} \in V'$.

- For every $\vec{y} \in V'$, we show that $\vec{y} \in aff(B')$

$$\begin{aligned} \vec{y} &= \vec{\beta} + \lambda_1\vec{u_1} + \lambda_2\vec{u_2} + ... + \lambda_k\vec{u_k} \\ &= \vec{\beta} + (\lambda_1 + \lambda_2 + ... + \lambda_k)\vec{\beta} - (\lambda_1 + \lambda_2 + ... + \lambda_k)\vec{\beta} + \lambda_1\vec{u_1} + \lambda_2\vec{u_2} + ... + \lambda_k\vec{u_k} \\ &= [1 - (\lambda_1 + \lambda_2 + ... + \lambda_k)]\vec{\beta} + \lambda_1(\vec{u_1} + \vec{\beta}) + \lambda_2(\vec{u_2} + \vec{\beta}) + .. + \lambda_k(\vec{u_k} + \vec{\beta}) \\ &= \lambda_0\vec{\beta} + \lambda_1(\vec{u_1} + \vec{\beta}) + \lambda_2(\vec{u_2} + \vec{\beta}) + ... + \lambda_k(\vec{v_k} + \vec{\beta}) \end{aligned}$$

So $\vec{y} \in aff(B')$

□

## 6.3 Convex Polytope

**Definition 6.3.1** (Convex body)**.** A **Convex body** $K$ can be defined as: for any two points $\vec{x}, \vec{y} \in K$ where $K \subseteq \mathbb{R}^d$, $K$ also contains the straight line segment $[\vec{x}, \vec{y}] = \{\lambda\vec{x} + (1-\lambda)\vec{y} | 0 \leq \lambda \leq 1\}$, or $[\vec{x}, \vec{y}] \subseteq K$.

**Example 6.3.2.** The graph in Figure 6.3(a) is a convex body since for any $\vec{x}, \vec{y} \in K, [\vec{x}, \vec{y}] \subseteq K$. However, the graph in Figure 6.3(b) is not a convex body since there exists an $\vec{x}, \vec{y} \in K'$ such that $[\vec{x}, \vec{y}] \nsubseteq K'$.

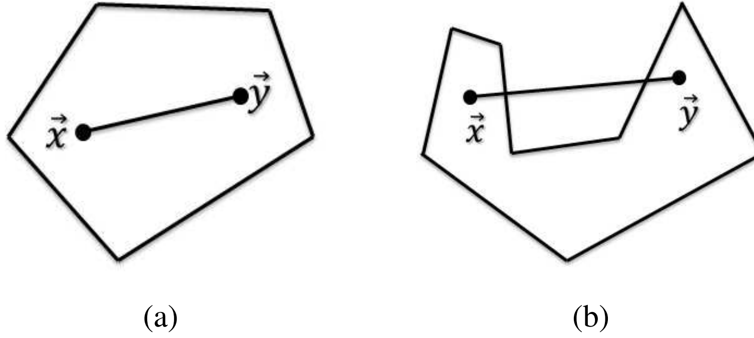



Figure 6.3: (a) An example of convex body (b) An example of non-convex body

**Definition 6.3.3** (Convex Hull)**.** For any $K \subseteq \mathbb{R}^d$, the "smallest" convex set containing $K$, called the **convex hull** of $K$, can be constructed as the intersection of all convex sets that contain $K$:

$$conv(K) := \bigcap\{U | U \subseteq \mathbb{R}^d, K \subseteq U, U \text{ is convex}\}.$$

**Definition 6.3.4** (Convex Hull)**.** An alternative definition of **convex hull** is that let $K = \{\vec{u_1}, \vec{u_2}, \ldots, \vec{u_k}\}$,

$$\widehat{conv}(K) = \{\vec{u} | \vec{u} = \lambda\vec{u_1} + \cdots \lambda_k\vec{u_k}, \sum_{i=0}^{k} \lambda_i = 1, \lambda_i \geq 0\}$$

**Claim 6.3.5.** $conv(K) = \widehat{conv}(K)$.

*Proof.* Since $conv(K)$ is the "smallest" convex set containing $K \subseteq \mathbb{R}^d$, $conv(K) \subseteq \widehat{conv}(K)$.

Next, we will use induction to prove that $\widehat{conv}(K) \subseteq conv(K)$ .

The base case is clearly correct, when $K = \{\vec{u_1}\} \subseteq \mathbb{R}^d$, $\widehat{conv}(K) \subseteq conv(K)$.

We now assume that for an arbitrary $k-1$ where $K = \{\vec{u_1}, \ldots, \vec{u_{k-1}}\} \subseteq \mathbb{R}^d$, we have $\widehat{conv}(K) \subseteq conv(K)$. On condition $k$, fix an arbitrary $\vec{u} \in \widehat{conv}(K)$ such that $\vec{u} = \lambda\vec{u_1} + \cdots \lambda_k\vec{u_k}$. Actually,

$$\vec{u} = (1-\lambda_k)\left(\frac{\lambda_1}{1-\lambda_k}\vec{u_1} + \cdots + \frac{\lambda_{k-1}}{1-\lambda_k}\vec{u_{k-1}}\right) + \lambda_k\vec{u_k},$$

where by hypothesis $\vec{u'} = \frac{\lambda_1}{1-\lambda_k}\vec{u_1} + \cdots + \frac{\lambda_{k-1}}{1-\lambda_k}\vec{u_{k-1}} \in conv(K)$, since the sum of coefficients is 1. Because $\vec{u'}, \vec{u_k} \in conv(K)$ and $conv(K)$ is convex, then $\vec{u} = (1-\lambda_k)\vec{u'} + \lambda_k\vec{u_k} \in [\vec{u'}, \vec{u_k}] \subseteq conv(K)$. Thus, by induction on $k$, $\widehat{conv}(K) \subseteq conv(K)$.

Hence, $conv(K) = \widehat{conv}(K)$. □

## 6.4 Polytope

In this section, we are going to introduce the concept of a V-Polytope and an H-Polytope.

**Definition 6.4.1** (V-polytope)**.** A **V-polytope** is convex hull of a finite set of points.

Before we give a definition of an H-Polytope, we need to give the definition of a hyperplane and an H-Polyhedron.

**Definition 6.4.2** (Hyperplane)**.** A **hyperplane** in $\mathbb{R}^d$ can be described as the set of points $x = (x_1, x_2, ..., x_d)^T$ which are solutions to $\alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_d x_d = \beta$, for fixed $\alpha_1, \alpha_2, \ldots, \alpha_d, \beta \in \mathbb{R}$.

**Example 6.4.3.** $x_1 + x_2 = 1$ is a hyperplane.

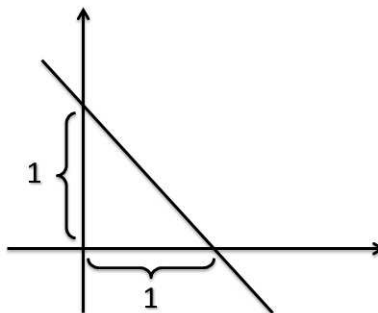


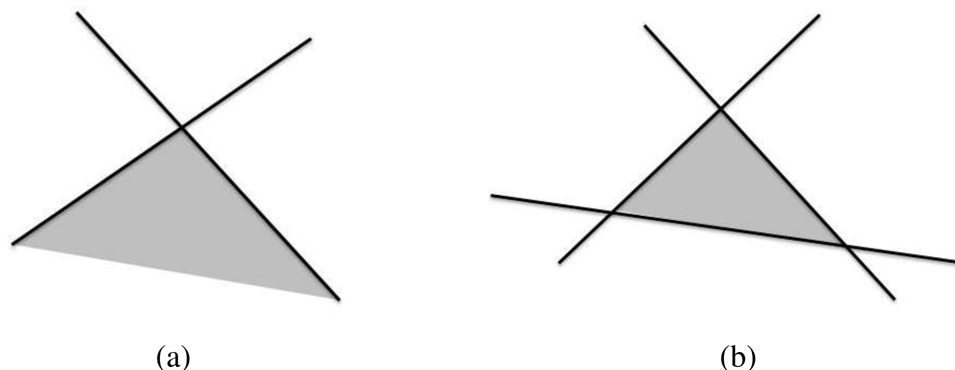

Figure 6.4: An example of hyperplane

**Definition 6.4.4** (Half-space)**.** A **Half-space** in $\mathbb{R}^d$ is the set of points $x = (x_1, x_2, ...x_d)^T$, which satisfies $\alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_d x_d \leq \beta$. for fixed $\alpha_1, \alpha_2, \ldots, \alpha_d, \beta \in \mathbb{R}$.

**Definition 6.4.5** (H-polyhedron)**.** An **H-polyhedron** is the intersection of a finite number of closed half-spaces.

**Definition 6.4.6.** An **H-polytope** is a bounded H-polyhedron.

A simple question follows: are the two types of polytopes equivalent? The answer is yes. However, the proof, Fourier-Motzkin elimination, is complicated.



(a) (b)

Figure 6.5: (a) H-polyhedron and (b)H-polytope

## 6.5 Linear Programming

The goal of a Linear program (LP) is to optimize linear functions over polytopes. Examples are to *minimize/maximize* $\alpha_1 x_1 + \alpha_2 x_2 + \cdots + \alpha_n x_n$ where $x_1, \ldots, x_n$ are variables and $\alpha_1, \ldots, \alpha_n$ are constants that are subject to a set of inequalities:

$$\alpha_{11}x_1 + \alpha_{12}x_2 + \cdots + \alpha_{1n}x_n \leq \beta_1$$
$$\alpha_{21}x_1 + \alpha_{22}x_2 + \cdots + \alpha_{2n}x_n \leq \beta_2$$
$$\vdots$$

It can be solved in polynomial time if $x_1, \ldots x_n$ are real numbers, but not when they are integers. Previously, it was believed that LPs were another complexity class between P and NP. However, Linear Programs can, in fact, be solved in time polynomial in their input size. The first polynomial time algorithm for LPs was the Ellipsoid algorithm. Since then, there has been a substantial amount of work in other polynomial time algorithms, e.g. Interior Point methods.

**Example 6.5.1.** Minimize $f(\vec{x}) = 7x_1 + x_2 + 5x_3$, subject to

$$\begin{aligned} x_1 + x_2 + 3x_3 &\geq 10 \\ 5x_1 + 2x_2 - x_3 &\geq 6 \\ x_1,\ x_2,\ x_3 &\geq 0 \end{aligned}$$

Every $\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$ that satisfies the constraints is called *feasible solution.*

**Question:** Can you "prove" $\exists \vec{x^*}$ that makes $f(\vec{x^*}) \geq 30$?

**Answer:** Yes. Here is such an $\vec{x^*}$: $\vec{x^*} = \begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix}$, $f(\vec{x^*}) = 30$.

To disprove a proposition or prove a lower bound, then we may need to come up with different combinations of inequalities, such as

$$7x_1 + x_2 + 5x_3 \geq (x_1 - x_2 + 3x_3) + (5x_1 + 2x_2 - x_3) \geq 16.$$

# Chapter 7

# Forms of Linear Programming

*We describe the different forms of Linear Programs (LPs), including the standard and canonical forms as well as how to do transform between them.*

## 7.1 Forms of Linear Programming

In the previous lecture, we introduced the notion of Linear Programming by giving a simple example. Linear Programs are commonly written in two forms: standard form and canonical form.

Recall from the previous lecture that an LP is a set of linear inequalities together with an objective function. Each linear inequality takes the form:

$$a_1x_1 + a_2x_2 + ... + a_nx_n\{\leq, =, \geq\}b$$

There may be many such linear inequalities. The objective function takes the form:

$$c_1x_1 + c_2x_2 + ... + c_nx_n$$

The goal of Linear Programming is to maximize or minimize an objective function over the assignments that satisfy all inequalities. The general form of an LP is:

$$\begin{array}{rc} \text{maximize/minimize} & \vec{c}^T\vec{x} \\ \text{subject to} & \text{equations} \\ & \text{inequalities} \\ & \text{non-negative vers clause} \\ & \text{unconstrained variable example:} x_i \gtreqless 0 \end{array}$$

There are a couple different ways to write a Linear Program.

**Definition 7.1.1** (Canonical Form)**.** An LP is said to be in canonical form if it is written as

$$\begin{array}{rl} \text{minimize} & \vec{c}^T\vec{x} \\ \text{subject to} & \mathbf{A}\vec{x} \geq \vec{b} \\ & \vec{x} \geq \vec{0} \end{array}$$

**Definition 7.1.2** (Standard Form)**.** An LP is said to be in standard form if it is written as

$$\begin{array}{rc} \text{minimize} & \vec{c}^T\vec{x} \\ \text{subject to} & \mathbf{A}\vec{x} = \vec{b} \\ & \vec{x} \geq \vec{0} \end{array}$$

# 7.2 Linear Programming Form Transformation

We present transformations that allow us to change an LP written in the general form into one written in the standard form, and vice versa.

1. **STEP1: From General form into Canonical form.**

$$\begin{array}{rcl} \text{maximize } \vec{c}^T\vec{x} & \Rightarrow & -\text{minimize } -\vec{c}^T\vec{x} \\ \vec{a}^T\vec{x} = b & \Rightarrow & \vec{a}^T\vec{x} \leq b \text{ and } \vec{a}^T\vec{x} \geq b \\ \vec{a}^T\vec{x} \leq b & \Rightarrow & -\vec{a}^T\vec{x} \geq -b \\ x_i \gtrless 0 & \Rightarrow & x_i^+ - x_i^-, x_i^+, x_i^- \geq 0 \end{array}$$

2. **STEP2: From Canonical form into Standard form**

$$\vec{a}^T\vec{x} \geq b \Rightarrow \vec{a}^T\vec{x} - y = b, y \geq 0$$

**Example 7.2.1.** Suppose we have an LP written in the general form, and we wish to transform it into the standard form:

$$\begin{array}{rrcl} \text{maximize} & 2x_1 + 5x_2 & & \\ \text{subject to} & x_1 + x_2 & \leq & 3 \\ & x_2 & \geq & 0 \\ & x_1 & \gtrless & 0 \end{array}$$

The standard form of the LP is:

$$\begin{array}{rrcl} \text{minimize} & -(2x_1^+ - 2x_1^- + 5x_2) & & \\ \text{subject to} & x_1^+ - x_1^- + x_2 + y & = & 3 \\ & x_1^+, x_1^-, x_2, y & \geq & 0 \end{array}$$

# Chapter 8

# Linear Programming Duality

*In this lecture, we introduce the dual of a Linear Program (LP) and give a geometric proof of the Linear Programming Duality Theorem from elementary principles. We also introduce the concept of complementary slackness conditions, which we use to characterize optimal solutions to LPs and their duals. This material finds many applications in subsequent lectures.*

## 8.1 Primal and Dual Linear Program

### 8.1.1 Primal Linear Program

Consider an arbitrary Linear Program, which we will call *primal* in order to distinguish it from the dual (introduced later on). In what follows, we assume that the primal Linear Program is a maximization problem, where variables take non-negative values.

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ for } i = 1, 2, \ldots, m \\ & x_j \geq 0 \;\text{ for } j = 1, 2, \ldots, n \end{aligned}$$

The above Linear Program can be rewritten as:

$$\text{maximize } \vec{c}^T \vec{x} \tag{8.1}$$

$$\text{subject to } \; A\vec{x} \leq \vec{b} \tag{8.2}$$

$$\vec{x} \geq \vec{0} \tag{8.3}$$

An example:

$$\begin{aligned}
\text{maximize} \quad & 3x_1 + x_2 + 3x_3 && (8.4)\\
\text{subject to} \quad & x_1 + x_2 + 3x_3 \le 30 && (8.5)\\
& 2x_1 + 2x_2 + 5x_3 \le 24 && (8.6)\\
& 4x_1 + x_2 + 2x_3 \le 36 && (8.7)\\
& x_1, x_2, x_3 \ge 0 && (8.8)
\end{aligned}$$

### 8.1.2 Dual Linear Program

Suppose that one wants to know whether there is a flow in a given network of value $\geq \alpha$. If such a flow exists, then we have a *certificate* of small size that *proves* that such a thing exists: the certificate is a flow $f$ such that $\alpha = v(f)$. Now, what if one wants to know whether or not every flow has value $< \alpha$? One certificate we can give to prove this is to list all flows (say, of integer values). But this certificate is far too large. From the max-flow min-cut theorem we know that the weight of every cut is an upper bound to the maximum flow. Observe that for this, we just list the cut, which is a small certificate. Note that the "size" of the certificate is one thing. We care both about the size and about the time required to verify using this certificate.

The above relation between max-flow and min-cut is called duality. The concept of duality appears to be very useful in the design of efficient combinatorial algorithms. As we have mentioned before, Linear Programming can be seen as a kind of "unification theory" for algorithms. Part of its classification as such a thing is because there is a systematic way of obtaining duality characterizations.

Any primal Linear Program can be converted into a dual linear program:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{m} b_i y_i \\
\text{subject to} \quad & \sum_{i=1}^{m} a_{ij} y_i \ge c_i \text{ for } j = 1, 2, \ldots, n \\
& y_i \ge 0 \text{ for } i = 1, 2, \ldots, m
\end{aligned}$$

The above Linear Program can be rewritten as:

$$\begin{aligned}
\text{minimize} \quad & \vec{b}^T \vec{y} && (8.9)\\
\text{subject to} \quad & A^T \vec{y} \ge \vec{c} && (8.10)\\
& \vec{y} \ge \vec{0} && (8.11)
\end{aligned}$$

The primal Linear Program example given in (8.4) - (8.8) can be converted to the following dual form:

$$\begin{aligned} \text{minimize } & 30y_1 + 24y_2 + 36y_3 \\ \text{subject to } & y_1 + 2y_2 + 4y_3 \geq 3 \\ & y_1 + 2y_2 + y_3 \leq 1 \\ & 3y_1 + 5y_2 + 2y_3 \leq 2 \\ & y_1, y_2, y_3 \geq 0 \end{aligned}$$

As we shall see, in cases when both the primal LP and the dual LP are feasible and bounded, the objective value of the dual form gives a bound on the objective value of the primal form. Furthermore, we shall see the two optimal objective values are actually equivalent!

## 8.2 Weak Duality

**Theorem 8.2.1.** *(Weak Duality) Let $\vec{x} = (x_1, x_2, \ldots, x_n)$ be any feasible solution to the primal Linear Program and let $\vec{y} = (y_1, y_2, \ldots, y_m)$ be any feasible solution to the dual Linear Program. Then $\vec{c}^T\vec{x} \leq \vec{b}^T\vec{y}$, which is to say*

$$\sum_{j=1}^{n} c_j x_j \leq \sum_{i=1}^{m} b_i y_i$$

*Proof.* We have

$$\begin{aligned} \sum_{j=1}^{n} c_j x_j &\leq \sum_{j=1}^{n} (\sum_{i=1}^{m} a_{ij} y_i) x_j \\ &= \sum_{i=1}^{m} (\sum_{j=1}^{n} a_{ij} x_j) y_i \\ &\leq \sum_{i=1}^{m} b_i y_i \end{aligned}$$

□

**Corollary 8.2.2.** *Let $\vec{x} = (x_1, x_2, \ldots, x_n)$ be a feasible solution to the primal Linear Program, and let $\vec{y} = (y_1, y_2, \ldots, y_m)$ be a feasible solution to the dual Linear Program. If*

$$\sum_{j=1}^{n} c_j x_j = \sum_{i=1}^{m} b_i y_i$$

*then $\vec{x}$ and $\vec{y}$ are optimal solutions to the primal and dual Linear Programs, respectively.*

*Proof.* Let $t = \sum_{j=1}^{n} c_j x_j = \sum_{i=1}^{m} b_i y_i$. By the Weak Duality Theorem we know the objective value of the primal Linear Program is upper bounded by $t$. If we set $x$ to be such that it meets its upper bound $t$, then $x$ is an optimal solution. The proof for $y$ is similar. □

## 8.3 Farkas' Lemma

Farkas lemma is an essential theorem which tells us that we can have small certificates of *infeasibilty* of a system of linear inequalities. Geometrically this certificate is a hyperplane which separates the feasible region of a set of constraints from a point that lies outside this region. Although the geometric intuition is clear (and straightforward), it takes a little bit of work to make it precise. The key part of the proof of Farkas lemma is the following geometric theorem.

### 8.3.1 Projection Theorem

**Theorem 8.3.1.** *(Projection Theorem) Let $K$ be a closed, convex and non-empty set in $\mathbb{R}^n$, and $\vec{b} \in \mathbb{R}^n$, $\vec{b} \notin K$. Define projection $\vec{p}$ of $\vec{b}$ onto $K$ to be $\vec{x} \in K$ such that $\|\vec{b} - \vec{x}\|$ is minimized. Then for all $\vec{z} \in K$, we have $(\vec{b} - \vec{p})^T(\vec{z} - \vec{p}) \leq \vec{0}$.*

*Proof.* Let $\vec{u} = \vec{b} - \vec{p}$, $\vec{v} = \vec{z} - \vec{p}$. Suppose $\vec{u} \cdot \vec{v} > 0$, then

1) If $\|\vec{v}\| \leq \frac{\vec{u}\cdot\vec{v}}{\|\vec{v}\|}$, then

$$\begin{aligned}(\vec{b} - \vec{z})^2 &= (\vec{u} - \vec{v})^2 \\ &= \vec{u}^2 - 2\vec{u} \cdot \vec{v} + \vec{v}^2 \\ &\leq \vec{u}^2 - \vec{u} \cdot \vec{v} \\ &< \vec{u}^2 \\ &= (\vec{b} - \vec{p})^2\end{aligned}$$

which means $\|\vec{b} - \vec{z}\| < \|\vec{b} - \vec{p}\|$.

2) If $\|\vec{v}\| > \frac{\vec{u}\cdot\vec{v}}{\|\vec{v}\|}$, and because $K$ is convex, $\vec{p} \in K$ and $\vec{z} = \vec{p} + \vec{v} \in K$, we know $\vec{z'} = \vec{p} + \frac{\vec{u}\cdot\vec{v}}{\|\vec{v}\|^2}\vec{v} \in K$. Then

$$\begin{aligned}(\vec{b} - \vec{z'})^2 &= (\vec{u} - \frac{\vec{u} \cdot \vec{v}}{\|\vec{v}\|^2}\vec{v})^2 \\ &= \vec{u}^2 - \frac{(\vec{u} \cdot \vec{v})^2}{\|\vec{v}\|^2} \\ &< \vec{u}^2 \\ &= (\vec{b} - \vec{p})^2\end{aligned}$$

which means $\|\vec{b} - \vec{z'}\| < \|\vec{b} - \vec{p}\|$. □

### 8.3.2 Farkas' Lemma

**Lemma 8.3.2.** *(Farkas' Lemma) One and only one of the following two assertions holds:*

*1. $A^T\vec{y} = \vec{c}, \vec{y} \geq \vec{0}$ has a solution.*

*2. $A\vec{x} \leq \vec{0}$ and $\vec{c}^T\vec{x} > 0$ has a solution.*

*Proof.* First, we show the two assertions cannot hold at the same time. If so, we have

$$\vec{c}^T\vec{x} = \vec{y}^T A\vec{x}$$

We know $\vec{y}^T \geq \vec{0}$ and $A\vec{x} \leq \vec{0}$. So $\vec{c}^T\vec{x} = \vec{y}^T A\vec{x} \leq 0$. But this contradicts $\vec{c}^T\vec{x} > 0$.

Next, we show at least one of the two assertions holds. Specifically, we show if Assertion 1 doesn't hold, then Assertion 2 must hold.

Assume $A^T\vec{y} = \vec{c}, \vec{y} \geq \vec{0}$ is not feasible. Let $K = \{A^T\vec{y} : \vec{y} \geq \vec{0}\}$ (which is obviously convex). Then $\vec{c} \notin K$. Let $\vec{p} = A^T\vec{w}(\vec{w} \geq \vec{0})$ be the projection of $\vec{c}$ onto $K$. Then from the Projection Theorem we know that

$$(\vec{c} - A^T\vec{w})^T(A^T\vec{y} - A^T\vec{w}) \leq 0 \text{ for all } \vec{y} \geq \vec{0} \tag{8.12}$$

Define $\vec{x} = \vec{c} - \vec{p} = \vec{c} - A^T\vec{w}$. Then

$$\vec{x}^T A^T(\vec{y} - \vec{w}) \leq 0 \text{ for all } \vec{y} \geq \vec{0}$$
$$(\vec{y} - \vec{w})^T A\vec{x} \leq 0 \text{ for all } \vec{y} \geq \vec{0}$$

Let $\vec{e_i}$ be the $n$ dimensional vector that has 1 in its $i$-th component and 0 everywhere else. Take $\vec{y} = \vec{w} + \vec{e_i}$. Then

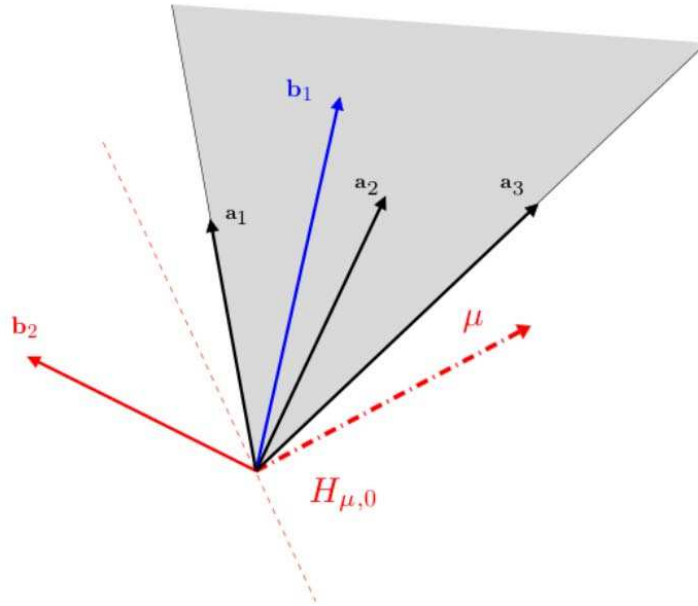
$$(A\vec{x})_i = \vec{e_i}^T A\vec{x} \leq 0 \text{ for } i = 1, 2, \ldots, m \tag{8.13}$$

Thus, each element of $A\vec{x}$ is non-positive, which means $A\vec{x} \leq \vec{0}$.

Now, $\vec{c}^T\vec{x} = (\vec{p} + \vec{x})^T\vec{x} = \vec{p}^T\vec{x} + \vec{x}^T\vec{x}$. Setting $\vec{y} = \vec{0}$ in (8.13), we have $-\vec{w}^T A\vec{x} = -\vec{p}^T\vec{x} \leq \vec{0}$, so $\vec{p}^T\vec{x} \geq 0$. Since $\vec{c} \neq \vec{p}$, $\vec{x} \neq \vec{0}$, so $\vec{x}^T\vec{x} > 0$. Therefore, $\vec{c}^T\vec{x} > 0$. □

### 8.3.3 Geometric Interpretation

If $\vec{c}$ lies in the cone formed by the column vectors of $A^T$, then Assertion 1 is satisfied. Otherwise, we can find a hyperplane which contains the origin and separates $\vec{c}$ from the cone, and thus Assertion 2 is satisfied. These two cases are depicted below.




The $\vec{a_i}$'s correspond to the column vectors of $A^T$. $\vec{b_1}$ and $\vec{b_2}$ correspond to $\vec{c}$ in Assertion 1 and Assertion 2, respectively.

### 8.3.4 More on Farkas Lemma

There are several variants of Farkas Lemma in the literature, as well as interpretations that one may give. In our context Farkas Lemma is being used to characterize linear systems that are infeasible. Among others, in some sense, this lemma provides a succinct way for witnessing infeasibility.

Now we give an variant of Farkas Lemma:

**Lemma 8.3.3.** *One and only one of the following two assertions holds:*
*1.* $A\vec{x} \geq \vec{b}$ *has a solution.*
*2.* $A^T\vec{y} = 0$, $\vec{b}^T\vec{y} > 0$, $\vec{y} \geq \vec{0}$ *has a solution.*

**Exercise:** Prove Lemma 8.3.3.

## 8.4 Strong Duality

Roughly speaking, Farkas Lemma and the Linear Programming (LP) Duality theorem are the same thing. The "hard" job has already been done when proving Farkas Lemma. Note that in Farkas we have a system of linear inequalities whereas in LP Duality we have a system of linear inequalities over which we wish to optimize a linear function. What remains to be done is to play around with formulas such that we translate LP Duality to Farkas. To do that we will somehow "encode" the function we optimize in the Linear Program inside the constraints of the system of linear inequalities.

**Theorem 8.4.1** (Strong Duality)**.** *Let $\vec{x}$ be a feasible solution to the primal Linear Program and let $\vec{y}$ be a feasible solution to the dual Linear Program (8.9)-(8.11). Then $\vec{x}$ and $\vec{y}$ are optimal*

$$\Longleftrightarrow$$

$$\vec{c}^T\vec{x} = \vec{b}^T\vec{y}$$

*Proof.* ($\Leftarrow$) See Corollary (8.2.2).
($\Rightarrow$) Since we already have $\vec{c}^T\vec{x} \leq \vec{b}^T\vec{y}$ by the Weak Duality Theorem, we just need to prove $\vec{c}^T\vec{x} \geq \vec{b}^T\vec{y}$. Let $z^* = \vec{c}^T\vec{x}$, $w^* = \vec{b}^T\vec{y}$.

**Claim 8.4.2.** *There exists a solution of dual of value at most $z^*$, i.e.,*

$$\exists\vec{y} : A^T\vec{y} \geq c, \vec{y} \geq \vec{0}, \vec{b}^T\vec{y} \leq z^*$$

*Proof.* We wish to prove that there is a $y$ satisfying:

$$\begin{pmatrix} A^T \\ -\vec{b}^T \end{pmatrix} \geq \begin{pmatrix} \vec{c} \\ -z^* \end{pmatrix}$$

Assume the claim is wrong. Then the variant of Farkas' Lemma implies that

$$\begin{aligned}
\left(A, -\vec{b}\right) \begin{pmatrix} \vec{x} \\ \lambda \end{pmatrix} &= \vec{0} \\
\left(\vec{c}^T, -z^*\right) \begin{pmatrix} \vec{x} \\ \lambda \end{pmatrix} &> 0 \\
\vec{x} &\geq \vec{0} \\
\lambda &\geq 0
\end{aligned}$$

has a solution. That is, there exist nonnegative $\vec{x}$, $\lambda$ such that

$$\begin{aligned}
A\vec{x} - \lambda\vec{b} &= \vec{0} \\
\vec{c}^T\vec{x} - \lambda z^* &> 0
\end{aligned}$$

Case I: $\lambda > 0$. Then there exists nonnegative $\vec{x}$ such that $A(\frac{\vec{x}}{\lambda}) = \vec{b}$, $\vec{c}^T(\frac{\vec{x}}{\lambda}) > z^*$. This contradicts the minimality of $z^*$ for the primal.

Case II: $\lambda = 0$. Then there exists nonnegative $\vec{x}$ such that $A\vec{x} = 0$, $\vec{c}^T\vec{x} > 0$. Take any feasible solution $\vec{x'}$ for primal LP. Then for every $\mu \geq 0$, $\vec{x'} + \mu\vec{x}$ is feasible for primal LP, since

1. $\vec{x'} + \mu\vec{x} \geq \vec{0}$ because $\vec{x'} \geq \vec{0}$, $\vec{x} \geq \vec{0}$, $\mu \geq 0$.
2. $A(\vec{x'} + \mu\vec{x}) = A\vec{x'} + \mu A\vec{x} = \vec{b} + \mu\vec{0} = \vec{b}$.
   But $\vec{c}^T(\vec{x'} + \mu\vec{x}) = \vec{c}^T\vec{x'} + \vec{c}^T\mu\vec{x} \to \infty$ as $\mu \to \infty$, This contradicts the assumption that the primal has finite solution.

The above claim shows that $z^* \geq w^*$. And we already have $z^* \leq w^*$ by Weak Duality Theorem, so $z^* = w^*$ □

□

## 8.5 Complementary Slackness

**Theorem 8.5.1.** *(Complementary Slackness) Let $\vec{x}$ be an optimal solution to the primal Linear Program given in (8.1)-(8.3), and let $\vec{y}$ be an optimal solution to the dual Linear Program given in (8.9)-(8.11). Then the following conditions are necessary and sufficient for $\vec{x}$ and $\vec{y}$ to be optimal:*

$$\sum_{i=1}^{m} a_{ij}y_i = c_j \textit{ or } x_j = 0 \textit{ for } j = 1, 2, \ldots, n \tag{8.14}$$

$$\sum_{j=1}^{n} a_{ij}x_j = b_i \textit{ or } y_i = 0 \textit{ for } i = 1, 2, \ldots, m \tag{8.15}$$

*Proof.* (Sufficient) Define $S_j = \{j : x_j > 0\}$ and $S_i = \{i : y_j > 0\}$. Then we have

$$\begin{aligned}
\vec{c}^T\vec{x} &= \sum_{j=1}^{n} c_j x_j \\
&= \sum_{j\in S_j}^{n} c_j x_j \\
&= \sum_{j\in S_j}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i\right) x_j \\
&= \sum_{j\in S_j}^{n} \left(\sum_{i\in S_i}^{m} a_{ij} y_i\right) x_j \\
\vec{b}^T\vec{y} &= \sum_{i=1}^{m} b_i y_i \\
&= \sum_{i\in S_i}^{m} b_i y_i \\
&= \sum_{i\in S_i}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j\right) y_i \\
&= \sum_{i\in S_i}^{m} \left(\sum_{j\in S_j}^{n} a_{ij} x_j\right) y_i
\end{aligned}$$

Therefore $\vec{c}^T\vec{x} = \vec{b}^T\vec{y}$. By the Strong Duality Theorem, we know $\vec{x}$ and $\vec{y}$ must be optimal solutions.
(Necessary) Suppose $\vec{x}$ and $\vec{y}$ are optimal solutions, we have

$$\begin{aligned}
\vec{c}^T\vec{x} &\le (A^T\vec{y})^T\vec{x} \\
&= \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i\right) x_j \\
\vec{b}^T\vec{y} &\ge (A\vec{x})^T\vec{y} \\
&= \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} x_j\right) y_i \\
\vec{c}^T\vec{x} &= \vec{b}^T\vec{y}
\end{aligned}$$

so

$$\vec{c}^T\vec{x} = \vec{b}^T\vec{y} = \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i\right) x_j \tag{8.16}$$

Suppose there exists $j'$ such that $\sum_{i=1}^{m} a_{ij'} y_i > c_j'$ and $x_j' > 0$. Then

$$\begin{aligned}
\vec{c}^T \vec{x} &= \sum_{j=1}^{n} c_j x_j \\
&= \sum_{j \neq j'} c_j x_j + c_{j'} x_j' \\
&< \sum_{j \neq j'} \left( \sum_{i=1}^{m} a_{ij} y_i \right) x_j + \left( \sum_{i=1}^{m} a_{ij'} y_i \right) x_j' \\
&= \sum_{j=1}^{n} \left( \sum_{i=1}^{m} a_{ij} y_i \right) x_j
\end{aligned}$$

which contradicts optimality. Therefore equation (8.14) must be satisfied. Equation (8.15) can be proved in a similar fashion. □

# Chapter 9

# Simplex Algorithm and Ellipsoid Algorithm

*In this chapter, we will introduce two methods for solving Linear Programs: the simplex algorithm and the ellipsoid algorithm. To better illustrate these two algorithms, some mathematical background is also introduced. The simplex algorithm doesn't run in polynomial time, but works well in practice. The ellipsoid runs in polynomial time, but usually performs poorly in practice.*

## 9.1 More on Basic Feasible Solutions

### 9.1.1 Assumptions and conventions

Since any LP instance could be converted to maximized standard form, we'll use the standard form of an LP to show some useful properties in the following lectures. We can write the maximized standard form of LP compactly as

$$\text{maximize } \vec{c}^T\vec{x} \tag{9.1}$$
$$\text{subject to } \; A\vec{x} = \vec{b} \tag{9.2}$$
$$\vec{x} \geq \vec{0} \tag{9.3}$$

We only consider the case when the LP (9.1) - (9.3) is feasible and bounded. In such cases, the feasible region is a polytope.

Since we have seen how to convert a Linear Program from normal forms to standard form in the previous lectures, we can simplify our notation by setting $A$ to be an $m \times n$ matrix, $\vec{c}$ and $\vec{x}$ as $n$-dimensional vectors, and $\vec{b}$ as an $m$-dimensional vector in (9.1) - (9.3).

It's easy to see that $m < n$. WLOG we suppose $rank(A) = m$ so that all constraints are useful.

### 9.1.2 Definitions

Suppose the Linear Program (9.1) - (9.3) is feasible and bounded. Denote the feasible region by $S$. It's easy to see that $S$ is non-empty and convex. Then we have the following definitions:

**Definition 9.1.1** (Corner)**.** A point $\vec{x}$ in $S$ is said to be a corner if there is an $n$-dimensional vector $\vec{w}$ and a scalar $t$ such that

1. $\vec{w}^T\vec{x} = t$,

2. $\vec{w}^T\vec{x}' > t$ for all $\vec{x}' \in S - \{\vec{x}\}$.

Intuitively, this definition says that all points in $S$ are on the same side of the hyperplane $H$ defined by $\vec{w}$ and $H \cap S = \{\vec{x}\}$. Also note that $\vec{w}^T\vec{y} \geq t$ for all $\vec{y} \in S$.

**Definition 9.1.2** (Extreme point)**.** A point $\vec{x}$ in $S$ is said to be an extreme point if there are no two distinct points $\vec{u}$ and $\vec{v}$ in $S$ such that $\vec{x} = \lambda\vec{u} + (1-\lambda)\vec{v}$ for some $0 < \lambda < 1$.

The above definition says there is no line segment in $S$ with $\vec{x}$ in its interior.

**Definition 9.1.3** (Basic Feasible Solution)**.** Let $B$ be any non-singular $m \times m$ sub-matrix made up of columns of $A$. If $\vec{x} \in S$ and all $n - m$ components of $\vec{x}$ not associated with columns of $B$ are equal to zero, we say $\vec{x}$ is a Basic Feasible Solution to $A\vec{x} = \vec{b}$ with respect to basis $B$.

Intuitively, this says that the columns of $B$ span the whole $m$-dimensional vector space. Since $\vec{b}$ is also $m$-dimensional, $\vec{b}$ is a linear composition of the columns of $B$. Therefore, there is a solution $\vec{x}$ with all $n - m$ components not associated with columns of $B$ set to zero.

### 9.1.3 Equivalence of the definitions

**Lemma 9.1.4.** *If $\vec{x}$ is a corner, then $\vec{x}$ is an extreme point.*

*Proof.* Suppose $\vec{x}$ is a corner, but not an extreme point. Then there exists $\vec{y} \neq \vec{0}$ such that $\vec{x}+\vec{y} \in S$ and $\vec{x} - \vec{y} \in S$. So $\vec{w}^T(\vec{x}+\vec{y}) \geq t$ and $\vec{w}^T(\vec{x}-\vec{y}) \geq t$. Since $\vec{w}^T\vec{x} = t$, we have $\vec{w}^T\vec{y} \geq 0$ and $\vec{w}^T\vec{y} \leq 0$ which implies that $\vec{w}^T\vec{y} = 0$. So $\vec{w}^T(\vec{x}+\vec{y}) = \vec{w}^T\vec{x} = t$. Since $\vec{x}+\vec{y} \in S$, $\vec{x}+\vec{y} = \vec{x}$ by the definition of corners. This means $\vec{y} = \vec{0}$, which is a contradiction. □

**Lemma 9.1.5.** *If $\vec{x}$ is an extreme point, then the columns $\{\vec{a_j} : x_j > 0\}$ of $A$ are linearly independent.*

*Proof.* WLOG suppose the columns are $\vec{a_1}, \vec{a_2}, \ldots, \vec{a_k}$. If they are not independent, then there are scalars $d_1, d_2, \ldots, d_k$ not all zero such that

$$d_1\vec{a_1} + d_2\vec{a_2} + \cdots + d_k\vec{a_k} = \vec{0} \tag{9.4}$$

WLOG we suppose $|d_j| < |\vec{x_j}|$ for $1 \leq j \leq k$. Let $\vec{d}$ be an n-dimensional vector such that $\vec{d} = (d_1, \ldots, d_k, 0, \ldots, 0)^T$. Then we know

$$A\vec{d} = d_1\vec{a_1} + d_2\vec{a_2} + \cdots + d_k\vec{a_k} + 0\vec{a_{k+1}} + \cdots + 0\vec{a_n} = \vec{0} \tag{9.5}$$

So $A(\vec{x}+\vec{d}) = A\vec{x} = \vec{b}$. Since $\vec{x}+\vec{d} = (x_1+d_1, \ldots, x_k+d_k, 0, \ldots, 0) \geq \vec{0}$, we have $\vec{x}+\vec{d} \in S$. Similarly, we have $\vec{x} - \vec{d} \in S$. Since $\vec{x} = \frac{1}{2}(\vec{x}+\vec{d}) + \frac{1}{2}(\vec{x}-\vec{d})$ and $\vec{d} \neq \vec{0}$, we see $\vec{x}$ is not an extreme point, which is a contradiction. □

**Lemma 9.1.6.** *If $\vec{x} \in S$ and the columns $\{\vec{a_j} : x_j > 0\}$ of $A$ are linearly independent, then $\vec{x}$ is a Basic Feasible Solution.*

*Proof.* WLOG suppose $\vec{x} = (x_1, \ldots, x_k, 0, \ldots, 0)$ where $x_j > 0$ for $1 \leq j \leq k$, which is to say, $B = \{\vec{a_1}, \ldots, \vec{a_k}\}$ are independent. Since $rank(A) = m$, we have $k \leq m$.

1) If $k = m$, we are done. This is because $B = [\vec{a_1} \ \cdots \ \vec{a_k}]$ is an $m \times m$ invertible sub-matrix of $A$.

2) If $k < m$, we can widen $B$ by incorporating more columns of $A$ while maintaining their independence. Finally, $B$ has $m$ columns. Now the components of $\vec{x}$ not corresponding to the columns in $B$ are still zero, so $\vec{x}$ is a Basic Feasible Solution. □

**Lemma 9.1.7.** *If $\vec{x}$ is a Basic Feasible Solution, then $\vec{x}$ is a corner.*

*Proof.* WLOG suppose $\vec{x} = (x_1, \ldots, x_m, 0, \ldots, 0)^T$ is a Basic Feasible Solution, which is to say $B = [\vec{a_1} \ \cdots \ \vec{a_m}]$ is an $m \times m$ non-singular sub-matrix of $A$. Set $\vec{w} = (0, \ldots, 0, 1, \ldots, 1)^T$ (with $m$ 0's and $n - m$ 1's) and $t = 0$. Then $\vec{w}^T \vec{x} = 0$. And $\vec{w}^T \vec{y} \geq 0$ for any $\vec{y} \in S$ since $\vec{y} \geq \vec{0}$.

If there exists $\vec{z} \in S$ such that $\vec{w}^T \vec{z} = 0$, then $\vec{z} = (z_1, \ldots, z_m, 0, \ldots, 0)^T$ and $\vec{b} = A\vec{z} = B[z_1 \ \cdots \ z_m]^T$. Since $\vec{b} = A\vec{x} = B[x_1 \ \cdots \ x_m]^T$, we have $[z_1 \ \cdots \ z_m]^T = [x_1 \ \cdots \ x_m]^T = B^{-1}\vec{b}$. So $\vec{z} = \vec{x}$.

Thus $\vec{w}^T \vec{x} = t$ and $\vec{w}^T \vec{x}' > t$ for all $\vec{x}' \in S - \{\vec{x}\}$, which proves $\vec{x}$ is a corner. □

Now we have the following theorem stating the equivalence of the various definitions:

**Theorem 9.1.8.** *Suppose the LP (9.1) - (9.3) is feasible with feasible region $S$. Then the following statements are equivalent:*

1. *$\vec{x}$ is a corner;*
2. *$\vec{x}$ is an extreme point;*
3. *$\vec{x}$ is a Basic Feasible Solution.*

*Proof.* From Lemma (9.1.4) - (9.1.7). □

### 9.1.4 Existence of Basic Feasible Solution

So far we have seen equivalent definitions for Basic Feasible Solutions. But when do we know when a BFS exists? The following theorem shows that a BFS exists if a feasible solution does.

**Theorem 9.1.9.** *If the Linear Program (9.1) - (9.3) is feasible then there is a BFS.*

*Proof.* Since the LP is feasible, the feasible region $S \neq \emptyset$. Let $\vec{x}$ be a feasible solution with a minimal number of non-zero components. If $\vec{x} = \vec{0}$, we are done since $\vec{0}$ is a BFS. Otherwise, set $I = \{i : x_i > 0\}$ then it's clear $I \neq \emptyset$. Suppose $\vec{x}$ is not basic, which is to say, $\{a_i : i \in I\}$ are not linearly independent. Then there exists $\vec{u} \neq \vec{0}$ with $u_i = 0$ for $i \notin I$ satisfying $A\vec{u} = \vec{0}$.

Consider $\vec{x}' = \vec{x} + \epsilon\vec{u}$. It's clear $A\vec{x}' = \vec{b}$ for every $\epsilon$. In addition, $\vec{x}' \geq \vec{0}$ for sufficiently small $\epsilon$. So $\vec{x}'$ is feasible. Increase or decrease $\epsilon$ until the first time one more components of $\vec{x}'$ hits 0. At that point, we have a feasible solution with fewer non-zero components, which is a contradiction. □

### 9.1.5 Existence of optimal Basic Feasible Solution

Suppose a Linear Program is feasible and finite, i.e. has a finite optimal value. Then we can show the LP has an optimal Basic Feasible Solution.

**Theorem 9.1.10.** *If the Linear Program (9.1) - (9.3) is feasible and finite, there is an optimal solution at an extreme point of the feasible region $S$.*

*Proof.* Since there exists an optimal solution, there exists an optimal solution $\vec{x}$ with a minimal number of non-zero components. Suppose $\vec{x}$ is not an extreme point, then $\vec{x} = \lambda\vec{u} + (1-\lambda)\vec{v}$ for some $\vec{u}, \vec{v} \in S$ and $0 < \lambda < 1$. Since $\vec{x}$ is optimal, we must have $\vec{c}^T\vec{u} \leq \vec{c}^T\vec{x}$ and $\vec{c}^T\vec{v} \leq \vec{c}^T\vec{x}$. Since $\vec{c}^T\vec{x} = \lambda\vec{c}^T\vec{u} + (1-\lambda)\vec{c}^T\vec{v}$, this implies $\vec{c}^T\vec{u} = \vec{c}^T\vec{v} = \vec{c}^T\vec{x}$. Thus $\vec{u}$ and $\vec{v}$ are also optimal.

Consider $\vec{x}' = \vec{x} + \epsilon(\vec{u} - \vec{v})$. The following statements are easy to check:

1. $\vec{c}^T\vec{x}' = \vec{c}^T\vec{x}$ for all $\epsilon$;
2. $x_i = 0 \Rightarrow u_i = v_i = 0 \Rightarrow x'_i = 0$;
3. $x_i > 0 \Rightarrow x'_i > 0$ for sufficiently small $\epsilon$.

Increase or decrease $\epsilon$ until the first time one more components of $\vec{x}'$ is zero. At that point, we have an optimal solution with fewer non-zero components, which is a contradiction. □

**Corollary 9.1.11.** *If the Linear Program (9.1) - (9.3) is feasible and finite, then there is an optimal Basic Feasible Solution.*

*Proof.* From Theorem (9.1.8) and Theorem (9.1.10). □

## 9.2 Simplex algorithm

### 9.2.1 The algorithm

The simplex algorithm is based on the existence of an optimal Basic Feasible Solution(BFS). The algorithm first chooses a BFS (or reports the LP is infeasible), then in each iteration, moves from the BFS to its neighbor (which differs from it in one column), and makes sure the objective value doesn't decrease when all non-basic variables are set to zero. This is called **pivoting**.

Consider the following example:

$$\begin{aligned}
z &= 27 + \frac{x_2}{4} + \frac{x_3}{2} - \frac{3x_6}{4} \\
x_1 &= 9 - \frac{x_2}{4} - \frac{x_3}{2} - \frac{x_6}{4} \\
x_4 &= 21 - \frac{3x_2}{4} - \frac{5x_3}{2} + \frac{x_6}{4} \\
x_5 &= 6 - \frac{3x_2}{2} - 4x_3 + \frac{x_6}{2}
\end{aligned}$$

In this LP, $\{x_1, x_4, x_5\}$ is a BFS. If we set non-basic variables to zero, the objective value is 27. After a pivoting with $x_3$ joining the BFS and $x_5$ leaving the BFS, it becomes:

$$
\begin{aligned}
z &= \frac{111}{4} + \frac{x_2}{16} - \frac{x_5}{8} - \frac{11x_6}{16} \\
x_1 &= \frac{33}{4} - \frac{x_2}{16} + \frac{x_5}{8} - \frac{5x_6}{16} \\
x_3 &= \frac{3}{2} - \frac{3x_2}{8} - \frac{x_5}{4} + \frac{x_6}{8} \\
x_4 &= \frac{69}{4} + \frac{3x_2}{16} + \frac{5x_5}{8} - \frac{x_6}{16}
\end{aligned}
$$

In this LP, $\{x_1, x_3, x_4\}$ is a BFS. If we set non-basic variables to zero, the objective value is $\frac{111}{4} > 27$. If we continue pivoting, either we reach the optimal BFS, or we conclude the LP is unbounded.

To prevent cycling, we can use Bland's rule, which says we should choose the variable with the smallest index for tie-breaking.

Interested readers should refer to Introduction to Algorithms, Chapter 29 for more details of the simplex algorithm. In particular, this will illustrate:

1. How to choose the initial BFS.
2. How to choose the pivoting element.
3. When to assert the LP is infeasible, unbounded or whether an optimal solution can found.

### 9.2.2 Efficiency

The simplex algorithm is not a polynomial time algorithm. Klee and Minty gave an example showing that the worst-case time complexity of simplex algorithm is exponential in 1972. But it is generally fast in real world applications.

## 9.3 Ellipsoid algorithm

In this section, we only consider when a finite feasible region exists.

### 9.3.1 History

If we formulate an LP in its decision version, then it's obvious LP $\in$ NP since a NTM can nondeterministically guess the solution. In addition, LP $\in$ co-NP since an NTM can guess the solution to the LP's dual. So LP $\in$ NP $\cap$ co-NP. However, it was not until Khachiyan posted his ellipsoid algorithm, a polynomial time algorithm, that people knew LP $\in$ P.

## 9.3.2 Mathematical background

### Carathéodory's theorem

**Theorem 9.3.1** (Carathéodory's theorem)**.** *If a point $\vec{x}$ of $\mathbb{R}^d$ lies in the convex hull of a set $P$, there is a subset $P'$ of $P$ consisting of $d+1$ or fewer points which are affinely independent such that $\vec{x}$ lies in the convex hull of $P$. Equivalently, $\vec{x}$ lies in an $r$-simplex with vertices in $P$, where $r \leq d$.*

*Proof.* Let $\vec{x}$ be a point in the convex hull of $P$. Then, $\vec{x}$ is a convex combination of a finite number of points in $P$:

$$\vec{x} = \sum_{j=1}^{k} \lambda_j x_j$$

where every $x_j$ is in $P$, every $\lambda_j$ is positive, and $\sum_{j=1}^{k} \lambda_j = 1$.

If $x_2 - x_1, \ldots, x_k - x_1$ are linearly independent, there are scalars $\mu_2, \ldots, \mu_k$ not all zero such that

$$\sum_{j=2}^{k} \mu_j (x_j - x_1) = 0$$

If $\mu_1$ is defined as $\mu_1 = -\sum_{j=2}^{k} \mu_j$, then $\sum_{j=1}^{k} \mu_j x_j = 0$ and $\sum_{j=1}^{k} \mu_j = 0$ and not all of the $\mu_j$'s are equal to zero. Therefore, at least one $\mu_j > 0$. Then,

$$\vec{x} = \sum_{j=1}^{k} \lambda_j x_j - \alpha \sum_{j=1}^{k} \mu_j x_j = \sum_{j=1}^{k} (\lambda_j - \alpha\mu_j) x_j$$

for any real $\alpha$. In particular, the equality will hold if $\alpha$ is defined as

$$\alpha = \min_{1 \leq j \leq k} \left\{ \tfrac{\lambda_j}{\mu_j} : \mu_j > 0 \right\} = \tfrac{\lambda_i}{\mu_i}.$$

Note that $\alpha > 0$, and for every $j$ between 1 and $k$, $\lambda_j - \alpha\mu_j \geq 0$. In particular, $\lambda_i - \alpha\mu_i = 0$ by definition of $\alpha$. Therefore,

$$\vec{x} = \sum_{j=1}^{k} (\lambda_j - \alpha\mu_j) x_j$$

where every $\lambda_j \alpha \mu_j$ is non-negative, their sum is one, and furthermore, $\lambda_i \alpha \mu_i = 0$. In other words, $\vec{x}$ is represented as a convex combination of one less point of $P$.

Note that if $k > d+1$, $x_2 - x_1, \ldots, x_k - x_1$ are always linearly dependent. Therefore, this process can be repeated until $\vec{x}$ is represented as a convex combination of $r \leq d+1$ points $x_{i_1}, \ldots, x_{i_r}$ in $P$. Since $x_{i_2} - x_{i_1}, \ldots, x_{i_r} - x_{i_1}$ are linearly independent, when this process terminates, $x_{i_1}, \ldots, x_{i_r}$ are affinely independent.

□

### Quadratic form

$\vec{x}^T A \vec{x}$ is called a quadratic form, where $\vec{x}$ is an $n$-dimensional vector and $A$ is a symmetric $n \times n$ matrix. If $A$ is positive definite, then $\vec{x}^T A \vec{x} = 1$ is an ellipsoid.

#### Volume of $n$-simplices

A set of $n+1$ affinely independent points $S \in \mathbb{R}^n$ make up an $n$-simplex. The volume of this $n$-simplex is

$$\frac{1}{n!}|det\begin{pmatrix}1 & 1 & \cdots & 1\\ p_0 & p_1 & \cdots & p_n\end{pmatrix}|$$

### 9.3.3 LP,LI and LSI

There are three kinds of problems that are related to each other.

**Question:** [Linear Programming/IP] Given an integer $m \times n$ matrix $A$, $m$-vector $\vec{b}$ and $n$-vector $\vec{c}$, either

(a) Find a rational $n$-vector $\vec{x}$ such that $\vec{x} \geq \vec{0}$, $A\vec{x} = \vec{b}$ and $\vec{c}^T\vec{x}$ is minimized subject to these conditions, or

(b) Report that there is no $n$-vector $\vec{x}$ such that $\vec{x} \geq \vec{0}$ and $A\vec{x} = \vec{b}$, or

(c) Report that the set $\{\vec{c}^T\vec{x} : A\vec{x} = \vec{b}, \vec{x} \geq \vec{0}\}$ has no lower bound.

**Question:** (Linear Inequalities) Given an integer $m \times n$ matrix $A$ and $m$-vector $\vec{b}$, is there an $n$-vector $\vec{x}$ such that $A\vec{x} \leq \vec{b}$?

**Question:** (Linear Strict Inequalities/LSI) Given an integer $m \times n$ matrix $A$ and $m$-vector $\vec{b}$, is there an $n$-vector $\vec{x}$ such that $A\vec{x} < \vec{b}$?

#### Polynomial reductions

**Theorem 9.3.2.** *There is a polynomial-time algorithm for an LP if and only if there is a polynomial-time algorithm for LI.*

**Theorem 9.3.3.** *If there is a polynomial-time algorithm for LSI then there is a polynomial-time algorithm for LI.*

Therefore, we just need to provide an polynomial-time algorithm for LSI to ensure there is a polynomial-time algorithm for LP, via the reduction LP $\leq$ LI $\leq$ LSI.

### 9.3.4 Ellipsoid algorithm

The main idea of the ellipsoid algorithm is to do a binary search. Initially the algorithm finds an ellipsoid large enough so that a part of the feasible region with lower-bounded volume $v_0$ is contained within it. In each iteration, the algorithm tests whether the center of the ellipsoid is a feasible solution. If so, it reports success. Otherwise, it finds a constraint that is violated, which defines a hyperplane. Since the center is on the wrong side of this hyperplane, at least half of the ellipsoid can be discarded. The algorithm proceeds by finding a smaller ellipsoid which contains the half of the original ellipsoid which is on the correct side. The ratio between the volume of the new ellipsoid and that of the old one is upper bounded by $r(n) < 1$. Therefore, after (polynomially) many iterations, either a solution is found, or the ellipsoid becomes small enough (the volume of it is less than $v_0$) such that no feasible region can be contained in it. In the latter case, the feasible region is can be declared empty.

The whole algorithm is given below:

**Algorithm 8:** Ellipsoid algorithm

**input** : An $m \times n$ system of linear strict inequalities $A\vec{x} < \vec{b}$, of size $L$
**output**: an $n$-vector such that $A\vec{x} < \vec{b}$, if such a vector exists; 'no' otherwise
1. Initialize
Set $j = 0, \vec{t_0} = \vec{0}, B_0 = n^2 2^{2L} \cdot I$.
/* $j$ counts the number of iterations so far. */
/* The current ellipsoid is $E_j = \{\vec{x} : (\vec{x} - \vec{t_j})^T B_j^{-1} (\vec{x} - \vec{t_j}) \leq 1\}$ */
2. Test
**if** *$\vec{t_j}$ is a solution to $A\vec{x} < \vec{b}$* **then**
  **return** $\vec{t_j}$
**end**
**if** $j > K = 16n(n+1)L$ **then**
  **return**'no'
**end**
3. Iterate:
Choose any inequality in $A\vec{x} < \vec{b}$ that is violated by $\vec{t_j}$; say $\vec{a}^T \vec{t_j} \geq \vec{b}$.
Set

$$\vec{t_{j+1}} = \vec{t_j} - \frac{1}{n+1} \frac{B_j a}{\sqrt{a^t B_j a}}$$

$$B_{j+1} = \frac{n^2}{n^2-1}(B_j - \frac{2}{n+1} \frac{(B_j a)(B_j a)^T}{a^T B_j a})$$

$$j = j + 1$$

Goto 2.

**Theorem 9.3.4.** *Ellipsoid algorithm runs in polynomial time.*

*Proof.* The algorithm runs for at most $16n(n+1)L$ rounds, and each round takes polynomial time. So the algorithm runs in polynomial time. □

# Chapter 10

# Max-Flow Min-Cut Through Linear Programming

*In this chapter, we give a proof of the Max-Flow Min-Cut Theorem using Linear Programming.*

## 10.1 Flow and Cut

First, we review the definition of Max-Flow and Min-Cut and introduce another definition of Max-Flow.

### 10.1.1 Flow

Let $N = (G, s, t, c)$ be a network (directed graph) with $s$ and $t$ the source and the sink of $N$, respectively. The capacity of an edge is a mapping c: $E \to R^+$, denoted by $c_{uv}$ or $c(u, v)$; this represents the maximum amount of "flow" that can pass through an edge.

**Definition 10.1.1** (Flow)**.** A flow is a mapping f: $E \to R^+$, denoted by $f_{uv}$ or $f(u, v)$, subject to the following two constraints:

1. $f_{uv} \leq c_{uv}$ for each $(u, v) \in E$ (capacity constraint)
2. $\sum_{u\colon\, (u,v)\in E} f_{uv} = \sum_{u\colon\, (v,u)\in E} f_{vu}$ for each $v \in V \setminus \{s, t\}$ (conservation constraint).

The value of flow represents the amount of flow passing from the source to the sink and is defined by $val(f) = \sum_{v\in V} f_{sv}$, where $s$ is the source of $N$. The maximum flow problem is to maximize $val(f)$, i.e. to route as much flow as possible from $s$ to the $t$.

**Definition 10.1.2** (Path)**.** A path is a sequence of vertices such that there is a directed path from one vertex to next vertex in the sequence. An $s - t$ path is a path that with $s$ at the beginning of the sequence and vertex $t$ the last.

### 10.1.2 An alternative Definition of Flow

We currently give another definition of flow, and later prove that it is equivalent to Definition 10.1.1.

**Definition 10.1.3** (Flow (alternate))**.** In $N = (G, s, t, c)$, let $P$ be the set of vertices that constitute the $s - t$ path. A flow is a mapping f': $P \rightarrow R^+$ , denoted by $f'_p$ or $f'(p)$, subject to the following constraint:

1. $\sum_{p:\ (u,v)\in p} f'_p \leq c_{uv}$ for each $(u, v) \in E$

The value of flow is defined by $val(f') = \sum_{p\in P} f'_p$.

### 10.1.3 Cut

**Definition 10.1.4** (Cut)**.** An $s$-$t$ cut $C = (S, T)$ is a partition of $V$ such that $s \in S$ and $t \in T$.

**Definition 10.1.5** (Cut Set)**.** The cut-set of C is the set $\{(u, v) \in E | u \in S, v \in T\}$.

Note that if the edges in the cut-set of C are removed, $val(f) = 0$.

**Definition 10.1.6** (Capacity)**.** The capacity of an $s - t$ cut is defined by $c(S, T) = \sum_{(u,v)\in S\times T} c_{uv}$.

The minimum cut problem is to minimize $c(S, T)$, i.e. to determine an $S$ and $T$ such that the capacity of the $S$-$T$ cut is minimal.

## 10.2 Max-flow Min-Cut Theorem

In this section, we prove the Max-Flow and Min-Cut Theorem by using Linear Programming. Before we go into the proof, we need the following two lemmas.

**Lemma 10.2.1.** *Given $N = (G, s, t, c)$, for every flow $val(f) > 0$, there exist an $s - t$ path $p$ such that $\forall (u, v) \in p, f_{uv} > 0$*

*Proof.* We prove the lemma by contradiction.
If such an $s - t$ path does not exist, let $A = \{v \in V | \exists$ s-v path $p, \forall (u, w) \in p, f_{uw} > 0\}$. Then $s \in A, t \notin A$ and $\forall u \in A, v \notin A, f_{uv} = 0$.

$$\begin{aligned}
\sum_{v\in A}\Big(\sum_{w:\ (v,w)\in E} f_{vw} - \sum_{u:\ (u,v)\in E} f_{uv}\Big) &= \sum_{w:\ (s,w)\in E} f_{sw} \\
&= val(f) \\
&> 0
\end{aligned}$$

But

$$\begin{aligned}
\sum_{v\in A}\Big(\sum_{w:\ (v,w)\in E} f_{vw} - \sum_{u:\ (u,v)\in E} f_{uv}\Big) &= \sum_{v\in A, w\notin A, (v,w)\in E} f_{vw} - \sum_{u\notin A, v\in A, (u,v)\in E} f_{uv} \\
&= -\sum_{u\notin A, v\in A, (u,v)\in E} f_{uv} \\
&\leq 0
\end{aligned}$$

This is a contradiction. □

**Lemma 10.2.2** (Flow Decomposition)**.** *The two definitions of flow 10.1.1 and 10.1.3 are equivalent. In $N = (G, s, t, c)$, for every $\alpha \in R^+$,*

$$\exists f : E \to R^+, val(f) = \alpha \Leftrightarrow \exists f' : P \to R^+, val(f') = \alpha$$

*Proof.* ($\Leftarrow$) Let $f_{uv} = \sum_{p:\ (u,v)\in p} f'_p$.

- First we prove that $f$ is a flow by examining the two constraints given in Definition 10.1.1.
  For the capacity constraint, we know by Definition 10.1.3 of $f'$, $f_{uv} \leq c_{uv}$ for each $(u, v) \in E$.
  For the conservation constraint, by the definition of path 10.1.2, for every path $p$, $\forall (u, v) \in p, \exists (v, w) \in p$ and $\forall (v, w) \in p, \exists (u, v) \in p$.
  Thus, for every path $p$ and node $v$ , $|\{(u, v) \in p|u \in V\}| = |\{(v, w) \in p|w \in V\}|$. So for every $v \in V$

$$\begin{aligned} \sum_{p\in P} \sum_{u:\ (u,v)\in p} f'_p &= \sum_{p\in P} \sum_{w:\ (v,w)\in p} f'_p \\ \implies \sum_{u:\ (u,v)\in E} f_{uv} &= \sum_{u:\ (v,u)\in E} f_{vu} \end{aligned}$$

- Second we prove that $val(f) = val(f')$

$$val(f) = \sum_{v\in V} f_{sv} = \sum_{v\in V} \sum_{(s,v)\in p} f'_p = \sum_{p\in P} f'_p = val(f')$$

($\Rightarrow$) We prove this direction by induction on $|E|$,

- The base case is obvious when $|E| \leq 2$,

- For the inductive step, if $val(f) = 0$, let $f'_p = 0$.
  If $|f| > 0$, then by Lemma 10.2.1 there exists an $s - t$ path $p$. Let $\alpha = \min_{f_{uv},(u,v)\in p}, \alpha > 0$.
  We construct a new flow $f_1$ by reducing $f_{uv}$ by $\alpha$ for every $(u, v) \in p$ and then remove all edges $(u, v)$ such that $f(u, v) = 0$. In the new flow, at least one edge is removed; thus, by induction, $\exists val(f'_1) = val(f_1)$. We can then get $f'$ by adding $p$ to $f'_1$.
  Thus we find

$$val(f') = val(f'_1) + \alpha = val(f_1) + \alpha = val(f)$$

□

**Theorem 10.2.3** (Max-flow Min-cut Theorem)**.** *Given* $N = (G, s, t, c)$*,* $\max\{val(f) : E \to R^+\} = \min\{val(f') : P \to R^+\}$

*Proof Sketch:*

1. Get the LP of the max-flow problem according to the Definition 10.1.3.
2. Find the dual of the LP. The equivalence of the two answers are guaranteed by the strong duality theorem.

3. Prove the solution of the dual is equal to the answer of the min-cut problem.

□

*Proof.* First, we present an LP of the max-flow problem along with its dual.

| Max-flow | Dual |
|---|---|
| maximize: $val(f') = \sum_{p \in P} f'_p$<br>subject to:<br>$\sum_{(u,v) \in p} f'_p \leq c_{uv} \quad (u,v) \in E$<br>$f'_p \geq 0 \quad p \in P$ | minimize: $\sum_{(u,v) \in E} c_{uv} y_{uv}$<br>subject to:<br>$\sum_{(u,v) \in p} y_{uv} \geq 1 \quad p \in P$<br>$y_{uv} \geq 0 \quad (u,v) \in E$ |

We prove the solution of the dual LP is equivalent to that of the min-cut of $N$ via the following two observations.

**Observation 10.2.4.** *In $N$, $\forall$ cuts $C = (S, T)$, there is a feasible solution to the dual LP of max-follow in $N$ with cost equal to $c(S, T)$.*

*Proof.* For $C = (S, T)$, set

$$y_{uv} = \begin{cases} 1 & u \in S, v \in T \\ 0 & \text{otherwise} \end{cases}$$

It is easy to check that this is a feasible solution of the dual LP and the cost is equal to $c(S, T)$. □

**Observation 10.2.5.** *In $N = (V, E)$, given a feasible solution $\{y_{uv}\}_{(u,v) \in E}$ to the dual LP, $\exists C = (S, T)$, where $c(S, T)$ is less than or equal to the cost of the solution.*

*Proof.* First, construct a graph $G = (V, E)$, where the distance between $u$ and $v$ is given by $dis(u, v) = y_{uv}, (u, v) \in E$.
Let $d(u, v)$ be the length of shortest path from $u$ to $v$ according to $dis$.
Thus the first $|P|$ constraints of the dual LP can be written as

$$d(s, t) \geq 1$$

Clearly, this stipulates that all paths from $s$ to $t$ should be greater than 1.
Pick a uniform random number $T \in [0, 1)$.
For a given $T$, we construct a cut in the following way,

$$v \in \begin{cases} S & d(s, v) \leq T \\ T & d(s, v) > T \end{cases}$$

We define the following function to make things more clear.

$$\mathsf{In}(u, v, S, T) = \begin{cases} 1 & u \in S, v \in T \\ 0 & \text{otherwise} \end{cases}$$

We prove that the expectation of $c(S,T)$ is equal to the cost of the solution of the dual LP.

$$
\begin{aligned}
\mathbb{E}(c(S,T)) &= \sum_{(u,v)\in E} \mathbb{E}(c_{uv}\mathsf{In}(u,v,S,T)) && (10.1)\\
&= \sum_{(u,v)\in E} c_{uv}\mathbb{E}(\mathsf{In}(u,v,S,T)) && (10.2)\\
&= \sum_{(u,v)\in E} c_{uv}\Pr(u\in S, v\in T) && (10.3)\\
&= \sum_{(u,v)\in E} c_{uv}\Pr(d(s,u)\le T \wedge d(s,v) > T) && (10.4)\\
&\le \sum_{(u,v)\in E} c_{uv}(d(s,v)-d(s,u)) && (10.5)\\
&\le \sum_{(u,v)\in E} c_{uv}\cdot dis(u,v) && (10.6)\\
&= \sum_{(u,v)\in E)} c_{uv}y_{uv} && (10.7)
\end{aligned}
$$

Because the expectation of $c(S,T)$ equals to the cost of the solution, then there must exist a cut $C=(S,T)$ which is less than or equal to the cost of the solution. □

According to Observation 10.2.4, we know that the minimum cost of the solution of the dual LP is less or equal to that of min-cut the problem.
According to Observation 10.2.5, we know that the solution to the min-cut problem is less than or equal to the minimum cost of the solution of the dual LP.
Thus, we arrive at the conclusion that the solution of the dual must exactly equal to that of the min-cut problem.

The proof of Max-flow Min-cut Theorem immediately follows. □

# Chapter 11

# Rounding Technique for Approximation Algorithms(I)

*In this chapter, we give some concrete examples about using Linear Programming to solve optimization problems. First, we give the general method to solve problems using LPs. Then, we give the Set Cover problem as an example to the general method. We give two approximation algorithms for Set Cover and apply LP-rounding on the Vertex Cover problem to get a 2-approximation algorithm.*

## 11.1 General Method to Use Linear Programming

Here, we show the general idea to solve problems using an LP.

To make things easier to understand, we will provide three examples of problems solved by Linear Programing: Set Cover, Vertex Cover and MAX-K SAT.

## 11.2 Set Cover Problem

### 11.2.1 Problem Description

We are given a set $\mathrm{U}$ of $n$ elements $e_1, e_2, \cdots, e_n$, and a set $\mathrm{S}$ of $k$ subsets $S_1, S_2, \cdots, S_k$ from $\mathrm{U}$. A *set cover* is a collection of subsets from $\mathrm{S}$ that satisfy the condition that every element in $\mathrm{U}$ belongs to one of the subsets. In addition, we are given a cost function $c : \mathrm{S} \to \mathbb{Z}^+$. The cost of a set cover is denoted by the sum of costs of each subset in the collection of selected subsets $\mathrm{S}$. Our objective is to find the collection of subsets that minimizes the cost.

**Example 11.2.1.** $\mathrm{U} = \{1, 2, \cdots, 6\}$ and $\mathrm{S}' = \{S_1, S_2, S_3\}$ where $S_1 = \{1, 2\}$, $S_2 = \{2, 3, 4\}$ and $S_3 = \{5, 6\}$. In addition, we have the cost of the three sets are $1$, $2$ and $3$ respectively. The example is illustrated below:

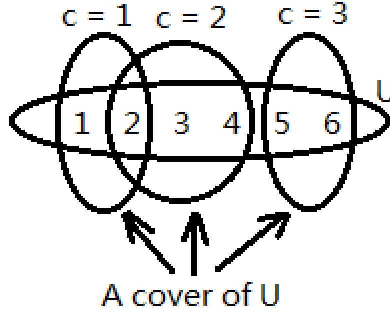



Figure 11.1: Set Cover Example

By definition, $\mathrm{S}'_1 = \{S_2, S_3\}$ is not a set cover of $(\mathrm{U}, \mathrm{S}, c)$ because $\mathrm{S}'_1$ does not cover 1. Conversely, $\mathrm{S}'_2 = \{S_1, S_2, S_3\}$ is the only set cover in this example and $c(\mathrm{S}'_2) = c(S_1)+c(S_2)+c(S_3) = 6$.

A Set Cover problem $(\mathrm{U}, \mathrm{S}, c)$ is denoted by
*Input*

$$\begin{aligned} \mathrm{U} &= \{e_1, e_2, \cdots, e_n\} \\ \mathrm{S} &= \{S_1, S_2, \cdots, S_k\} \qquad S_i \subseteq \mathrm{U} \\ c &: \mathrm{S} \to \mathbb{Z}^+ \end{aligned}$$

*Output*

$$\min_{\mathrm{S}'} c(\mathrm{S}') = \min_{\mathrm{S}'} \sum_{S \in \mathrm{S}'} c(S) \qquad \text{subject to} \quad \forall i, \exists j \text{ s.t. } S_j \in \mathrm{S}' \text{ and } e_i \in S_j$$

### 11.2.2 Complexity

**Theorem 11.2.2.** *The Set Cover Problem is NP-hard.*

*Proof.* Our proof is by reduction from Vertex Cover. For any given instance of Vertex Cover in graph $G = (\mathrm{V}, \mathrm{E})$ and an integer $j$, the Vertex Cover problem determines whether or not there exist $j$ vertices that cover all edges.

We can construct a corresponding Set Cover Problem $(\mathrm{U}, \mathrm{S}, c)$ as follows. We let $\mathrm{U} = E$; for each vertex in $v \in \mathrm{V}$, we form a set $S \in \mathrm{S}$ that contains all edges in E directly linked to the vertex $v$. Further, for every $S \in \mathrm{S}$, let $c(S) = 1$. This construction can be done in time that is polynomial in the size of the Vertex Cover instance.

Suppose that $G$ has a vertex cover of size at most $j$; then we have a set cover in the constructed problem by simply choosing the subsets corresponding to the selected vertices. On the other hand, suppose that we have a set cover in the constructed problem; we can therefore choose those vertices corresponding to the selected subsets. In this way, we have Vertex Cover $\leq_P$ Set Cover. It follows that the Set Cover Problem is NP-hard. □

### 11.2.3 Greedy Set Cover

**Algorithm Description**

---

**Algorithm 9:** Greedy Set Cover Algorithm

---

**Input**: element set $\mathrm{U} = \{e_1, e_2, \cdots, e_n\}$, subset set $\mathrm{S} = \{S_1, S_2, \cdots, S_k\}$ and cost function $c : \mathrm{S} \to \mathbb{Z}^+$

**Output**: set cover C with minimum cost

$\mathrm{C} \leftarrow \phi$;

**while** $\mathrm{C} \neq \mathrm{U}$ **do**

  Calculate cost effectiveness (defined below): $\alpha_1, \alpha_2, \cdots, \alpha_l$ of the unpicked sets $S_1, S_2, \cdots, S_l$ respectively;

  pick a set $S_j$ with minimum cost effectiveness $\alpha$;

  $\mathrm{C} \leftarrow \mathrm{C} \cup S_j$;

output C;

---

The cost effectiveness $\alpha$ of subset $S$ above is denoted by

$$\alpha = \frac{c(S)}{|S - \mathrm{C}|}$$

Let $\mathrm{C}_i$ be the set of those sets chosen up to and including that in iteration $i$. $|S - \mathrm{C}_i|$ means the size of the subtraction from set $S$ of the all union of all subsets in $\mathrm{C}_i$. It is obvious that for any particular subset $S_j$, its cost effectiveness $\alpha_j$ varies from one iteration to another.

**Example 11.2.3.** $\mathrm{U} = \{1, 2, \cdots, 6\}$ and $\mathrm{S}' = \{S_1, S_2, \cdots, S_{10}\}$. The elements and cost of each set are given below

$$\begin{array}{rclrcl}
S_1 & = & \{1,2\} & c(S_1) & = & 3 \\
S_2 & = & \{3,4\} & c(S_2) & = & 3 \\
S_3 & = & \{5,6\} & c(S_3) & = & 3 \\
S_4 & = & \{1,2,3,4,5,6\} & c(S_4) & = & 14 \\
S_5 & = & \{1\} & c(S_5) & = & 1 \\
S_6 & = & \{2\} & c(S_5) & = & 1 \\
S_7 & = & \{3\} & c(S_5) & = & 1 \\
S_8 & = & \{4\} & c(S_5) & = & 1 \\
S_9 & = & \{5\} & c(S_5) & = & 1 \\
S_{10} & = & \{6\} & c(S_5) & = & 1
\end{array}$$

The example is depicted in graphical form in Figure 11.2.3.

There are several feasible set covers in this example, e.g. $\{S_1, S_2, S_3\}$ and $\{S_2, S_5, S_6, S_9, S_{10}\}$. However, the set cover with the minimum cost is $\{S_5, S_6, S_7, S_8, S_9, S_{10}\}$, with a corresponding cost of 6.

We give a simple example of how to compute cost effectiveness. Let $\mathrm{C} = \{S_1, S_2, S_9\}$. Then, for subset $S_3$ we have

$$\alpha = \frac{c(S_3)}{|S_3 - \mathrm{C}|} = \frac{3}{1} = 3$$

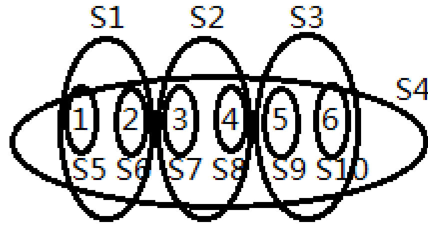



Figure 11.2: Greedy Set Cover Example

**Approximation Rate of the Greedy Set Cover**

We sort the elements $e_1, e_2, \cdots, e_n$ by the iteration when they were added to C. Let this sorted list be $e'_1, e'_2, \cdots, e'_n$.

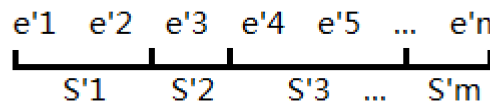



Figure 11.3: sort elements by iteration time

Notice that the above subsets $S'_j$ are different from the original $S'_j$, namely, the set only includes those elements added by itself to C.

For each $e'_j$ which is added to $S'_j$, we define

$$price(e'_j) = \alpha_{S'_j}$$

namely the cost effectiveness of the set where $e'_j$ was covered.

**Observation 11.2.4.** $A(I) = \sum_{k=1}^{n} price(e'_k)$

*We denote the cost of the set cover of Greedy Set Cover by $A(I)$, where $I$ is the input. Therefore, we have*

$$\begin{aligned} A(I) &= c(\mathrm{C}) \\ &= \sum_{S'_i \in \mathrm{C}} c(S'_i) \\ &= \sum_{S'_i \in \mathrm{C}} \sum_{e' \in S'_i} \frac{c(S'_i)}{|S'_i|} \\ &= \sum_{k=1}^{n} price(e'_k) \end{aligned}$$

*The third equation above breaks the cost of a subset into the costs of those elements in the subset.*

**Lemma 11.2.5.** *Let $k = 1, 2, \cdots, n$, we have $price(e'_k) \leq OPT/(n-k+1)$.*

*Proof.* Note that the remaining elements $e'_k, e'_{k+1}, \cdots, e'_n$ can be covered at cost at most OPT, namely in the OPT solution of the problem. Therefore, there is a subset in OPT which has not been selected and with a cost effectiveness at most $OPT/(n-k+1)$. Further, our algorithm chooses to cover a set with minimum cost effectiveness, so that we should have $price(e'_k) \leq OPT/(n-k+1)$ to guarantee our algorithm could pick the selected subset. □

**Theorem 11.2.6.** *Greedy Set Cover is* $\ln(n)$*-approx*

*Proof.*

$$\begin{aligned} A(I) &= \sum_{k=1}^{n} price(e'_k) \\ &\leq (\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2} + 1)OPT \\ &\leq \lceil \ln(n) \rceil OPT \end{aligned}$$

The first equation is due to Observation 11.2.4, and the second is derived from the above lemma. Thus, the Greedy Set Cover is an $\ln(n)$-approximation algorithm. □

### Tightness of Greedy Set Cover

U $= \{e_1, e_2, \cdots, e_n\}$ and S $= \{S_1, S_2, \cdots, S_n\}$. The elements and cost of each set is given below:

$$\begin{aligned} S_1 &= \{1\} & c(S_1) &= 1/n \\ S_2 &= \{2\} & c(S_2) &= 1/(n-1) \\ S_3 &= \{3\} & c(S_3) &= 1/(n-2) \\ &\vdots & &\vdots \\ S_{n-1} &= \{n-1\} & c(S_{n-1}) &= 1/2 \\ S_n &= \{1, 2, \cdots, n\} & c(S_n) &= 1+\epsilon \end{aligned}$$

The $\epsilon$ above is some negligible value. It is obvious that the OPT Set Cover is $\{S_n\}$ with cost $n$. However, in our algorithm, in iteration $i$ we will pick the $S_j$ that has the minimum cost effectiveness. The $\epsilon$ is used here to make sure the cost effectiveness of $S_j$ is smaller that of $S_n$. In this way, we will choose all $n$ subsets and get a solution with cost

$$\frac{1}{n} + \frac{1}{n-1} + \cdots + 1 = \lceil \ln(n) \rceil$$

So we construct a worst case for any $n$ with approximation ratio $\ln(n)$. In conclusion, the $\ln(n)$ approximation is tight for Greedy Set Cover.

### 11.2.4 LP-rounding Set Cover

**Integer Program**

We express the Set Cover problem as an Integer Programs in the following form:
**Minimize**

$$\sum_{i=1}^{k} c(S_i)u_i$$

**Subject to**

$$\begin{array}{rl} \sum_{i:e\in S_i} u_i \geq 1 & , \quad \forall e \in \mathrm{U} \\ u_i \in \{0,1\} & , \quad i = 1,2,\cdots,k \end{array}$$

where $u_i$ is the boolean variable that shows whether or not set $S_i$ has been chosen. $S_i$ is not chosen when $u_i = 0$, while $S_i$ is chosen when $u_i = 1$.

**Example 11.2.7.** $\mathrm{U} = \{1,2,3,4\}$ and the sets and cost of each are given below:

$$\begin{array}{lllll} S_1 & = & \{1,2\} & c(S_1) & = & 5 \\ S_2 & = & \{2,3\} & c(S_2) & = & 100 \\ S_3 & = & \{2,3,4\} & c(S_3) & = & 1 \end{array}$$

We can change the set cover problem above into the integer programs as follows:
**Minimize**

$$5u_1 + 100u_2 + u_3$$

**Subject to**

$$\begin{array}{rcrcl} 1 & : & u_1 & \geq & 1 \\ 2 & : & u_1 + u_2 + u_3 & \geq & 1 \\ 3 & : & u_2 + u_3 & \geq & 1 \\ 4 & : & u_3 & \geq & 1 \end{array}$$

$$u_i \in \{0,1\} \qquad i \in \{1,2,3\}$$

**Relaxing to a Linear Program**

The above construction an Integer Program properly represents the original Set Cover problem; however, the Integer Program is still NP-hard.

To get an approximation algorithm, we relax the integral constraints and get a Linear Program.

| Integer Program | Linear Program |
|---|---|
| **Minimize :** $\sum_{i=1}^{k} u_i c(S_i)$<br>**Subject to:**<br>$\sum_{i:e\in S_i} u_i \geq 1 \quad , \quad \forall e \in \mathrm{U}$<br>$u_i \in \{0,1\} \quad , \quad i = 1,2,\cdots,k$ | **Minimize :** $\sum_{i=1}^{k} u_i c(S_i)$<br>**Subject to:**<br>$\sum_{i:e\in S_i} u_i \geq 1 \quad , \quad \forall e \in \mathrm{U}$<br>$0 \leq u_i \leq 1 \quad , \quad i = 1,2,\cdots,k$ |

We relax only the non-linear constraints for $u_i$. In this way, we can easily solve the Linear Program in polynomial time. However, the relaxation of constraints brings about a new problem: How can we interpret the result vector $u = (u_1, u_2, \cdots, u_k)$ when some $u_i$'s are fractions?

**Algorithm 10:** LP-rounding Set Cover Algorithm

**Input**: element set $\mathrm{U} = \{e_1, e_2, \cdots, e_n\}$, subset set $\mathrm{S} = \{S_1, S_2, \cdots, S_k\}$ and cost function $c : \mathrm{S} \to \mathbb{Z}^+$

**Output**: set cover $\mathrm{C}$ with minimum cost

1. Write the original Set Cover problem as an Integer Program.
2. Relax the Integer Program into a Linear Program.
3. Solve the Linear Program using the Ellipsoid Algorithm and obtain $\vec{u} = (u_1, u_2, \cdots, u_k) \in \mathbb{R}^k$.
4. Let $f$ be the maximum frequency (the number of times that element appears in distinct subsets).
5. Output deterministic rounding $\vec{u'} = (u'_1, u'_2, \cdots, u'_k) \in \{0, 1\}^k$, where $u'_i$ satisfies $u'_i = \begin{cases} 1 & , \quad u_i \geq 1/f \\ 0 & , \quad \text{otherwise} \end{cases}$

---

**LP-rounding Set Cover Algorithm**

The algorithm first changes the original Set Cover problem into an Integer Program, then relaxes the constraints of the IP to obtain a Linear Program. By solving this LP, we get a solution $\vec{u} \in \mathbb{R}^k$. Lastly, we round each element of the result vector $\vec{u}$ using the $f$ below.

**Effectiveness of LP-rounding**

**Lemma 11.2.8.** *LP-rounding Set Cover is an $f$-approximation.*

*Proof.* First, we prove that $\vec{u'}$ is a feasible solution. In order to show this, we need to examine each constraint and satisfy it. Suppose

$$u_1 + u_2 + \cdots + u_l \geq 1$$

be one of the constraints. By definition, we have $l \leq f$. Now, there must be some $i$ such that $u_i \geq 1/l \geq 1/f$. In this way, $u_i$ will be rounded to 1, and therefore satisfies the proper constraint. Thus, the resulting vector $\vec{u'}$ is a feasible solution.

On the other hand, we look into the rounding process from $u_i$ to $u'_i$:

$$u'_i = \begin{cases} 1 & , \quad u_i \geq 1/f \\ 0 & , \quad \text{otherwise} \end{cases}$$

We can easily check that $u'_i \leq f \cdot u_i$ for both cases. So that

$$\begin{aligned}\sum_{i=1}^{k} c(S_i) \cdot u'_i &\leq \sum_{i=1}^{k} c(S_i) \cdot f \cdot u_i \\ &= f \sum_{i=1}^{k} c(S_i) \cdot u_i \\ &\leq f \cdot OPT\end{aligned}$$

The first inequality is due to Lemma 11.2.8. The second inequality comes from relaxing the constraints to enlarge the feasible solution area. Finally, we know that the OPT from the Integer Program will be included in the solution space of the Linear Program.

$$IP \xrightarrow{relax} LP \qquad \longrightarrow \qquad OPT_{IP} \geq OPT_{LP}$$

This shows that the LP-rounding of Set Cover produces feasible solution for the problem, and gives approximation ratio of $f$. □

## 11.3 Vertex Cover

As we proved in Section 11.2.2, any Vertex Cover problem can be changed into Set Cover problem.

**Example 11.3.1.** $G = (V, E)$, where $V = \{u_1, u_2, u_3, u_4, u_5\}$ and $E = \{e_1, e_2, \cdots, e_8\}$. We can

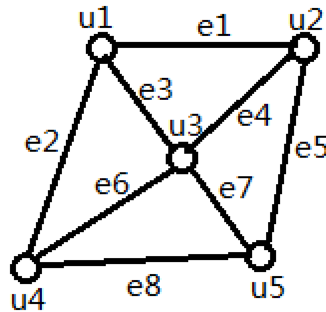



Figure 11.4: Vertex Cover Example

change the above Vertex Cover problem into a Set Cover problem, where

$$\begin{aligned}\text{U} &= \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\} \\ \text{S} &= \{(e_1, e_2, e_3), (e_1, e_4, e_5), (e_3, e_4, e_6, e_7), (e_2, e_6, e_8), (e_5, e_7, e_8)\} \\ c(S_i) &= 1 \qquad , \qquad \forall S_i \in \text{S}\end{aligned}$$

From the above construction, i.e. using LP-rounding for Set Cover, we may by extension solve the Vertex Cover problem, the maximum frequency corresponding to the number of vertices that belong to a particular edge, which is 2. We have, therefore, $f = 2$ and the fact that by using LP-rounding for Set Cover, we can get a 2-approximation algorithm for Vertex Cover.

# Chapter 12

# Rounding Technique for Approximation Algorithms(II)

*In this chapter, we will give a randomized algorithm for Set Cover. The algorithm gives an approximation factor of* $\log n$ *for Set Cover, and reaches this approximation ratio with high probability. Then, we give a randomized algorithm for the MAX-K SAT problem. Furthermore, we will give a deterministic algorithm which derandomizes the aforementioned MAX-K SAT algorithm using the method of conditional expectation.*

## 12.1 Randomized Algorithm for Integer Program of Set Cover

### 12.1.1 Algorithm Description

We know from the previous lecture how to express the Set Cover problem as a Linear Program. For the randomized algorithm, we do the same thing as the last lecture, perform a linear relaxation.

**Minimize :**

$$\sum_{i=1}^{k} u_i c(S_i)$$

**Subject to:**

$$\begin{array}{rcl} \sum_{i:e\in S_i} u_i \geq 1 & , & \forall e \in \mathrm{U} \\ u_i \in [0,1] & , & i = 1, 2, \cdots, k \end{array}$$

Here is the algorithm:

### 12.1.2 The correctness of the Algorithm

Firstly, we want to know the expectation of $C(G)$, where $C(G)$ is the configuration of a random collection of sets in all iterations. Assume $\overline{G}$ is the random collection of sets in one iteration. Then

**Algorithm 11:** LP-rounding Set Cover Algorithm

**Input**: element set $\mathrm{U} = \{e_1, e_2, \cdots, e_n\}$, subset set $\mathrm{S} = \{S_1, S_2, \cdots, S_k\}$ and cost function $c : \mathrm{S} \rightarrow \mathbb{Z}^+$

**Output**: set cover $\mathrm{C}$ with minimum cost

1. Solve the LP-rounding Set Cover, and let the solution be $(x_1, x_2 \ldots x_n) \in \mathbb{R}^n$(e.g. using the Ellipsoid algorithm)

2. Randomly choose $u'_i \in_R [0, 1]$

   If $u'_i \leq x_i$, $u_i \leftarrow 1$, otherwise $u_i \leftarrow 0$

   Then $Pr(u_i = 1) = x_i$ and $Pr(u_i = 0) = 1 - x_i$

3. Repeat the second step $2 \ln n$ times. Output the collection $G$ of sets corresponding to $u_i$ (If $u_i = 1$, then we picked $S_i$ into G)

---

the expectation of $C(\overline{G})$ is

$$\begin{aligned} E[C(\overline{G})] &= \sum_{i=1}^{k} Pr(S_i \text{ is picked}) C(S_i) \\ &= \sum_{i=1}^{k} x_i C(S_i) \\ &= OPT_{\text{fractional}} \\ &\leq OPT \end{aligned}$$

So the expectation of $C(G)$ is

$$\begin{aligned} E[C(G)] &\leq 2 \ln n \times E[C(\overline{G})] \\ &\leq (2 \ln n) OPT_{\text{fraction}} \\ &\leq 2 \ln n OPT \end{aligned}$$

Markov's inequality:

$$Pr(|X| \geq \alpha) \leq \frac{E(|X|)}{a}$$

Using Markov's inequality, we have

$$Pr[C(G) \geq 8 \ln n OPT] \leq \frac{1}{4}$$

Now, we want to know the probability that every $e_i$ has been covered. We know one element is

not covered during *one* iteration of step 2 in algorithm with probability

$$\begin{aligned} Pr[\text{e is not covered in one iteration}] &= \prod_{i:e\in S_i} (1-x_i) \\ &\leq \prod_{i:e\in S_i} e^{-x_i} \\ &= e^{-\sum_{i:e\in S_i} x_i} \\ &\leq \frac{1}{e} \end{aligned}$$

The last inequality is due to $\sum_{i:e\in S_i} x_i \geq 1$

Then the probability that $e$ is not covered in any of the $2\ln n$ repetitions is:

$$P \leq (\frac{1}{e})^{2\ln n} \leq (\frac{1}{n})^2 = \frac{1}{n^2}$$

We know that:

$$Pr(A \vee B) \leq Pr(A) + Pr(B)$$

So the probability of having an element which is not covered is:

$$\begin{aligned} Pr[\exists e \text{is not covered}] &= Pr[e_1 \text{ is not covered} \vee e_2 \text{ is not covered} \cdots \vee e_n \text{ is not covered}] \\ &\leq \frac{1}{n^2} \cdot n \\ &= \frac{1}{n} \end{aligned}$$

So

$$Pr[\text{all elements are covered}] \geq 1 - \frac{1}{n}$$

Let $\mathcal{E}$ be the event that all elements are covered and $\mathcal{E}'$ be the event that $C(G) \geq 8\ln n \cdot OPT$. Then we have:

$$\begin{aligned} Pr[\overline{\mathcal{E}'} \wedge \mathcal{E}] &\geq 1 - \frac{1}{n} - \frac{1}{4} \\ &> \frac{2}{3} \end{aligned}$$

Hence, we have an algorithm that succeeds with probability higher than $\frac{1}{2}$. Using Chernoff bounds we can repeat the algorithm a polynomial number of times so as to succeed with high probability.

## 12.2 Method of Computation Expectations

### 12.2.1 MAX-K-SAT problem

Let us introduce the MAX-K-SAT problem.

| **Input:** |
|---|
| 1. K-CNF $c_1 \wedge c_2 \cdots \wedge c_m$ |
| 2. $C : c_1, c_2 \ldots c_m \rightarrow \mathbb{Z}^+$ |
| **Output:** |
| output a truth assignments that maximizes profit, i.e. satisfies a maximal number of clauses |

We know that MAX-2-SAT is an NP-Hard problem. But we have a derandomized algorithm to get at least one half of the optimal for the problem.

### 12.2.2 Derandomized Algorithm

The derandomized algorithm can be described as follows:

We choose every variable's value with $\frac{1}{2}$ probability of 1 and $\frac{1}{2}$ probability of 0. Using $E(X + Y) = E(X) + E(Y)$ and the method of conditional expectation, we can calculate the expectation of the total profit. More than that, if we have decided some variable's value, we can also calculate the expectation of the total profit in polynomial time for the rest random variables. By making use of these facts, we can design a derandomized algorithm that always achieves an approximation ratio of at least one half of of OPT.

Assume $W_{c_i} = C(c_i)$ if $c_i$ is satisfied , otherwise $W_{c_i} = 0$. And $W = \sum_{i=1}^{m} W_{c_i}$

For step i:

1. We calculate $E[W|_{x_i=1}]$ and $E[W|_{x_i=0}]$ in polynomial time.
2. We choose $x_i$'s value with bigger expectation for $W$

### 12.2.3 The proof of correctness

Firstly, we have

$$E[W] = \frac{1}{2}E[W|_{x_i=1}] + \frac{1}{2}E[W|_{x_i=0}]$$

So the bigger one of $E[W|_{x_i=1}]$ and $E[W|_{x_i=0}]$ must be bigger than E[W] too. At the end of the algorithm, we must have profits bigger than $E[W]$.

And we know

$$Pr[c_i \text{ is satisfied}] = 1 - \frac{1}{2^k} \geq \frac{1}{2}$$

So

$$E[W_{c_i}] \geq \frac{1}{2}C(c_i)$$

So

$$\begin{aligned} E[W] &= E[W_{c_1}] + E[W_{c_2}] + \cdots + E[W_{c_m}] \\ &\geq \frac{1}{2}(C(c_1) + C(c_2) + \cdots + C(c_m)) \\ &\geq \frac{1}{2}OPT \end{aligned}$$

So the profit (total number of clauses satisfied) we get at the end of algorithm is greater than or equal to $\frac{1}{2}OPT$.

# Chapter 13

# Primal Dual Method

*In this lecture, we introduce Primal-Dual approximation algorithms. We also construct an approximation algorithm to solve SET COVER as an example.*

## 13.1 Definition

In the previous lecture, we know that every Linear Programming problem has a corresponding Dual problem:

| PRIMAL | DUAL |
|---|---|
| **Minimize :** $\sum_{j=1}^{n} c_j x_j$<br>**Subject to:**<br>$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad , \quad i = 1, 2, \cdots, m$<br>$x_j \geq 0 \quad , \quad j = 1, 2, \cdots, n$ | **Maximize :** $\sum_{i=1}^{m} b_i y_i$<br>**Subject to:**<br>$\sum_{i=1}^{m} a_{ij} y_i \leq c_j \quad , \quad j = 1, 2, \cdots, n$<br>$y_i \geq 0 \quad , \quad i = 1, 2, \cdots, m$ |

Also from previous lectures, we know that the Primal and Dual have optimal solutions if and only if:

- **Primal:** $\forall j = 1, 2, \cdots, m$, $x_j = 0$ or $\sum_{i=1}^{m} a_{ij} y_i = c_j$
- **Dual:** $\forall i = 1, 2, \cdots, n$, $y_i = 0$ or $\sum_{j=1}^{n} a_{ij} x_j = b_i$

We give a relaxation of the above conditions as follows:

- **Primal complementary slackness conditions:**

  For $\alpha \geq 1$, $\forall j = 1, 2, ..., n$, either $x_j = 0$ or $\frac{c_j}{\alpha} \leq \sum_{i=1}^{m} a_{ij} y_i \leq \alpha c_j$

- **Dual complementary slackness conditions:**

  For $\beta \geq 1$, $\forall i = 1, 2, ..., m$, either $y_i = 0$ or $\frac{b_i}{\beta} \leq \sum_{j=1}^{n} a_{ij} x_j \leq \beta b_i$

**Theorem 13.1.1.** *If $\vec{x}, \vec{y}$ are feasible solutions to PRIMAL and DUAL respectively, and they satisfy primal conditions and dual conditions, then $\sum_{j=1}^{n} c_j x_j \leq \alpha\beta OPT$.*

*Proof.* $\sum_{j=1}^{n} c_j x_j \leq \alpha \sum_{j=1}^{n} \sum_{i=1}^{m} a_{ij} y_i x_j \leq \alpha \sum_{i=1}^{m} y_i \sum_{j=1}^{n} a_{ij} x_j \leq \alpha\beta \sum_{i=1}^{m} b_i y_i \leq \alpha\beta$ OPT. □

With the above theorem, the basic idea of primal dual method is: set $\vec{x} = \vec{0}, \vec{y} = \vec{0}$ at the beginning, then modify them to get an optimal solution.

## 13.2 Set Cover Problem

Since we have discussed a lot about the Set Cover Problem in the previous lectures, here we only give a brief description of the Set Cover Problem:

**Input:**
$U = \{e_1, e_2, ..., e_n\}$,$\mathcal{S} = \{S_1, S_2, ..., S_k\}$,$C : \mathcal{S} \to Z^+$.
**Output:**
A cover $\mathcal{S}' \subseteq \mathcal{S}$ of $U$ with minimal total cost.

We presently give a PRIMAL for the Set Cover problem as well as its DUAL:

- **PRIMAL form:** Minimize $\sum_{S\in\mathcal{S}} c(S)x_S$,subject to $\forall e \in U, \sum_{S:e\in S} x_S \geq 1$.
- **DUAL form:** Maximize $\sum_{e\in U} y_e$,subject to $\forall S \in \mathcal{S}, \sum_{e:e\in S} y_e \leq c(S), y_e \geq 0$.

We also have the following Primal and Dual conditions:

- **The Primal conditions:** Let $\alpha$=1,$x_s \neq 0, \Rightarrow \sum_{e:e\in S} y_e = c(S)$.
- **The Dual conditions:** Let $\beta = f$,$y_e \neq 0, \Rightarrow \sum_{S:e\in S} x_s \leq f$. (here f is the maximal number of times an element appears in a set.)

**Algorithm 12:** Primal Dual Algorithm for Set Cover

Let $\vec{x} \leftarrow \vec{0}, \vec{y} \leftarrow \vec{0}$.
We call a set $S$ satisfying the primal condition "tight".
**repeat**
    Pick an uncovered $e \in U$.
    Increase $y_e$ until a set $S$ becomes tight.
    Pick all tight sets in the cover and update $\vec{x}$,($x_S = 1$.)
    let the elements of these sets as covered
**until** *all elements covered*;

**Claim 13.2.1.** *The algorithm is an $f$-approximation for SET-COVER.*

*Proof.* At the end of the execution the algorithm, the algorithm has covered every element e, so the constrains of the Primal form are satisfied, i.e. $\vec{x}$ is a feasible solution for the primal.

In addition, the algorithm never violates the conditions of the Dual. Thus, $\vec{x}, \vec{y}$ are feasible for the dual.

By definition, an element $e \in U$ is contained in at most $f$ sets and $\forall S \in \mathcal{S}$, $x_s \leq 1 \Rightarrow$ the Primal conditions are always satisfied.

Finally, the Dual conditions are satisfied by the way the algorithm works.

We therefore end up with an $f$-approximation algorithm. □

Finally, we give an example which shows that the approximation ratio $f$ is also the lower bound of the algorithm.

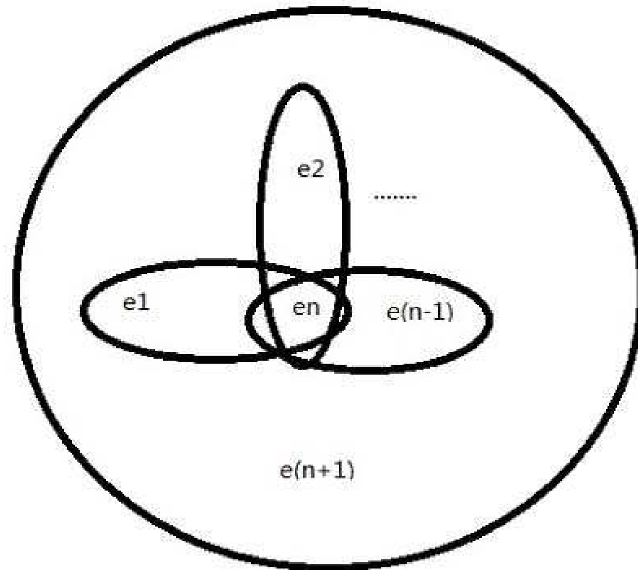



Figure 13.1: An example of a case where the solution of the Primal Dual algorithm is $f$ times the optimal

Let $c(S) = 1$ for all $S \in \mathcal{S}$. We can see that OPT is $x_{n+1} = 1$ with total cost $value = 1+\varepsilon$. The Primal Dual algorithm will choose sets $x_1 = x_2 = ... = x_{n+1}$=1, with a total cost $value = n+1$. So we have that $\frac{n+1}{1+\varepsilon} \to n+1 = f$. This example shows that $f$ is a lower bound for the approximation ratio of the Primal Dual method.