*Course Offering*
*Spring '94*

# IEOR 269: Integer Programming and Combinatorial Optimization

| | |
|---|---|
| *Instructor:* Prof. Dorit S. Hochbaum | CCN# 42384 |
| *Time:* TTh 3:30-5:00 | 433 Latimer |
| *Prerequisite:* IEOR 262A | 3 Units |

This course is concerned with discrete optimization problems, their application, complexity and methods for obtaining effective solutions to such problems. The purpose of the course is to provide students with skills for recognizing the difficulty of particular discrete problems and to identify certain algorithmic tools that could be useful in devising algorithms for such problems.

There will be two exams - a midterm and a final. Both will be *take home* exams. The midterm contributes 50% towards the final grade, and the final 45%. There is also a required assignment (biweekly). Note taking (scribe, using Latex) for up to two lectures during the semester is a requirement. The quality of the notes and the assignments contribute 5% towards the final grade.

---

**Lecturer:** Dorit Hochbaum
**Scribe:** Anu Pathria

**Handouts:** Course Outline; ILP handout

*A few administrative details:*
**Instructor:** Prof. Dorit S. Hochbaum, dorit@hochbaum.berkeley.edu
**Office:** 4181 Etcheverry Hall, 642-4998
**Office Hours:** Thursday 1:00-3:00
**TA:** Anu Pathria, pathria@cimsim.berkeley.edu
**Office Hours:** By appointment, or using e-mail

---

Consider the general form of a linear program:

$$\begin{array}{rc} \max & \sum_{i=1}^{n} c_i x_i \\ \text{subject to} & Ax \le b \end{array}$$

In an *integer* programming optimization problem, the additional restriction that the $x_i$ must be integer-valued is also present.

While at first it may seem that the integrality condition limits the number of possible solutions and could thereby make the integer problem easier than the continuous problem, the opposite is actually true. Linear programming optimization problems have the property that there exists an optimal solution at a so-called *extreme point* (a basic solution); the optimal solution in an integer program, however, is not guaranteed to satisfy any such property and the number of possible integer valued solutions to consider becomes prohibitively large, in general.

While linear programming belongs to the class of problems $P$ for which "good" algorithms exist (an algorithm is said to be good if its running time is bounded by a polynomial in the size of the input), integer programming belongs to the class of *NP-hard* problems for which it is considered highly unlikely that a "good" algorithm exists. For some integer programming problems, such as the *Assignment Problem* (which is described later in this lecture), efficient algorithms do exist. Unlike linear programming, however, for which effective general purpose solution techniques exist (ex. simplex method, ellipsoid method, Karmarkar's algorithm), integer programming problems tend to be best handled in an ad hoc manner. While there are general techniques for dealing with integer programs (ex. branch-and-bound, simulated annealing, cutting planes), it is usually better to take advantage of the structure of the specific integer programming problems you are dealing with and to develop special purposes approaches to take advantage of this particular structure.

Thus, it is important to become familiar with a wide variety of different classes of integer programming problems. Gaining such an understanding will be useful when you are later confronted with a new integer programming problem and have to determine how best to deal with it. This will be a main focus of this course.

One natural idea for solving an integer program is to first solve the "LP-relaxation" of the problem (ignore the integrality constraints), and then round the solution. As indicated in the class handout, there are several fundamental problems to using this as a general approach:

1. The rounded solutions may not be feasible.
2. The rounded solutions, even if some are feasible, may not contain the optimal solution.
3. Even if one of the rounded solutions is optimal, checking all roundings is computationally expensive. There are $2^n$ possible roundings to consider for an $n$ variable problem, which becomes prohibitive for even moderate sized $n$.

## Formulations

We next formulate a couple of problems as integer linear programs.

**Knapsack Problem:** Suppose you are going on a hike, and have a backpack with capacity $B$ (a weight capacity, say). There is a list of $n$ items that you can possibly take on your trip, with item $i$ having utility $u_i$ and weight $w_i$, for $i = 1, \dots, n$. The problem is to select a subset of the $n$ items of maximum total utility that does not violate the capacity constraint.

To formulate this problem as an integer program, we define the binary decision variable $x_i$ to indicate whether or not item $i$ is selected:

$$x_i = \begin{cases} 1, & \text{if item } i \text{ is taken} \\ 0, & \text{otherwise.} \end{cases}$$

The problem can then be formulated as:

$$\begin{array}{rll} \max & \sum_{i=1}^n u_i x_i & \\ \text{subject to} & \sum_{i=1}^n w_i x_i \le B, & \\ & 0 \le x_i \le 1 & x_i \text{ integer}, \ i = 1, \dots, n \end{array}$$

Sometimes the last set of constraints is written as $x_i \in \{0, 1\}, \ i = 1, \dots, n$.

This problem, which is sometimes referred to as the *0-1 Knapsack Problem*, is known to be "weakly" NP-hard, a concept that will be defined later in the course. There is a similar problem called the *Integer Knapsack Problem* in which the upper bound of 1 on each $x_i$ is replaced by some value $U_i$.

**Assignment Problem:** Given $n$ people and $n$ tasks, there is a utility associated with a particular person performing a particular task. The problem is to assign exactly one task per person so that the overall utility is maximized.

We let the binary decision variable $x_{ij}$ indicate whether or not task $j$ is assigned to person $i$:

$$x_{ij} = \begin{cases} 1, & \text{if task } j \text{ is assigned to person } i \\ 0, & \text{otherwise.} \end{cases}$$

$$\begin{array}{lll} \max & \sum_{j=1}^{n}\sum_{i=1}^{n} \; u_{ij}x_{ij} & \\ \text{subject to} & \sum_{i=1}^{n} x_{ij} = 1, & \forall j = 1,\ldots,n \\ & \sum_{j=1}^{n} x_{ij} = 1, & \forall i = 1,\ldots,n \\ & x_{ij} \in \{0,1\} & \end{array}$$

The first set of constraints ensures that exactly one person is assigned to each task; the second set of constraints ensures that each person is assigned exactly one task.

The linear programming relaxation of this problem (in which the $x_{ij} \in \{0,1\}$ constraint is replaced by $0 \leq x_{ij} \leq 1$) has the special property that all extreme points are integer valued; hence, the given integer program can be solved in polynomial time using linear programming algorithms. In fact, this problem, which is sometimes referred to as the *marriage problem* or the *maximum weight bipartite matching problem*, can be solved even more efficiently via network flow techniques.

There is a well-known related problem called the *maximum weight non-bipartite matching* problem. Given an even number of individuals, $n$, and a utility $u_{ij}$ associated with pairing up persons $i$ and $j$, the problem is to pair up all individuals so as to maximize overall utility.

We associate the binary indicator variable $x_{ij}$ with the decision to match up persons $i$ and $j$. Because $i$ being paired with $j$ implies that $j$ is paired with $i$, we need only consider $x_{ij}$ for $1 \leq i < j \leq n$. We can formulate the problem as an integer program ($u_{ij}$ is the utility associated with person $i$ performing task $j$):

$$\begin{array}{lll} \max & \sum_{1\leq i<j\leq n} \; u_{ij}x_{ij} & \\ \text{subject to} & \sum_{j=1}^{i-1} x_{ji} + \sum_{j=i+1}^{n} x_{ij} = 1, & \forall i = 1,\ldots,n \\ & x_{ij} \in \{0,1\} & \end{array}$$

Unlike the bipartite case, the LP-relaxation to the above formulation is not guaranteed to give integer solutions. A non-trivial algorithm with polynomial running time for this problem was first developed by J. Edmonds.

**Lecturer:** Dorit Hochbaum
**Scribe:** Anu Pathria

**Handouts:** Assignment #1
**Homework:** Read chapter 1 of text

# 1 Additional Formulations

In the last class, we formulated the *knapsack* and *assignment* problems as integer linear programs. We now consider formulating several additional problems as such. In particular, these are binary problems; that is, as is the case in many applications, the integer variables take on 0-1 values.

**Non-Linear Objective Functions:** Two problems in which the objective function is non-linear are considered.

1. *Production Problem, with set-up costs.*
   Consider a problem in which it must be determined what quantity $x_j$ ($x_j$ can be fractional) of item $j$ to produce, for $j = 1, \ldots, n$. The item quantities must satisfy a set of constraints $Ax \leq b,\ x \geq 0$.
   There is a revenue $R_j$ attained per unit of item $j$. There is a cost $c_j$ per unit of item $j$ produced; in addition, there is a set-up cost $s_j$ that is incurred if any non-zero quantity of item $j$ is produced. In other words, as depicted in the graph of Figure 1,

   $$\text{Cost to produce } x_j \text{ units of item } j = \begin{cases} s_j + c_j x_j, & \text{if } x_j > 0. \\ 0, & \text{if } x_j = 0. \end{cases}$$

   The objective is to maximize profit.
   Let $y_j$ be an indicator variable with value 1 if a non-zero quantity of item $j$ is produced, and 0 otherwise. The production can be formulated as:

   $$\begin{array}{rll} \max & \sum_{j=1}^{n}([R_j - c_j]x_j - s_j y_j) & \\ \text{subject to} & Ax \leq b & \\ & x_j \leq M y_j, & j = 1, \ldots, n \\ & x_j \geq 0, y_j \in \{0,1\}, & j = 1, \ldots, n \end{array}$$

   where $M$ is large constant (upper bound on amount that can be produced, say). The constraints involving $M$ ensure that $x_j > 0 \Rightarrow y_j = 1$; when $x_j = 0$, $y_j$ will be 0 in an optimal solution, because of the form of the objective function.

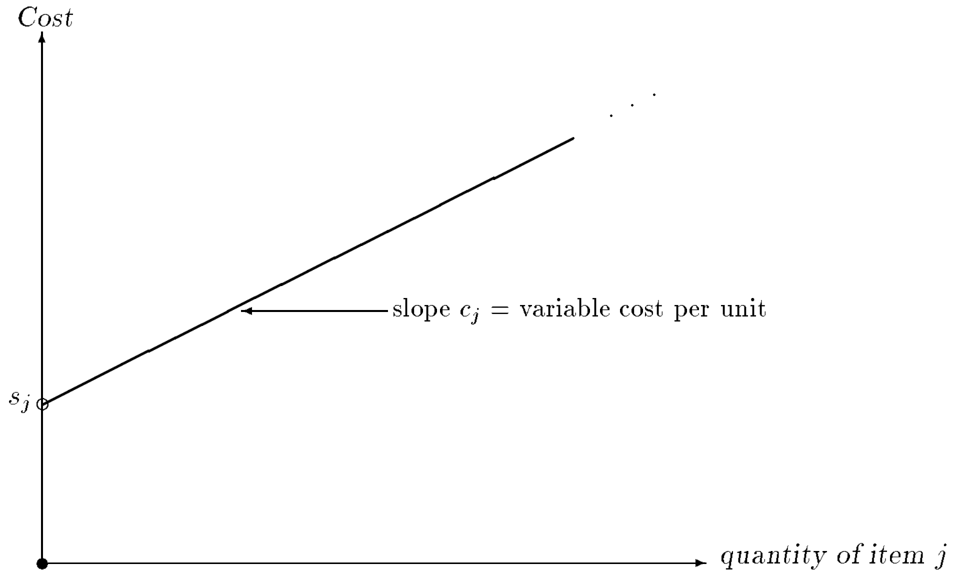



Figure 1: Cost of producing item $j$.

This is sometimes called a *mixed-integer program*, because there are both continuous and integer variables. In this course we will tend to simply refer to this as an integer program.

2. *Piece-wise linear objective function.*
Consider the problem of minimizing

$$\left\{ \sum_i f_i(x_i) \mid Ax \geq b, x \geq 0 \right\} ,$$

where each $f_i$ is a continuous piece-wise linear function. This problem can be modeled by a linear integer program as follows:
Consider $f_i$ for a given $i$, and assume $f_i$ is made up of $k_i$ linear pieces. Let $F_i^0 = f_i(0)$; let the length and slope of the $j$th linear piece be $\delta_i^j$ and $c_i^j$, respectively, for $j = 1, \ldots, k_i$. This is depicted in Figure 2.
For $j = 1, \ldots, k_i$, let $x_i^j$ be a variable that corresponds to the length of $x_i$ in interval $j$, and let

$$y_i^j = \begin{cases} 1, & \text{if } x_i \text{ is in interval } j \\ 0, & \text{otherwise.} \end{cases}$$

Then, the problem can be formulated as follows, with the additional constraints ensuring that $x_i$ covers the continuous interval $[0, \sum_j^{k_i} x_i^j]$.

$$\begin{array}{rll} \min & \sum_i [F_i^0 + \sum_j^{k_i} c_i^j x_i^j] & \\ \text{subject to} & Ax \leq b & \\ & \delta_i^j y_i^{j+1} \leq x_i^j \leq \delta_i^j y_i^j, \quad \forall\ i, j & (1) \end{array}$$

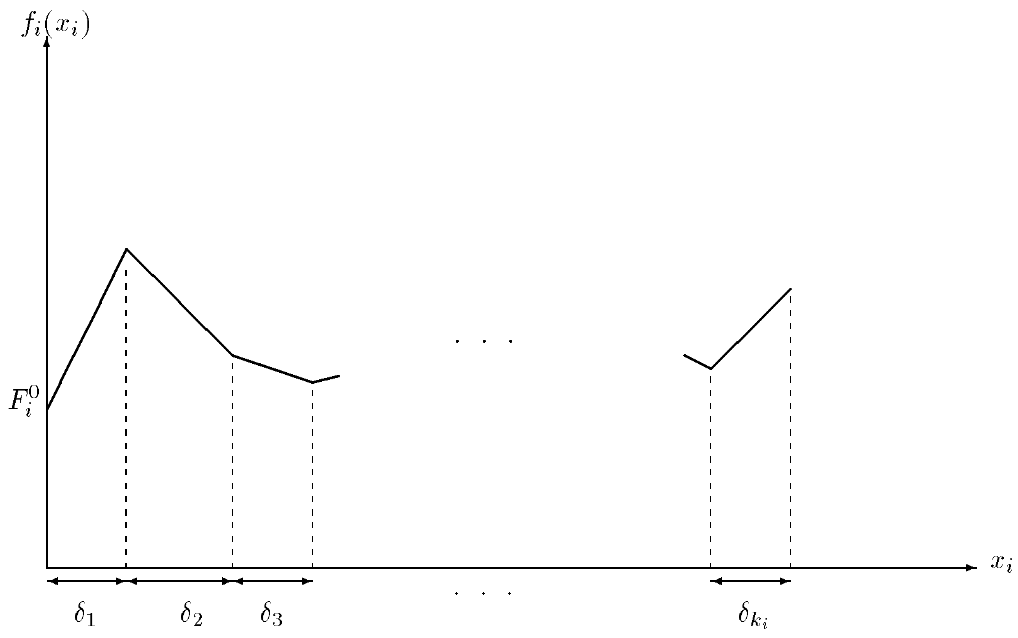



Figure 2: The piece-wise linear function $f_i$.

$$\begin{aligned} y_i^j \geq y_i^{j+1}, \qquad & \forall\ i,j \qquad (2) \\ 0 \leq x_i^j \leq \delta_i^j, y_i^j \in \{0,1\}, \quad & \forall\ i,j \end{aligned}$$

The constraints (2) ensure that

$$y_i^{j+1} = 1 \ \Rightarrow\ y_i^j = 1\ ,$$

and (1) ensures that

$$y_i^{j+1} = 1 \Rightarrow x_i^j \text{ is at its upper bound } \delta_i^j\ .$$

(In fact, the constraints (1) and (2) make the $x_i^j \leq \delta_i^j$ restriction unnecessary, as it will automatically be satisfied).

Note that if the $f_i(x_i)$ are convex, then this problem reduces to LP.

**Disjunctive Constraints:** Given $m$ constraints (inequalities), the goal is to ensure that *at least* $k$ of them are satisfied. Let $A_i y \leq b_i$ be the $i$th constraint. Associate the variable $x_i$ with the satisfaction of constraint $i$; that is, $x_i = 1\ \Rightarrow$ constraint $i$ is satisfied. We then get the following formulation:

$$\begin{aligned} \text{For } i = 1,\ldots,m: \quad & A_i y \leq b_i + M(1 - x_i), \quad \text{for suitably large } M \\ & \textstyle\sum_{i=1}^m x_i \geq k \\ & x_i \in \{0,1\}, \qquad i = 1,\ldots,m \end{aligned}$$

Notice that $x_i$ is not exactly an indicator variable, as constraint $i$ may be satisfied even if $x_i = 0$.

**Traveling Salesman Problem:** Perhaps the most famous of the NP-hard optimization problems is the *traveling salesman problem.* Given $n$ cities, with $c_{ij}$ the cost of going from city $i$ to city $j$ ($c_{ij} = c_{ji}$), the objective is to construct a *tour* (a connected circuit in which each city is visited exactly once, with the salesman returning to the starting point) of minimum total cost.

This problem is somewhat challenging to formulate as an integer linear program, and is left as an exercise. One natural attempt is to associate binary variables $x_{ij}$ with cities $i$ and $j$ being adjacent on the tour, and to require that each city is adjacent to exactly two cities. The difficulty with this formulation is that a disjoint set of cycles may be returned as a solution.

*Clue:* The most natural formulations require a large number of constraints (say $2^n$ constraints for a problem on $n$ cities).

# 2 General Optimization

Consider the general optimization problem:

$$\begin{array}{rl} \text{opt} & f(x) \\ \text{subject to} & x \in \mathcal{S} \end{array}$$

where opt $\in \{\max, \min\}$, $\mathcal{S}$ is the feasible set, and $f : R^n \to R$.

There are several difficult cases, including:

1. *$\mathcal{S}$ is not a convex set.* The main problem here, as can be imagined for $f : R \to R$, is that of resolving global minima for local minima, as each convex region of $\mathcal{S}$ needs to be examined separately.
2. *$f$ is not convex (for minimization problems; or $f$ is not concave for maximization problems).* A similar phenomena occurs, making it difficult to distinguish global minima from local minima, as all minima in the feasible set need to be found and compared.

It is nice when a local minimum is guaranteed to also be a global minimum. Then, one can employ an algorithm based on local search, such as steepest descent. Indeed, for linear programming, the simplex method is a gradient descent approach (Dantzig's rule is a steepest descent implementation) in which the algorithm goes from extreme point to extreme point until a local (which is a global) minimum is reached. Consult David Luenberger's book for further reading on topics concerning local search methods.

Consider a <u>general convex optimization</u> (minimization) problem, which can be described as follows:

$$\begin{array}{rll} \min & f(x) & \\ \text{subject to} & g_i(x) \geq 0, & i = 1, \ldots, m \\ & h_j(x) = 0, & j = 1, \ldots, k \end{array}$$

where $f$ is convex, $g_i$ is concave for $i = 1, \ldots, m$, and $h_j$ is linear for $j = 1, \ldots, k$. The feasible set $\mathcal{S}$ is defined by the functions $g_i$ and $h_j$.

**Lemma 2.1** *The feasible set $\mathcal{S}$ is convex.*

**Proof:** Suppose $x \in \mathcal{S}$ and $y \in \mathcal{S}$. We wish to show that, for any $0 \leq \lambda \leq 1$, $z = \lambda x + (1 - \lambda)y$ is in $\mathcal{S}$.

First consider any of the concave functions $g_i$:

$$\begin{aligned} g_i(z) &= g_i(\lambda x + (1-\lambda)y) \\ &\geq \lambda g_i(x) + (1-\lambda)g_i(y) && (g_i \text{ is concave}) \\ &\geq \lambda \min\{g_i(x), g_i(y)\} + (1-\lambda)\min\{g_i(x), g_i(y)\} \\ &= \min\{g_i(x), g_i(y)\} \\ &\geq 0\ . && (\text{because } g_i(x) \geq 0 \text{ and } g_i(y) \geq 0) \end{aligned}$$

Now consider any of the linear functions $h_j$:

$$\begin{aligned} h_j(z) &= h_j(\lambda x + (1-\lambda)y) \\ &= \lambda h_j(x) + (1-\lambda)h_j(y) && (\text{by linearity of } h_j) \\ &= 0\ . \end{aligned}$$

■

**Lecturer:** Dorit Hochbaum
**Scribe:** Youxun Shen

# 1 Last class

- IP formulation of nonlinear objective including production problem, piecewise linear objective, Disconjunctive constraint.
- General optimization is difficult if objective is not convex (for min) or feasible region is not convex.

# 2 Today

- Putting IP in perspective of general optimization.
- Introducing notions of Local and global optima and exact neighborhood.

# 3 A general convex minimization problem

$$\begin{array}{lll} \text{Minimize} & f(x) & \\ \text{Subject to:} & g_i(x) \geq 0 & i = 1, 2, \ldots, m \\ & h_i(x) = 0 & j = 1, 2, \ldots, k \end{array}$$

where $f$ is convex, $g_i$ is concave and $h_j$ is linear.

Since for convex programming problems any locally optimal solution is globally optimal as well, we can employ a local search technique, steepest descent say, to solve the problems. Further, if $f$ is linear, then the problem also belongs to the realm of concave programming, thereby enabling us to focus our effort on finding the optimal solution on the boundary, which is a surface with lower dimension than the whole Euclidean space. If we also let $g_i$ be linear, then the problem is the well-known LP problem, with the salient feature that we can always achieve the optimum at a vertex of the polyhedron defining the feasible set.

In the LP case, the problem is somewhat discrete in nature because we can just enumerate all the vertices in search of an optimal point. However, this method usually is hindered for we may have to check as many as $\begin{pmatrix} m+n \\ m \end{pmatrix}$ vertices, a number which is exponentially growing with problem size. But with the friendly property of convex programming in mind, we can get around through resorting to the steepest descent approach, which gives rise to the famous Simplex Method in particular. Although in the last 15 years the world has witnessed the emergence of polynomial algorithms for LP like ellipsoid algorithm and interior-point algorithm, these algorithms implemented directly are not the ones of choice for real world applications.

## 4 Integer Linear Programming

If we restrict the decision variables in an LP problem to be integers, then we get the so-called Integer Linear Programming problem. This simple added restriction turns out to destroy the convexity of the feasible region, thus making the usual local search technique break down. One may argue that we can solve the LP-relaxation instead and then round the solutions up or down to get a near-optimal solution. Usually, if it is known a priori that the optimal variables assume rather large value, this heuristic might be faring very well. Nonetheless, in many combinatorial optimization problems, we will be dealing with binary variables and rounding is essentially equivalent to exhausting all the possible combinations. Also, as unfolded in the pretty example in the handout, rounding may not lead to any feasible solution.

Next let us get a feel about the magic of number 2 in combinatorial optimization. Problems involving edges are typically easier, as edges have two endpoints, but vertices have an arbitrary number of edges adjacent to them. For example (see Figure 3 for problem definitions):

| **Hard** | **Easy** |
|---|---|
| Vertex packing (Independent Set) | Edge packing (Matching) |
| Traveling Salesman Problem (Vertex tour) | Chinese Postman Problem (Edge tour) |
| 3-CUT with specified vertices | 2-CUT with specified vertices |
| 3-SAT | 2-SAT |

Moreover, problems with even input are usually solved more efficiently. For example,

- b-matching for b even
- Vertex cover with the requirement to cover each edge twice or an even number of times
- Integer programming for which each inequality has at most two variables in the left hand side and has even right hand side and variables are bounded with some fixed number
- Set cover where each set is of size 2 or less

## 5 Exact neighborhood

Given an optimization problem with feasible set $F$ and a neighborhood $N$, if whenever $f \in F$ is locally optimal with respect to $N$ is also globally optimal, we say the neighborhood $N$ is exact.

Let us illustrate this concept through Minimum Spanning Tree (MST) problem. The problem is to find a spanning tree on a graph with minimum total weight. Following are equivalent characterizations for T to be a spanning tree:

1. T is a connected acyclic graph
2. T is an acyclic graph with $n - 1$ edges

---

**Vertex Packing (Independent Set):** In a graph $G = (V, E)$, find a maximum weight subset of vertices, $V' \subseteq V$, such that no two vertices in $V'$ are adjacent.

**Edge Packing (Matching):** In a graph $G = (V, E)$, find a maximum weight subset of edges, $E' \subseteq E$, such that no two edges in $E'$ are incident.

**Traveling Salesman:** Find a minimum cost tour that visits each vertex exactly once.

**Chinese Postman:** Find a minimum cost tour that visits each edge at least once.

**$k$-CUT:** Given a network $G = (V, E)$, partition the vertices into $k$ subsets such that the weighted sum of those edges that have endpoints in different subsets is minimized. The *"with pre-specified vertices"* version of the problem adds the restriction that particular vertices must be in particular subsets.

**$k$-SAT:** Same as *SAT*, but restricted to instances in which each clause has at most $k$ literals.

---

Figure 3: Problem Definitions

3. T is connected and has $n - 1$ edges

In the MST, we can define a neighborhood of a spanning tree $T$ to be the set of spanning trees obtained as follows: add an edge $e$ to the tree $T$, producing a cycle; then deleting any edge on the cycle. It can be established that this neighborhood is exact, which is intimately related to the fact that MST can be solved in polynomial time.

Roughly speaking, we are better off if we can find a nice exact neighborhood for an optimization problem. Intuitively exact neighborhood serves as an indicator for the hardness of the problem.

**Lecturer:** Dorit Hochbaum
**Scribe:** Youxun Shen

**Last class**

1. IP in perspective
2. The magic of number 2
3. Definition of exact neighborhood, in conjunction with its applications to TSP and MST

**Today**

1. More on exact neighborhoods
2. Complexity of algorithm or problem, including measuring input size, measuring number of steps and drawbacks of complexity analysis (illustrated with Ellipsoid vs. Simplex), etc.

# 1 Exact neighborhood cond.

The exact neighborhood implies the following algorithm:

1. Find a feasible solution
2. Apply (steepest) descent search to move to another feasible solution in the neighborhood. If there exists no better solution in the neighborhood, then the current solution is optimal.

**Claim 1.1** *The implied algorithm for MST is finite.*

**Proof:** Since the objective value is strictly moving downhill, the number of steps is at most the number of feasible solutions. No feasible solution is visited more than once. ■

Nonetheless, the algorithm is not necessarily efficient since there may be too many descent steps before the optimal solution is found.

**Theorem 1.1** *If $T$ is a local optimum with respect to the neighborhood for MST defined in last class, then it is globally optimal as well.*

**Proof:** See the solution for Homework 1.
Hint: Look at the symmetric difference between T and T$^*$, where T$^*$ is a minimum spanning tree whose symmetric difference with $T$ has minimum cardinality amongst all minimum spanning trees. ■

Intuitively, the larger the size of the exact neighborhood, the harder the problem. For instance, $n$-edge exchange neighborhood is exact for TSP, but the size of this neighborhood is huge.

## 2 Complexity Analysis

In order to differentiate hard and easy problems, we need to associate with each algorithm for the problem a "reasonable" measure of complexity, e.g the number of steps. We can expect that the the number of basic operations is closely related to the size of the problem description, namely, input size. Consequently, complexity will be a function of input size.

One common approach is to define the input size as the length of the binary string needed to represent the input data. For example, the input size for a linear program is $\Theta(mn + \lceil \log |P| \rceil)$, where $m$ and $n$ are the dimensions of the constraint matrix and $P$ is the product of all nonzero coefficients.

Moreover, in order to compute time bounds, we have to specify what we mean by basic operations. Usually, multiplication, addition and comparison are deemed basic and they all require one unit time even though the physical time for implementing those operations might be different. One caveat is to avoid using square root as basic operation in the complexity analysis since in most cases it is impossible to use a polynomial number of the aforementioned basic operations to compute the square root within infinite accuracy.

For the ease of analysis, we usually consider the worst-case complexity. Mathematically, the complexity of an algorithm can be defined as the supremum of the number of basic operations required by the algorithm over all possible instances of the problem, while the complexity of a problem is the infimum of the complexity of all possible algorithms for the problem. As a rule of thumb, in most cases we are interested in the asymptotic behavior in conducting complexity analysis. For this purpose, we introduce the notations of $O$. $\Theta$ and $\Omega$. For detailed coverage, please consult the handout on "Complexity Analysis".

To fix ideas, let us consider the 0-1 knapsack problem, taking the following form:

$$\begin{array}{lll} \text{Minimize} & G_n(S) = \sum_{i=1}^{n} u_i x_i & \\ \text{Subject to:} & \sum_{i=1}^{n} v_i x_i \leq B & i = 1, 2, \ldots, n \\ & x_i \in \{0, 1\} & i = 1, 2, \ldots, n \end{array}$$

For this problem, the input size is $n + \sum_i \log |u_i| + \sum_i \log |v_i| + \log |S|$.

For $j = 1, \ldots, n$ and $S = 0, \ldots, B$, we have the following recurrence relation:

$$G_j(S) = \max \{G_{j-1}(S - v_j) + u_j, G_{j-1}(S)\}$$

In conjunction with the boundary condition $G_j(0) = 0, \forall j$, this enables us to use dynamic programming scheme to solve this problem. A straightforward analysis gives the time bound $\Theta(nB)$, which is pseudo-polynomial.

Finally, let us list some of the drawbacks of complexity analysis.

1. It is a worst-case analysis, and may not have much correlation with average behavior.
2. It is asymptotic, e.g. $10^{10}n$ is preferred to $2^{\frac{n}{10^{10}}}$.

*An example:* The Ellipsoid algorithm is known to be polynomial, while the Simplex algorithm can exhibit exponential behavior for some pathological problems. However, in practice, Simplex outperforms Ellipsoid. For many problems, the observed running time for Simplex is about three or four times the number of constraints. On the contrary, Ellipsoid can assume its worst-case bound for many instances. Note that Ellipsoid is not strong polynomial; it highly depends on the parameters in the constraint matrix.

---

**Lecturer:** Dorit Hochbaum
**Scribe:** Thomas R. Nitsch

**Assignment:** Read Chapter 15 [P&S]

---

## Last Class

- exact neighborhoods and an implied algorithm
  a) find feasible solution
  b) search for better solution in neighborhood (size of neighborhood is important)
  c) number of steps till termination (finite)
- complexity of algorithms and problems
  - definition
  - measuring input size (binary bits)
  - measuring number of operations
  - comparing functions (asymptotic analysis)

  The notions of $O$, $o$, $\Omega$, and $\Theta$ correspond to an ordering of functions, with

  $f \in \mathbf{O}(g)$ corresponding to $f \leq g$;
  $f \in \mathbf{o}(g)$ corresponding to $f < g$;
  $f \in \mathbf{\Omega}(g)$ corresponding to $f \geq g$;
  $f \in \mathbf{\Theta}(g)$ corresponding to $f \cong g$.

## Drawbacks of Complexity Analysis

1. It is a worst case analysis

   ex: SIMPLEX is an *exponential* algorithm - for the family of Klee-Minti problems it takes $O(2^n)$ iterations. However, in practice, SIMPLEX works efficiently. Also, for several random distributions SIMPLEX has been proven to be polynomial.

   The ELLIPSOID method on the other hand has running time $O(n^2L)$ and is hence a *polynomial* algorithm. However, on the average (and in practice) it takes the worst case number of iterations, which is excessive.

2. It is an asymptotic analysis, so we get aberations such as:

   $f = 2^{10^6} n$ is $O(n)$;
   $g = 2^{n/10^6}$ is an exponential function in $n$.

## ELLIPSOID Method

$$\begin{array}{rll} \min & cx & a_{ij}, b_i, c_j \text{ are integers} \\ \text{subject to} & Ax \geq b & \\ & x \geq 0 & \end{array}$$

ELLIPSOID can establish the feasibility of a system of strict inequalities. This suffices to solve LP.

### Facts

1. $\exists$ a *polynomially* bounded interval, $[-\frac{D}{2}, \frac{D}{2}]$, of length $D$ for the value of the optimal solution (if feasible and bounded)

2. $\exists$ finite accuracy to an optimal solution, i.e. a solution is obtained on an $\epsilon$-grid ($\epsilon$ is polynomially dependent on the input data).

**ELLIPSOID as a blackbox** Given a blackbox that, for a system of inequalities, returns either a feasible solution or states the infeasibility of the problem - how can we apply the blackbox to solve a given problem ?

**a) Binary Search**

$$\begin{array}{rcl} cx & < & M \quad \text{where } M \text{ is the median point of the bounded interval.} \\ Ax & \geq & b \\ x & \geq & 0 \end{array}$$

If feasible solution
$\Rightarrow$ optimal solution $\in$ lower half of $D$ and $M$ is updated to be the median of the lower interval; if not, then $M$ is updated to be the median of the upper interval.
Since the interval size shrinks by a factor of 2 at each iteration, we need at most $\lceil log_2 \frac{D}{\epsilon} \rceil$ calls to ELLIPSOID to solve the problem.
After k calls we know the optimal solution is in an interval of length $(\frac{1}{2})^k D$;

$k$ has to be such that: $(\frac{1}{2})^k D < \epsilon \Rightarrow \frac{D}{\epsilon} < 2^k$

**Note:** We need to know that $D$ is not too large and that the final interval of length $\epsilon$ is also polynomial in the size of the input (i.e. to establish facts 1 and 2).

**b) Primal-Dual**

$$\begin{array}{rcl} cx & \leq & yb \\ x & \geq & b \\ x & \geq & 0 \text{ contraints of primal} \\ yA & \leq & c \\ y & \geq & 0 \text{ contraints of dual} \end{array}$$

**Other: Moving Cut Implementation.** Find a feasible solution to

$$Ax \geq b, \; x \geq 0 \; .$$

Whenever a feasible solution $\bar{x}$ is found, a constraint $cx < c\bar{x}$ is added.

We now prove both facts referred to earlier.

**Proof:** Fact (1)

If feasible and bounded then $\exists$ extreme point solution *(bfs)*. Let $B_{m*m}$ be the basic column matrix

$$\begin{aligned}
Bx_B &= b \qquad\qquad (X_N = 0) \\
x_B &= B^{-1}b \overset{[\text{Cramer'sRule}]}{=} \frac{1}{detB} C^T b \qquad \text{where } C = \text{cofactor matrix of } B \\
B_{ij} &= det(B) \text{with the } i\text{-th row and the } j\text{-th column taken out}
\end{aligned}$$

Let $\alpha = max_{i,j}|a_{ij}| \;\; \beta = max_i|b_i| \;\; A = (a_{ij})$

$$\begin{aligned}
|det\ B| &= \left| \sum_{\sigma \in S_m} sign(\sigma) \; \prod_{i=1}^{m} a_{i\sigma(i)} \right| \\
&\leq \sum_{\sigma \in S_m} \prod_{i=1}^{n} |a_{i\sigma(i)}| \\
&\leq m! \; \alpha^m
\end{aligned}$$

By the same argument $|B_{ij}| \leq (m-1)! \; \alpha^{m-1}$.
The numerator is then bounded by

$$\begin{aligned}
\left| \sum_{i=1}^{m} B_{ij} \; b_i \right| &\leq m \; (m-1)! \; \alpha^{m-1}\beta \\
&= m! \; \alpha^{m-1}\beta \\
(X_B)_j &= \frac{\sum_i B_{ij} \; b_i}{det \; B} \\
\Rightarrow \; |X_j| &\leq m! \; \alpha^{m-1}\beta \\
\|X\|_\infty &\leq m! \; \alpha^{m-1}\beta
\end{aligned}$$

To write down the solution it takes $m\ log\ m + (m-1)\ log\ \alpha + log\ \beta$ . This number is polynomial in the length of the input (L). Hence,

$$\begin{aligned}\text{Length of D} &\leq log\ |C_max| + log\ n + m\ log\ m + \\ &\quad +(m-1)\ log\ \alpha + log\ \beta \\ &< m\ L\end{aligned}$$

■

**Proof:** Fact (2)

Consider the distance between 2 bfs's: $X_{B_1}, X_{B_2}$. Let $\Delta$ be the largest subdeterminant (of size $m \times m$) of $A$.

$$\begin{aligned}|X_{B_1} - X_{B_2}| &\geq \frac{\cdots}{\Delta} - \frac{\cdots}{\Delta - 1} \\ &= \left|\frac{\text{“some integer”}}{\Delta\ (\Delta - 1)}\right| \\ &\geq \frac{1}{\Delta^2}\end{aligned}$$

Also, $\Delta \leq 2^L$. So, accuracy of $2^{-2L}$ suffices.

The Ellipsoid method is used to solve LP:

**(i) To rule out infeasibility**

- Given the black box (Ellipsoid method) the system

$$\begin{aligned}A\ x &\leq b \\ x &\geq 0\end{aligned}$$

is checked for feasibility.

**(ii) To rule out unboundedness**

- check for dual feasibility (if primal unbounded, dual is infeasible)

$$\begin{aligned}yA &\leq c \\ y &\geq 0\end{aligned}$$

- or, solve for $M > 2^L$

$$\begin{aligned}c\ x &\leq -M - 1 \\ A\ x &\geq b \\ y &\geq 0\end{aligned}$$

■

**Statement:** Suffices to solve a system of *strict* inequalities

**Lemma 0.1 ((P&S) 8.7, p.173-174)**

$$\begin{array}{cccc} \textit{(LI)}\ a_i\ x & \leq & b_i & (i = 1, ..., m) \\ & \textit{has a solution} & & \\ & \Updownarrow & & \\ \textit{(LSI)}\ a_i\ x & < & b_i + 2^{-2L} & (i = 1, ..., m) \\ & \textit{has a solution} & & \end{array}$$

**Proof:** (LI) has a feasible solution $\Rightarrow$ (LSI) has a feasible solution. See [PS] for the other direction. ■

**Corollary 0.1** *(LI) is solvable in polynomial time* $\Leftrightarrow$ *(LSI) is solvable in polynomial time.*

**Proof:** Can reduce one problem to the other in polynomial time with only a polynomial expansion in the size of the input.

$$\begin{aligned} a_i\ x &< b_i + 2^{-2L} \\ 2^{2L}\ a_i\ x &< 2^{2L}\ b_i + 1 \end{aligned}$$

If (LSI) has a solution in polynomial time in the length of the input $(2L * L)$ then the polynomial function $(2\ L^2)$ is also polynomial in the size of the input L. ■

**Lecturer:** Dorit Hochbaum
**Scribe:** Ilya Vinarsky

Last class, we finished with the ellipsoid method, and defined optimization, evaluation and recognition problems.

1. Optimization: find an optimal solution to a problem.
2. Evaluation: find *the value* of an optimal solution.
3. Recognition: given $k$, find out if there exists a solution to the optimization problem with value $\geq$ (or $\leq$) $k$.

Today we will discuss search (aka optimization) and decision (aka recognition) problems, the notion of certificate and the definitions of NP and NP-complete.

# 1 Search vs. Decision

An important problem for which the distinction is significant is primality. It is an open question whether or not there exist polynomial-time algorithms for testing whether or not an integer is a prime number. However, there are reasonably fast superpolynomial-time ones, and also Miller-Rabin's algorithm, a randomized algorithm which runs in polynomial time and tells either "I don't know" or "Not prime" with a certain probability. However, for the corresponding search problem, finding all factors of an integer, no similarly efficient algorithm is known.

*Another example*: A graph is called k-connected if one has to remove at least k vertices in order to make it disconnected. A theorem says that there exists a partition of the graph into $k$ connected components of sizes $n_1, n_2 \ldots n_k$ such that $\sum_{i=1}^{k} n_k = n$ (the number of vertices), such that each component contains vertices $v_1, v_2 \ldots v_k$. The optimization problem is finding a partition such that $\sum_i |n_i - \bar{n}|$ is minimal. No polynomial-time algorithm is known for $k \geq 4$. However, the corresponding recognition problem is trivial, as the answer is always *Yes* once the graph has been verified (in polynomial time) to be $k$-connected. So is the evaluation problem: the answer is always 0.

# 2 NP and co-NP

A very important class of decision problems is called *NP*, which stands for nondeterministic polynomial-time. It is an abstract class, not specifically tied to optimization.

**Definition 2.1** *A decision problem is said to be in NP if for all "Yes" instances of it there exists a polynomial-length "certificate" that can be used to verify in polynomial time that the answer is indeed Yes.*

*Example:* a proof of a theorem is a certificate telling that the theorem is correct. However, since only one proof is given, it is unclear what "polynomial time" means in this case.

*Another example:* The dominating set problem. A vertex in a graph is said to dominate itself and all its neighbors. The problem is, given a graph, to find the smallest set of vertices that dominates all vertices. This is an optimization problem. A corresponding recognition problem is finding whether or not there is a dominating set $S$ such that $|S| \leq k$ for a given $k$. The dominating set problem is in *NP* because there is a certificate, namely, the set $S$, such that its length is polynomial in length of the input (the graph in any reasonable representation), that can be verified in polynomial time (for example, by going through every vertex and checking if it has a neighbor from $S$). In fact, the dominating set problem is known to be *NP-complete.*

The certificate problem is less trivial for primality. In fact, it was only in 1976 that Pratt proved that Primality∈*NP*.

For linear programming, a recognition problem takes form: Are there $x_1 \geq 0, x_2 \geq 0 \ldots x_n \geq 0$ such that for some $k$ $\sum_{i=1}^{n} c_i x_i \leq k$ and $A\vec{x} \leq b$ ? A certificate for this problem would be a feasible vector $\vec{x}$ for which the inequality would hold. These two properties are checkable in polynomial-time. So LP∈*NP*.

Suppose the answer to the recognition problem is *No*. How would one certify this? If a number is composite, a factor would be a good certificate to verify that the number is *not* prime. If a linear program is infeasible, a feasible nonoptimal solution of its *dual* would do (because of the Weak Duality Theorem).

**Definition 2.2** *The class of problems for which the certificate exists for all "No" instances is called co-NP.*

So LP∈*NP*∩*co-NP*, as is Primality.

*Another example*: the Minimum Spanning Tree problem (in its decision version - for a given graph, is there a spanning tree of weight $\leq k$) is (obviously) in *NP*. It is also ∈*co-NP* because there is a polynomial-time algorithm that builds the tree.

**Definition 2.3** *Let P be the class of polynomial-time (as a function of the length of the input) solvable problems.*

It is easy to see that *P*⊆*NP*∩*co-NP* - just find an optimal solution. It is not known, however, whether, *P*=*NP*∩*co-NP*; the problems that are in the intersection but for which no polynomial-time algorithm is known are usually very pathological. Primality ∈ *NP*∩*co-NP*; this gives rise the conjecture that primality testing can be carried out in polynomial time.

For integer linear programming, a "*Yes*" certificate is a feasible vector, just like general LP. It can be certified in polynomial time; it is also necessary to prove that the entries in the vector are not too large, so that the certificate is only polynomially long. This is done in the book. So ILP∈*NP*.

## 3 NP-completeness and reductions

Consider the following two problems, called *k-center* and *k-median*. The first one is as follows: given a graph $G = (V, E)$ weighted by $w : V \times V \mapsto R^+$, is there a subset $S \subset V$, $|S| = k$, such that $\forall v \in V \setminus S \exists s \in S$ such that $w(v, s) \leq m$ for a given $m$? For example, it may be necessary to locate emergency facilities of some kind so that the maximum time from any point to the facility would be minimal. The second problem consists of choosing $S$ so that $\sum_{v\in V\setminus S} \min_{s\in S} w(v,s)$ is less than or equal to $M$ for a given $M$. This problem deals with "total social good" rather than the worst case.

For both problems, a certificate is a subset of $V$. So, both problems are in *NP*.

A major open question in contemporary computer science is whether *P=NP*. It is currently believed that *P≠NP*. Another open question is whether *NP=co-NP*. It is believed that *NP≠co-NP*.

As long as these two conjectures remain unproven, instead of proving that a certain *NP* problem is not in *P*, we have to be satisfied with a slightly weaker statement: if the problem is in *P* then *P=NP*, that is, a problem is "at least as hard" as any other *NP* problem. Cook has proven this for a problem called *Satisfiability*; for other problems, this can be proven using *reductions*.

There are two definitions of reducibility, Karp and Turing; they are known to describe the same class of problems. The following definition uses Karp reducibility.

**Definition 3.1** *A problem $P_1$ is said to reduce in polynomial time to problem $P_2$ if there exists a polynomial-time algorithm $A_1$ for $P_1$, where each call to a subroutine solving $P_2$ is counted as a single operation.*

If $P_2 \in P$ then $P_1 \in P$, because the ring of polynomials is closed under substitutions. Turing reductions allow for *one* call to the subroutine.

*Example*: the *k-center* problem is at least as hard as the *Dominating set* problem, defined as follows: for a given graph, is there a set of vertices, called *dominating* such that every vertex not in the set is adjacent to one in the set?. The reduction proceeds as follows: Given a graph $G = (V, E)$, define a weight function on $K_{|V|}$ (the complete graph with $|V|$ vertices) to be 0 if the two vertices share an edge in $G$, and 1 if they do not. Then, to find a dominating set in $G$ it is sufficient to find a 0-center in $K_{|V|}$ with this weight function. There only nontrivial computation involved is building $K_{|V|}$, which is clearly polynomial-time.

**Definition 3.2** *A problem is said to be NP-complete if it is in NP and all problems in NP can be reduced to it in polynomial-time.*

The *Dominating set* problem is known to be *NP-complete*; therefore, so is *k-center*.

**Lecturer:** Anu Pathria
**Scribe:** Anu Pathria

# 1 *NP* and *co-NP*

**Definition:** A recognition problem is said to be in *NP* if for every "yes" instance there is a certificate, whose size is polynomial in the input, that can verify in polynomial time that the answer is indeed "yes".

*Examples:* Non-primality (certificate is a non-trivial divisor), TSP (certificate is an ordered list of cities in a tour), LP (certificate is a feasible solution – the fact that a *polynomial* size solution exists follows from the fact that all basic solutions are of size polynomial in the input).

Imagine that you are a *verifier* and that you want to be able to confirm that the answer to a given recognition problem is indeed "yes". Problems in *NP* have certificates that, without necessarily indicating how the answer was obtained, allow for this verification in polynomial time.

For a definition of *co-NP*, replace "yes" by "no" in the definition of *NP*; that is, for problems in *co-NP* there exist certificates that allow for polynomial time verification of "no" instances. As a verifier, it is clearly nice to have certificates to confirm both "yes" *and* "no" answers; that is, for the problem to be in both *NP* and *co-NP*. For many problems in *NP*, such as TSP, there is no obvious polynomial "no" certificate .

**Intersection of *NP* and *co-NP*:** Consider those problems that are in $NP \cap co\text{-}NP$. When is a recognition problem $\mathcal{Q}$ in this intersection? We consider some cases:

1. *Polynomial:* Consider a problem $\mathcal{Q}$ for which there exits an algorithm with polynomial running time; for example, the recognition version of MST (*"Is there a spanning tree with cost less than or equal to $K$"*) can be answered in polynomial time. Such problems are in $NP \cap co\text{-}NP$: the algorithm itself (along with a proof of the algorithm's correctness) is the certificate for both "yes" and "no" instances, and verification can be carried out in polynomial time by running the algorithm.
2. *"Nice" Exact Neighborhoods:* Consider an optimization problem that has an exact neighborhood (i.e. a local optimum is a global optimum) which can be searched in polynomial time (this is what we mean by "nice"). For example, as seen in the homework, MST has a "nice" exact neighborhood (those trees that can generated by replacing a single edge of the current tree by a single non-tree edge). Assuming that the exact neighborhood can be searched in polynomial time, the recognition problem

corresponding to such problems is in both *NP* and *co-NP*: the "yes" certificate is an optimal solution; the "no" certificate is also an optimal solution (along with a description of the neighborhood). So, even if we were not aware of a polynomial algorithm for MST, the existence of a nice exact neighborhood is enough to tell us that MST is in both *NP* and *co-NP*.

3. *Min-Max Problems:* Consider optimization problems, like LP, which satisfy so-called *min-max* relationships; in the case of LP (assume the LP problem is a minimization problem), the minimum of the primal is equal to the maximum of the dual. The recognition versions corresponding to such min-max problems, assuming that solutions to the min and max problems can be verified and evaluated in polynomial time, are in $NP \cap co\text{-}NP$: a "yes" certificate is an optimal solution to the min problem; a "no" certificate is an optimal solution to the corresponding max problem.

   For example, LP is in *NP* and *co-NP*. Similarly, the decision version of min s-t cut (*"Is there an s-t cut with capacity less than or equal K?"*) is in *NP* (a certificate is such a cut) and in *co-NP* (a certificate is a flow whose values exceeds $K$).

Notice that the fact that a problem belongs to either of the last two categories does not automatically suggest a polynomial time algorithm for the problem; however, being in $NP \cap co\text{-}NP$ is usually considered as strong evidence that there *exists* a polynomial algorithm. Indeed, many problems were known to be in $NP \cap co\text{-}NP$ well before a polynomial algorithm was developed (ex. LP, non-bipartite matching); this membership was considered to be strong evidence that there was a polynomial algorithm out there, giving incentive for researchers to put their efforts into finding one.

On the flip-side, *NP*-complete problems, which we turn to next, are considered to be immune to any guaranteed polynomial algorithm.

# 2 *NP*-Completeness

**Definition and Implications:** A recognition problem $\mathcal{Q}$ is said to be *NP*-complete if:

1. $\mathcal{Q} \in NP$.
2. All other problems in *NP* are polynomially reducible to $\mathcal{Q}$. ($\mathcal{Q}_1$ is said to be *polynomially reducible* to $\mathcal{Q}$ (written as "$\mathcal{Q}_1 \propto \mathcal{Q}$") if, given an oracle that answers $\mathcal{Q}$ at unit cost, $\mathcal{Q}_1$ can be answered in polynomial time. It follows that if *any* *NP*-complete problem were to have a polynomial algorithm, they all would.)

**Conjecture 2.1** *$P \neq NP$.*

So, we do not expect any polynomial algorithm to exist for an *NP*-complete problem. Therefore, if $\mathcal{Q}$ is *NP*-complete, we should not expect there to exist a nice exact neighborhood for the optimization problem associated with $\mathcal{Q}$. (ex. partition from homework). We will see later in the course that *simulated annealing* is a search technique that tries to deal with the fact that a neighborhood is not exact; that is, that a local minimum may not be a global one.

***SAT* is *NP*-Complete:** Cook and Levin independently established that all problems in *NP* could be polynomially reduced to *SAT*. See Figure 4 for a definition of *SAT*.

**Theorem 2.1 (Cook-Levin (1970))** *Let $X$ be an instance of a problem $\mathcal{Q}$, where $\mathcal{Q} \in$ NP. Then, there exists a SAT instance $F(X, \mathcal{Q})$, whose size is polynomially bounded in the size of $X$, such that $F(X, \mathcal{Q})$ is satisfiable if and only if $X$ is a "yes" instance of $\mathcal{Q}$.*

**Proof:** In very vague terms: Consider the process of verifying the correctness of a "yes" instance, and consider a Turing machine that formalizes this verification process. Now, one can construct a *SAT* instance (polynomial in the size of the input) mimicking the actions of the Turing machine, such that the *SAT* expression is satisfiable if and only if verification is successful. More details will be given in the next lecture. ∎

**Corollary 2.1** *SAT is NP-complete.*

**Proof:** Clearly SAT is in *NP*. Completeness follows from the above theorem. ∎

**Reductions:** Karp [1] pointed out the practical importance of Cook's theorem, via the concept of reducibility. To show that a problem $\mathcal{Q}$ is *NP*-complete, one need only demonstrate two things:

1. $\mathcal{Q} \in NP$.
2. Show that $\mathcal{Q}_1 \propto \mathcal{Q}$, for some *NP*-complete problem $\mathcal{Q}_1$.
   Because all problems in *NP* are polynomially reducible to $\mathcal{Q}_1$,

$$\mathcal{Q}_1 \propto \mathcal{Q} \;\Rightarrow\; \text{all problems in } NP \text{ are polynomially reducible to } \mathcal{Q}.$$

Because all problems in *NP* are polynomially reducible to $\mathcal{Q}_1$, if $\mathcal{Q}_1 \propto \mathcal{Q}$, it follows that $\mathcal{Q}$ is at least as hard as $\mathcal{Q}_1$ and that all problems in *NP* are polynomially reducible to $\mathcal{Q}$.

Starting with SAT, Karp produced a series of problems (18) that he showed to be *NP*-complete. [2]

Examples of proving *NP*-completeness using reductions (see Figure 4 for problem definitions):

1. *0-1 Knapsack* is *NP*-complete.

   **Proof:** First of all, we know that it is in *NP*, as a list of items acts as the certificate. Now, we show that *Partition* $\propto$ *0-1 Knapsack*. Consider an instance of *Partition*. Construct an instance of *0-1 Knapsack* with $w_i = c_i = b_i$ for $i = 1, 2, \ldots, n$, and $W = K = \frac{1}{2}\sum_{i=1}^{n} b_i$. Then, a solution exists to the *Partition* instance if and only if one exists to the constructed *0-1 Knapsack* instance. ∎

   Exercise: Try to find a reduction in the opposite direction (such a reduction is given in in P&S).

2. *Independent Set* is *NP*-complete.

[1] Reference: R. M. Karp. *Reducibility Among Combinatorial Problems*, pages 85–103. Complexity of Computer Computations. Plenum Press, New York, 1972. R. E. Miller and J. W. Thatcher (eds.).

[2] For a good discussion on the theory of *NP*-completeness, as well as an extensive summary of many known *NP*-complete problems, see: M.R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.

**SAT:**

*Instance:* A list of boolean variables and a set of clauses, where each clause is a disjunction of some of the variables and/or their negations.

*Question:* Is there an assignment of *true/false* to the variables such that each clause is satisfied?

**Partition:**

*Instance:* A list of integers $b_1, b_2, \ldots, b_n$.

*Question:* Does there exist a subset $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in S} b_i = \sum_{i \notin S} b_i$ ?

**0-1 Knapsack:**

*Instance:* Two lists of numbers $w_1, w_2, \ldots, w_n$ and $c_1, c_2, \ldots, c_n$, and two values $W$ and $K$.

*Question:* Does there exist an assignment to the 0-1 variables $x_1, x_2, \ldots, x_n$ such that $\sum_{i=1}^{n} w_i x_i \leq W$ and $\sum_{i=1}^{n} c_i x_i \geq K$?

**Independent Set:**

*Instance:* A graph $G = (V, E)$, and an integer $K$.

*Question:* Is there a subset $S \subseteq V$, with $|S| \geq K$, such that there is no edge $e \in E$ that has both of its endpoints in $S$?

Figure 4: Definitions of some *NP*-complete problems.

**Proof:** Clearly *Independent Set* $\in$ *NP*, as a list of the vertices in the independent set suffices as a certificate.

Now, we show that *SAT* $\propto$ *Independent Set*. Suppose you are given an instance of SAT, with $k$ clauses. Form a graph that has a component corresponding to each clause. For a given clause, the corresponding component is a complete graph with one vertex for each variable in the clause. Now, connect two nodes in different components if and only if they correspond to a variable and its negation. Now, it is easy to see that this resulting graph has an independent set of size at least $k$ if and only the *SAT* instance is satisfiable. ∎

In this course, we explore several approaches for dealing with *NP*-complete problems.

Integer Programming and Combinatorial Optimization — **Lecture #9**
IEOR 269 — **September 20, 1994**

---


**Lecturer:** Dorit Hochbaum
**Scribe:** Wooseung Jang


---

Today we will learn more about *SAT* , and begin discussion of Turing machines and Cook's theorem.

*SAT*

A *2-SAT* formula : $(u_1 \vee \overline{u_2}) \wedge (\overline{u_1} \vee u_2)$
Truth assignments : $u_1 = u_2 = T$ or $u_1 = u_2 = F$.
*2-SAT* (as a decision problem) is polynomial. However, maximizing the number of satisfied clauses is *NP-complete* even for *2-SAT*. Minimize the weight of true variables in a truth assignment (variable $i$ has weight $w_i$) is also *NP-complete* for *2-SAT* (even for $w_i = 1$).

## Zero Knowledge Proofs

**Graph Isomorphism:** One of the major open problems is to determine if this problem is *NP-complete* or polynomial.
*Input:* $G_1, G_2$.

Prover generates a graph $H$ by a random permutation of (say) $G_1$. Now verifier flips a coin. If it is head, request the permutation verifying that $H \approx G_1$. If tail, request the permutation verifying $H \approx G_2$. If prover is cheating, the probability of accepting that $G_1 \approx G_2$ is $(\frac{1}{2})^k$ for $k$ coin flips. So we can be sure that $G_1 \approx G_2$ with probability $\geq 1 - (\frac{1}{2})^k$. But verifier knows nothing of the isomorphism between $G_1$ and $G_2$. In fact this form of zero knowledge certificates is possible for all *NP*problems.

Can we give this kind of proof (interactive proof) also for "no " answer? Prover claims "no", $G_1$ is not isomorphic to $G_2$. Verifyer flips a coin to choose 1 or 2. If it is 1, then construct by random permutation $H \approx G_1$. Ask prover to show that $H$ is isomorphic to one of $G_1$ or $G_2$. If prover gave the right response, then with probability $\frac{1}{2}$ verifyer is convinced that the answer is "no". Prover cheats successfully in $r$ iteration with probability $(\frac{1}{2})^k$. So, zero-knowledge co-NP certificates are possible with interactive proofs.

## Proof of Cook's theorem

Any certificate checking algorithm can be represented by the following Turing machine model.

○read–write head
↓

| instance : $x$ | \$ | certificate : $c(x)$ |
|---|---|---|

The read-write head reads and modifies the symbols in a string – one position at a time, starting with the left most position.
Instruction :

$\#l$ : If $\sigma$ then $(\sigma', \delta, l')$.
where $\sigma$ means read a symbol $\sigma$,
$\sigma'$ means erase and write $\sigma'$ instead,
$\delta$ means move $\delta$ position to the right
and $l'$ means execute next instruction $\#l'$.
If not $\sigma$, then execute next instruction (in same position, say).

A string $x\$c(x)$ is "accepted" — certified as a "yes" instance by an algorithm polynomial in $p(n)$, if it executes $\leq p(|x|)$ instructions and reaches the "accept" instruction. *NP*is a set of recognition problem for which there exists a certificate checking algorithm $\mathcal{A}$ and a polynomial $p(n)$ such that $x$ is a yes instance iff $\exists$ string $c(x)$ with $|c(x)| \leq p(|x|)$ such that $\mathcal{A}$ accepts $x\$c(x)$ while executes $\leq p(|x|)$ instructions.

We now show that *SAT* is *NP-complete* by reducing any *NP*problem to *SAT* . The following is a sketch of the proof:

Let $F(x)$ be a boolean formula *CNF*.

$$F(x) = U(x) \bigwedge S(x) \bigwedge W(x) \bigwedge E(x)$$

Define variables.

$$x_{ij\sigma} = \begin{cases} 1, & \text{at time } i \text{, the } j\text{th position contains } \sigma, 0 \leq i, j \leq p(|x|) \\ 0, & \text{otherwise} \end{cases}$$

$$y_{ijl} = \begin{cases} 1, & \text{at time } i \text{, the } j\text{th position is scanned and } l\text{th instruction is executed} \\ 0, & \text{otherwise} \end{cases}$$

1. The purpose of $U(x)$ is to make sure that at each time $i$, $0 \leq i \leq p(|x|)$, each position of the string contains a *unique* symbol, the head scans a *unique* position within the bounds of the string, and the program executes a *single* statement.

$$U(x) = \Big( \prod_{0 \leq i,j \leq p(|x|), \sigma \neq \sigma'} (\overline{x_{ij\sigma}} + \overline{x_{ij\sigma'}}) \Big) \bigwedge \Big( \prod_{0 \leq i \leq p(|x|), j \neq j' or l \neq l'} (\overline{y_{ijl}} + \overline{y_{ij'l'}}) \Big)$$

$$\bigwedge \Big( \prod_{0 \leq i \leq p(|x|), 1 \leq l \leq |\mathcal{A}|} \overline{y_{i0l}} * \overline{y_{i,p(|x|)+1,l}} \Big)$$

$$\bigwedge \Big( \prod_{0 \leq i \leq p(|x|)} \Big( \Big( \prod_{1 \leq j \leq p(|x|)} \sum_{\sigma \in \sum} x_{ij\sigma} \Big) \bigwedge \prod_{1 \leq j \leq p(|x|), 1 \leq l \leq |\mathcal{A}|} y_{ijl} \Big) \Big)$$

2. The formula $S(x)$ states that the operation of $\mathcal{A}$ *starts* correctly; in other words, at time zero the leftmost $|x| + 1$ symbols in the string spell $x\$$,the read–write head scans the leftmost

symbol of the string, and the first instruction of $\mathcal{A}$ is about to be executed.

$$S(x) = (\prod_{j=1}^{|x|} x_{0jx(j)}) \bigwedge x_{0,|x|+1,\$} \bigwedge y_{011}$$

( Here $x(j)$ stands for the $j$th symbol of the string $x$.)

3. The formula $W(x)$ states that $\mathcal{A}$ *works* right, according to the instructions of the program; $W(x)$ is the conjunction of the formulas $W_{ij\sigma l}$, one for each $0 \leq i \leq p(|x|)$, $1 \leq j \leq p(|x|)$, $\sigma \in \sum, 1 \leq l < |\mathcal{A}|$, such that the $l$th instruction of $\mathcal{A}$ is

$$l : \text{if } \sigma \text{ then } (\sigma'; o; l').$$

The formula $W_{ij\sigma l}$ is defined as follows :

$$W_{ij\sigma l} = (\overline{x_{ij\sigma}} + \overline{y_{ijl}} + x_{i+1,j,\sigma'}) \bigwedge (\overline{x_{ij\sigma}} + \overline{y_{ijl}} + y_{i+1,j+o,l'})$$

$$\bigwedge \prod_{\tau \neq \sigma} ((\overline{x_{ij\tau}} + \overline{y_{ijl}} + x_{i+1,j,\tau}) (\overline{x_{ij\tau}} + \overline{y_{ijl}} + y_{i+1,j,l+1}))$$

This means that whenever $x_{ij\sigma}$ and $y_{ijl}$ are both true, in the next time instant the $x$-and$y$-variables stating that $\mathcal{A}$ made the right move must also be true. For the last instruction of $\mathcal{A}$, we have for each $i, j$ and $\sigma$

$$W_{ij\sigma|\mathcal{A}|} = (\overline{x_{ij\sigma}} + \overline{y_{ij|\mathcal{A}|}} + y_{i+1,j,|\mathcal{A}|})$$

stating that once the algorithm reaches the **accept** instruction, it stays there. Finally, $W(x)$ contains the clauses

$$\prod_{0 \leq i \leq p(|x|), \sigma \in \sum, 1 \leq l \leq |\mathcal{A}|, j \neq j'} (\overline{x_{ij\sigma}} + \overline{y_{ij'l}} + x_{i+1,j,\sigma})$$

meaning that whenever $\mathcal{A}$ is scanning a position different from the $j$th, the symbol of the $j$th position remains unchanged.

4. The last part of $F(x)$ states simply that the operation of $\mathcal{A}$ *ends* correctly, with the program executing the **accept** instruction. It consists of just one clause:

$$E(x) = \sum_{j=1}^{p(|x|)} y_{p(|x|),j,|\mathcal{A}|}$$

This completes the construction of $F(x)$. First let us observe that this construction requires only a polynomial —in $|x|$—amount of time. Hence, to show that the construction is a polynomial transformation from A to $SAT$ , we have only to show that $F(x)$ is satisfiable if and only if $x$ is a *yes* instance of A. ( You may refer to p356-358, [P&S]. )

### Conclusion from Cook's Theorem

*SAT* is a hardest *NP*problem. There are other *NP*problems that are at least as hard as *SAT* and hence hardest in *NP*, all are called *NP-complete* problems.
For a problem to be *NP-complete* :
(1) It is in *NP*,
(2) It is "harder" than an *NP-complete* problem.

### Polynomial Time Reduction

$$\begin{array}{ccccccc}
\textbf{SAT} & \Rightarrow & \textbf{3-SAT} & \Rightarrow & \textbf{MAXclique} & \Rightarrow & \textbf{node cover} \\
\Downarrow & & & & & & \Downarrow \\
\textbf{3-dim matching} & & & & & & \textbf{independent set} \\
\Downarrow & & & & & & \\
\textbf{3 exact cover} & & & & & & \\
\Downarrow & & & & & & \\
\textbf{0-1 knapsack} & \Rightarrow & \textbf{integer knapsack} & & & &
\end{array}$$

*NP-hard*

If a problem is only shown to be harder than an *NP-complete* problem but not in *NP*, then it is called *NP-hard*. The optimization version of each decision problem is *NP-hard* (but with frequent abuse of terminology, is often referred to as *NP-complete* ).

**Lecturer:** Dorit Hochbaum
**Scribe:** Eli Olinick

**Handouts:** The Branch-and-Bound Technique

**Last Class:**

1. Interactive proof for "zero knowledge" certificates.
2. Overview of proof of Cook's Theorem

**Today:**

1. When is IP polynomial - a partial list
2. Coping with intractability: Brach and Bound (B&B)

# 1 When Integer Programming is Polynomial

Although Integer Programming is in general NP-Complete, there are several major classes of IP's for which polynomial solutions are know. The following is a partial list of these classes.

## 1.1 Integer Linear Equations (Diophantine Equations)

Here, we seek integer solutions to the system of linear equations $Ax = b$. If there are $n$ equations and $m$ variables with $n > m$, we can solve this system for an arbitrary basis of n of the variables and the solve for the other variables in terms of the basic ones. Note that the fact that we have no non-negativity constraints allows us to solve the system for an arbitrary basis. Thus, we can't use this method to solve the optimization problem

$$\begin{array}{rl} \max & \sum_{i=1}^{n} c_i x_i \\ \text{subject to} & Ax = b \\ x \geq 0 & \end{array}$$

### 1.2 IP a with fixed number of variables (Lenstra)

Examples:

1. K-Cut Problems

   (a) K-Cut Problem:
   *Instance:* A graph $G = (V, E)$ with edge weights, $w_e\ \forall\ e \in E$, and an integer $K$.
   *Problem:* Partition the nodes in $V$ into $K$ nonempty, disjoint subsets so that the total weight of the cut, all edges $(i, j)$ where $i$ and $j$ are in different subsets, is minimized.
   *Complexity:* $O(n^{k^2})$ if k is fixed, this is a polynomial in the size of the graph.

   (b) K-Cut Problem with specified nodes:
   *Instance:* A graph $G = (V, E)$ with edge weights, $w_e\ \forall\ e \in E)$, an integer $K$ and nodes $v_1, v_2, \ldots v_k$.
   *Problem:* Partition the nodes in $V$ into $K$ nonempty, disjoint subsets $s_1, s_2, \ldots s_k$ such that $v_i \in s_i$ and the total weight of the cut, all edges $(i, j)$ where $i$ and $j$ are in different subsets, is minimized.
   *Complexity:* This problem can be solved in polynomial time with network flow algorithms for $k = 2$ , but for $k \geq 3$, this problem is NP-Complete.

   There is a big difference in complexity between 3-cut and 3-cut with specified $v_1, v_2$ and $v_3$.

2. IP on $n$ variables and $m$ constraints, $n < m$
   The complexity is $O(n^{\min(m,n)})$ which is a polynomial if $m$ or $n$ is fixed. In other words, polynomial for small values of $m$ and $n$.

### 1.3 IP with Totally Unimodular (t.u.) Matrices

max $cx$
s.t. $Ax \leq b$
x integer
Where $A$ is t.u.

Ex. Network Flow Problems

The constraint matrix $A$ is t.u. if all its subdeterminants are -1, 0 or 1. Recall from our discussion of the Ellipsoid method, that $x_B = B^{-1}b = \frac{C^T b}{detB}$ where $C$ = cofactor matrix of $B$. Suppose $A$ is totally unimodular. If $detB \neq 0$, it must be $\pm 1$ which implies that all entries in $x_B$ are integers and extreme point solutions of the LP will also be integer solutions. Thus, the complexity for these problems is polynomial.

### 1.4 Convex, separable IP over t.u. constraints

min $\sum_{i=1}^{n} f_i(x_i)$
s.t. $Ax \geq b$
x integer
Where $A$ is t.u. and $f_i$ are convex

Even if $A$ is t.u., it doesn't follow from the previous discussion that this problem has polynomial complexity because the optimal solution could be an interior point. It is not necessarily a basic solution. Also, the $f_i$ do not have to be linear. We could make them linear, but the resulting number of variables is not a polynomial in the original number of variables.

### 1.5 Matroids, optimization over submodular constraints

Ex. Minimum Spanning Tree

## 2 Coping with NP-Completeness

One method for handling those cases where IP is not polynomial is the use of Branch-And-Bound (B&B) techniques. The remainder of this discussion is concerned with applying an LP-Based B&B algorithm to the ILP given on page 2 of the handout.

If we ignore , or relax, the constraint that $x_1$ and $x_2$ must be integer valued, we would have a LP. This is know as the Linear Programming-relaxation ( LP relax) of the ILP. The feasible region for the LP relax of the ILP is graphed on page 2 of the hand out. Looking at this graph, we can see that there are 25 integer lattice points in the LP relax's feasible region. The objective function $Z = 5x_1 + 4x_2$ is shown with a dotted line and we can see that optimal solution to the LP relax is not one of these integer solutions. So, solving the LP relax in this case did not give us the optimal solution to the ILP. It does, however, gives an upper bound on the optimal solution.

How can we use this information to solve the ILP ?

One approach is to slide the objective function line back until it first touches a feasible integer solution - in this case (6,1). This is idea is incorporated in what are called cutting plane methods. Any inequality which excludes the LP optimal solution, but is satisfied by all feasible integer points is called a cutting plane. Cutting plane methods work by successively adding new constraints which eliminate more of the LP feasible solutions, but not any integer solutions, until the optimal ILP solution is found. Here, for example, we add the objective function as a constraint and require solutions to be on one side of it (i.e. and the constraint $5x_1 + 4x_2 \leq 37$ ). By successively lowering the value of $Z$ (sliding the objective function back) we will eventually find the optimal ILP solution.

A key issue in cutting plane methods is finding "good" planes or planes that will lead us to the optimal solution quickly. In this example, some good cutting planes are the lines that connect (6,1) and (4,3), (0,5) and (4,3) and $x_1 \leq 6$. These lines describe a polytope for which all the extreme points are integer points, which means that we can solve the LP relax to find the optimal ILP solution. This polytope is called the convex hull of the integer feasibility set. "Very good" cutting planes, called facets, are on the convex hull and in some cases they can be found and are quite useful in practice. Unfortunately the problem of finding the convex hull is itself NP-Complete.

Generating cutting planes and facets are part of the "polyhedral approach" to IP. The approach we will be discussing today, however, is the use of LP based Branch-And-Bound (B&B) method.

### LP based B&B

For the example ILP, the solution to the LP relaxation is $\bar{x} = (5.2, 2.8)$ which gives an optimal value of $Z^* = 37.2$. Since relaxing the integer constraint only increases the feasible region, the LP relax gives an upper (lower) bound for the original ILP maximization (minimization) problem. Thus, we know that optimal IP solution will be $\leq 37$.

The idea of B&B is to get bound, in this case we get it from the LP, and then partition the feasible region in to subregions which we will search in turn. Partitioning the feasible region is called branching. Here, for example, we can partition the original feasible region into those points such that $x_1 \leq \lfloor 5.2 \rfloor$ and those such that $x_1 \geq \lceil 5.2 \rceil$. Now, we consider two new LPs:

1. The original ILP, minus the integer constraint, with the added constraint $x_1 \leq 5$. This is the feasible region on the left of the line $x_1 = 5$.

2. The original ILP, minus the integer constraint, with the added constrain $x_1 \geq 6$. This is the feasible region the right of the line $x_1 = 6$.

Before solving these LPs, we record the least upper bound, $Z_u$, and the greatest lower bound, $Z_l$, that we currently have for the ILP- in this case $Z_u = 37$ (from the LP-relax) and $Z_l = 0$ (since we can see that (0,0) is a feasible solution). For the example problem, finding a feasible solution, and an initial lower bound, was easy. In general, however, even finding an initial feasible ILP solution is hard, so we start we a lower bound of -$\infty$. Solving LP 1 gives an optimal solution of $\bar{x} = (5, 2.85)$ and objective function value of 36.5. Solving LP 2 gives $\bar{x} = (6, 1.4)$ and objective function value of 35.6.

The computations of the B&B procedure can be represented graphically as tree. The original LP relax forms the root node which has two branches: one corresponding to $x_1 \leq 5$ and the other for $x_1 \geq 6$. The tree for the entire computation is given at the end of the notes.

So far, we have used a LP relax to get an initial bound on the optimal ILP solution and then branched on $x_1$. We can branch on any noninteger variable, so could have also branched on $x_2 \leq 2, x_2 \geq 3$. For the next step, we will either explore node 2 by branching on $x_2 \leq 2, x_2 \geq 3$, or explore node 3 by branching on $x_2 \leq 1, x_2 \geq 2$. If node 1 leads to an integer solution, it will have an object function value $\leq 36$ (since the LP relaxation is always an upper bound on the ILP optimal solution). Likewise, exploring node 3 will lead to a solution with an object function value of at most 35. Thus, we can safely assume that 36 is an upper bound on the ILP solution and update $Z_u$ accordingly.

There are no hard and fast rules about how to pick the node to explore next, but a reasonable heuristic is to choose the one that has the "most promising" upper bound. In this case, node 2 had a greater upper bound, so we explore it first. So, we add node 4, which corresponds to adding the constraint $x_2 \leq 2$ to LP 1, and node 5 which corresponds to adding the constraint $x_2 >= 3$. Solving the LP for node 4 gives us the solution $\bar{x} = (5, 2)$ with $Z = 33$. Since this is an integer solution, we will not explore node 4 any further. This is know as fathoming the node and is indicated on the tree by two horizontal lines drawn under the node. Also, since this is a feasible integer solution, we can now update $Z_l$ to 33. The LP solved in node 5 gives the solution $\bar{x} = (4.6, 3)$ and Z = 35.

Next, we explore Node 2 since it is now the most promising. Here we branch on $x_2 \leq 1$ and $x_2 \geq 2$. Adding the constraint $x_2 \geq 2$ to LP 2, we can see that resulting feasible region contains no integer points. Thus, we can fathom node 7. The LP for node 6 gives the solution $\bar{x} = (6.25, 1)$ with $Z = 35.25$. At this point, we can update $Z_u$ to 35 since the next integer solution we find will come from following node 5 or 6. In general, we have an opportunity to update our bounds whenever we explore both sons of a node.

Node 6 is explored next by branching on $x_1 \leq 6$ and $x_1 \geq 7$. Note that since we already imposed the constraint $x_1 \leq 6$ when we branched to node 3, we actually add the constraint

$x_1 = 6$ to the LP for node 10. That is, the LP for Node 10 is LP 2 with the added constraints $x_2 \leq 1$ and $x_1 = 6$. This LP gives the solution $\bar{x} = (6, 1)$ and $Z = 34$. Since this an integer solution we fathom the node and update $Z_l$ to 34. For node 11, we add the constraint $x_1 \geq 7$ which makes the LP infeasible. Thus, we fathom node 11 as well.

Now, node 5 is only remaining node that is neither fathomed nor explored. We branch on $x_1 \leq 4$ and $x_1 \geq 5$. Since we have already imposed the constraint $x_1 \leq 5$ when we branched to node 2, thus we add the constraint $x_1 = 5$ for node 9. The resulting region has no feasible points, so node 9 is fathomed. The LP for node 8 gives the solution $\bar{x} = (4, 3.25)$ and $Z = 33$. Since we already have an integer solution with $Z = 33$ at node 4, we fathom node 8 and say that is *dominated* by node 4. At this point, we have seen all the cases in which a node is fathomed:

1. Infeasibility: the subset of the feasible region is empty or in this case the resulting LP relax is infeasible.

2. Integer solution found: the optimal solution for the LP relax is an integer solution and thus a solution to the ILP. Adding more constraints only give us a worse solution.

3. Dominated node: the best $Z$ value that could be found from continuing this branch is not any better than the best $Z$ value for some *known* feasible solution.

Since the entire is tree is fathomed, we stop the B&B procedure and conclude that (6,1) is the optimal solution to the original ILP. Note that we need not carry the procedure out until an optimal ILP solution is found. Instead, we can stop when the gap between the upper and lower bounds is "small enough". For example, after fathoming Node 4, we have an upper bound of 36 and a lower bound of 34. Thus, the absolute error is $\leq 2$ and the relative error, $\frac{Z_u - Z_l}{Zu}$ , is less than 8.333%. If an 8% error were good enough for our application, we could stop the B&B and use $x_1 = 5, x_2 = 2$ as an approximate solution.

Another issue with B&B techniques is storing the information in the computation tree. The tree for our example is fairly small, but it could get quite large for a bigger problem. This is another reason why the procedure may be stopped prior to finding the optimal solution.

**Lecturer:** Dorit Hochbaum
**Scribe:** E. Mahalingam

**Last class**

- LP-based Branch and Bound
  - feasible solution
  - branching (to two subsets) using a heuristic rule
    1. lowest index $x_i$
    2. variable farthest away from an integer
  - bounding by LP
  - fathoming (if for max)
    1. $Z_U \leq Z_L$ { $Z_U$ is the current solution ( non-integer which is an upper bound). $Z_L$ is the current best integer solution. }
    2. Infeasibility
    3. Integer Solution found
- Defined — cutting planes, facets and convex hulls

**Today**

- Definition of $\delta$ approximation.
- Another example of LP-based branch-and-bound ( B&B for short).
- Balas's additive algorithm (another B&B procedure)
- Lagrangian relaxation introduced via a discussion of TSP

A branch and bound algorithm need not be carried out to termination. We can stop at the current best $Z_L$, if it is not too far from the current lower bound. One rule might be to stop B&B when the relative error is "small enough".

relative error $\leq \frac{Z_U - Z_L}{Z_L}$ (assume $Z_L > 0$)

# 1 $\delta$-approximation algorithm.

For a minimization problem, a $\delta$-approximation algorithm is a polynomial algorithm, such that for every instance I , the algorithm delivers a value $Alg(I)$ with $Opt(I) \leq Alg(I) \leq \delta.Opt(I)$, where $Opt(I)$ is the optimal solution value.

i.e.

$$\mathrm{Sup}_I \left\{ \frac{Alg(I)}{Opt(I)} \right\} \leq \delta$$

An algorithm that is a $\delta$-approximation, guarantees relative error $\leq \delta - 1$:

$$\frac{Alg(I) - Opt(I)}{Opt(I)} \leq \delta - 1 \ .$$

For a maximization problem, $Opt(I) \geq Alg(I) \geq \delta.Opt(I)$, $0 \leq \delta \leq 1$.

**Example**

Consider $TSP$ with $\Delta$-inequality (weights on edges satisfying $\forall_{ijk} w_{ij} + w_{jk} \geq w_{ik}$). We have a 2-approximation algorithm.

MST-Algorithm:

**Step 1:** Find a minimum spanning tree (MST) in the graph.

**Step 2:** "Double" the edges of the spanning tree (to get a graph in which each node has even degree). Construct an Eulerian tour.

**Step 3:** Construct a valid tour that visits each node exactly once using "short-cuts".

Now, by the $\Delta$-inequality, the length of the tour is no more than two times the length of the minimum spanning tree (denoted by $|MST|$). Also $|MST| \leq |TSP|$ for $|TSP|$ denoting the length of the optimal tour, because a tour is a spanning tree with one additional edge:

$$\Rightarrow |MST| \leq |TSP| \leq 2|MST| \ .$$

Now,

$$|\text{MST-Algorithm}(I)| \leq 2|MST| \leq 2|TSP| = 2OPT(I)$$

$\Rightarrow$ MST-Algorithm is a 2-approximation algorithm.

Some asides:

- Devise a more clever scheme for TSP with $\Delta$-inequality, which uses the fact that some nodes are of even degree and doubles fewer of the edges to give a better $\delta$-approximation.
- Peek back at the B&B example on Page 3 of the handout ("The Branch-and-Bound Technique") from Lecture 10. We went through the example, until all nodes expanded were fathomed, and we got the best integer solution.
- Dual Simplex method for B&B, so as to reduce the work of the Simplex Method for solving the LP's at the nodes of the B&B tree.

## 2 Balas's additive algorithm

Balas's additive algorithm is a B&B algorithm that does not use LP. It considers binary ILP in the following standard form.

$$min \qquad \sum_{j=1}^{n} c_j x_j \qquad\qquad s.t. \tag{1}$$

$$\sum_{j=1}^{n} a_{ij} x_j \geq b_i \qquad i = 1, \ldots, m \qquad x_j \in \{0, 1\} \tag{2}$$

$$0 \leq c_1 \leq c_2 \leq \cdots \leq c_n \tag{3}$$

All the coefficients $c_i$ are non-negative and in increasing order.

**Theorem 2.1** *Any ILP can be reduced to the standard form above.*

**Proof:**

1. Given a binary IP, to achieve (3), if $c_j < 0$, define a new variable $x_{j'} = 1 - x_j$. Now $c_{j'} = -c_j$. Re-index the variables so that the coefficients are sorted.
2. To achieve (2), if the inequality is $\leq$ , multiply the inequality by -1 to get it in hte desired $\geq$ form.
3. If the IP is not in binary variables, then any variable $x_j$ can be coded in $\lceil \log u_j \rceil$ binary variables, if there are explicit upper bounds $u_j \geq x_j \geq 0$. Define $\lceil \log u_j \rceil$ new variables, such that

   $$x_j = \sum_{k=0}^{\lceil \log u_j \rceil} 2^k x_{kj}$$

   If the $u_j$ are not given explicitly, we still know that it exists as a polynomial function of the input data (refer to Lemma 8.5 [PS]).

■

**Comments**

One advantage of Balas' algorithm is that it uses only additions and subtractions.

Finally, we worked out rules for branching, bounding, and infeasibility checking using the example from Page 8 of the handout in class.

**Lecturer:** Dorit Hochbaum
**Scribe:** Angela Chuang

**Agenda**

Last time, we discussed the definition of a $\delta$-approximation algorithm, and gave an example of a 2-approximation algorithm for the Traveling Salesman Problem satisfying the triangle inequality restriction. Further discussion was made on linear programming relaxation based Branch and Bound, and Balas' Additive Algorithm (another B&B algorithm) was introduced, along with an example.

Today we will cover success factors of B&B, and begin discussion on Lagrangian relaxation, which will be introduced via three ways of formulating TSP:

1. Assignment formulation $\rightarrow$ leads to B&B algorithm
2. Tucker's formulation $\rightarrow$ a compact formulation
3. 1-tree $\rightarrow$ leads to Lagrangian relaxation and subgradient method

# 1 Success Factors of Branch and Bound

## 1.1 Formulation of the integer programming problem

**Example**: *The Fixed Cost Plant Location Problem.*

This problem is related to the $k$-median problem. However, you can choose as many nodes to be plants as you want, and each plant costs something.

Let $f_j$ be the cost of having a plant at location $j$, and let $d_{ij}$ be the cost of transportation from location $i$ to $j$.

$$y_j = \begin{cases} 1, & \text{if there is a plant at location } j \\ 0, & \text{otherwise} \end{cases}$$

$$x_{ij} = \begin{cases} 1, & \text{if customer } i \text{ is assigned to facility } j \\ 0, & \text{otherwise} \end{cases}$$

The objective is to minimize the sum of the fixed costs and variable costs.

The *aggregate formulation* follows:

$$
\begin{array}{lll}
\min & \sum_{j=1}^{n} y_j f_j \ + \ \sum_{j=1}^{n} \sum_{i=1}^{n} x_{ij} d_{ij} & \\
\text{subject to} & \sum_{j=1}^{n} x_{ij} = 1, & \forall i = 1, \ldots, n \\
& \sum_{i=1}^{n} x_{ij} \leq n y_j, & \forall j = 1, \ldots, n \\
& x_{ij} \in \{0,1\} & \\
& y_j \in \{0,1\} &
\end{array}
$$

The *disaggregate formulation* follows:

$$
\begin{array}{lll}
\min & \sum_{j=1}^{n} y_j f_j \ + \ \sum_{j=1}^{n} \sum_{i=1}^{n} x_{ij} d_{ij} & \\
\text{subject to} & \sum_{j=1}^{n} x_{ij} = 1, & \forall i = 1, \ldots, n \\
& x_{ij} \leq y_j, & \forall j = 1, \ldots, n \ \ \forall i = 1, \ldots, n \\
& x_{ij} \in \{0,1\} & \\
& y_j \in \{0,1\} &
\end{array}
$$

Using a good formulation of a problem can speed up fathoming in B&B. Note that the disaggregate formulation has more constraints than the aggregate formulation. Hence, the former will have a smaller feasible set. Because the search regions are tighter after the integer constraints are relaxed, the disaggregate formulation is a better formulation to use in Branch and Bound.

### 1.2 Bound quality

For example, the bound from LP relaxation is typically better than the bound from Balas' Additive Algorithm.

### 1.3 Number of nodes

The number of nodes in the Branch and Bound tree is unpredictable in advance.

### 1.4 Branching rule

### 1.5 Luck!

Much depends on luck.

## 2 TSP Formulations

### 2.1 Assignment Formulation

Let

$$
x_{ij} = \begin{cases} 1, & \text{if edge (i,j) is chosen to be in the optimum tour} \\ 0, & \text{otherwise} \end{cases}
$$

The Assignment Formulation of TSP follows:

$$
\begin{array}{rll}
\min & \sum_{j=1}^{n}\sum_{i=1}^{n} c_{ij}x_{ij} & \\
\text{subject to} & \sum_{j=1}^{n} x_{ij} = 1, & \forall i = 1,\ldots,n \\
& \sum_{i=1}^{n} x_{ij} = 1, & \forall j = 1,\ldots,n \\
\text{(subtour elimination constraints)} & \sum_{i,j\in S} x_{ij} \le |S|-1, & \forall subtours\ S \subsetneq V \\
& x_{ij} \in \{0,1\} &
\end{array}
$$

This formulation can be used in "subtour elimination relaxation B&B":
First relax the subtour elimination constraints. The remaining problem is Assignment and solvable in polynomial time. Exercise Branch and Bound where branching is done on each edge eliminated from a chosen subtour. At each node, the value of the Assignment solution serves as a lower bound on the optimal value of the TSP.

## 2.2 Tucker's Formulation

Tucker's Formulation follows:

$$
\begin{array}{rll}
\min & \sum_{j=0}^{n}\sum_{i=0}^{n} c_{ij}x_{ij} & \\
\text{subject to} & \sum_{j=0}^{n} x_{ij} = 1, & \forall i = 1,\ldots,n \\
& \sum_{i=0}^{n} x_{ij} = 1, & \forall j = 0,\ldots,n \\
\text{(subtour elimination constraints)} & u_i - u_j + nx_{ij} \le n-1, & 1 \le i \ne j \le n \\
& x_{ij} \in \{0,1\} &
\end{array}
$$

Tucker's Formulation is a mixed integer programming formulation, where some variables are required to be integer and some are not. It is similar to the previous Assignment formulation of TSP, except that the subtour elimination constraints are replaced by the following:

$$u_i - u_j + nx_{ij} \le n-1, \quad 1 \le i \ne j \le n \ ,$$

where $u$'s are some constants [Reference: p.309 (PS)].

Note that there are $O(n^2)$ constraints here, compared to the $2^{|V|}$ subtour elimination constraints found in the Assignment Formulation. Hence, Tucker's Formulation is more compact in that it has a polynomial number of constraints. Nevertheless, it is basically useless computationally speaking, and is another example of an instance in which reducing the number of constraints does not necessarily lead to a better algorithm.

- *To see why these constraints eliminate a subtour:*
  Consider a subtour $\{i_1, i_2, ..., i_k\}$. Then,

$$
\begin{array}{rcl}
u_{i_1} - u_{i_2} + n & \le & n-1 \\
u_{i_2} - u_{i_3} + n & \le & n-1 \\
& \vdots & \\
u_{i_{k-1}} - u_{i_k} + n & \le & n-1 \\
u_{i_k} - u_{i_1} + n & \le & n-1
\end{array}
$$

Adding the left hand sides of the above inequalities together, and adding the right hand sides together, we see that the above constraints imply $kn \leq k(n-1)$ or $n \leq n-1$ which is always false. So, all subtours violate the subtour elimination constraints.

- *To see why a tour satisfies these constraints:*
  Let $\{i_0, i_1, i_2, ..., i_n, i_0\}$ be a tour. Assign $u_{i_0} = 0$ , $u_{i_1} = 1$ , $u_{i_2} = 2$ , ... , $u_{i_n} = n$.
  If $x_{ij} = 0$, then $u_i - u_j \leq n-1, \quad 1 \leq i \neq j \leq n$ which is always true.
  If $x_{ij} = 1$, then $u_i - u_j + n \leq n-1$, which is true since $u_i - u_j = -1$ for i and j successive cities in the tour.

**Lecturer:** Dorit Hochbaum
**Scribe:** Tianxiong Xue

**Last class**

- Success factors in Branch & Bound
- Lagrange Relaxation, introduced via TSP
  3 formulations:
  1. Assignment (leads to Branch & Bound [not LP-based])
  2. Tucker's formulation (assignment + a poly number of sub-tour elimination constraints)
  3. 1-tree formulation (to be discussed this lecture)

**Today**

1. 3. 1-tree formulation (Held & Karp)
   Lagrange relaxation & subgradient method
2. Incorporating subgradient method in Branch & Bound
3. 3/2 - approx for TSP with $\Delta$-inequality

One way of formulating a tour is as a spanning tree + 1 edge
Constraints: every node of degree 2

$$\begin{aligned} \sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} &= 2, \quad i = 1, \ldots, n-1 \\ \sum_{1 \leq i < j \leq n} x_{ij} &= n \end{aligned}$$

There is no need to specify that node $n$ is of degree 2 because of the following lemma from graph theory: *Given a graph with nodes of degrees $d_1, d_2, \ldots, d_n$, then, $m = |E| = \sum_{i=1}^n d_i/2$.*

Let node 1 be "special".
1-tree is a spanning tree on nodes $2, \ldots, n (V - \{1\})$ to which 2 edges adjacent to "1" are added.
*Problem:* Find minimum (edge) weight 1-tree.
Solvable in polynomial time: Find MST ($O(m \log n)$) on $\{2, \ldots, n\}$. Add 2 cheapest edges adjacent to 1 ($O(n)$).
Total complexity: $O(m \log n)$
In order to guarantee that the 1-tree is a tour, we require that there is no cycle in $\{2, \ldots, n\}$ (subtour elimination).

TSP* $= \min \sum_{1\le i<j\le n} c_{ij}x_{ij}$
subject to:

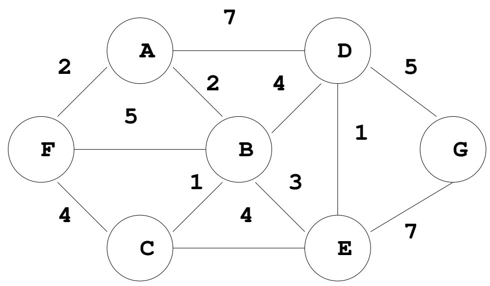
$$\begin{aligned}
\sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} &= 2, i = 2,\ldots,n-1 \ \ (i)\\
\sum_{i<j, i,j\in S} x_{ij} &\le |S|-1, \ \ (ii)\\
\sum_{j} x_{ij} &= 2 \ \ (iii)\\
\sum_{1\le i<j\le n} x_{ij} &= n, \ \ (iv)\\
x_{ij} &\in (0,1), i<j, \ \ (v)
\end{aligned}$$

1-tree optimal value $\le$ TSP optimal value.

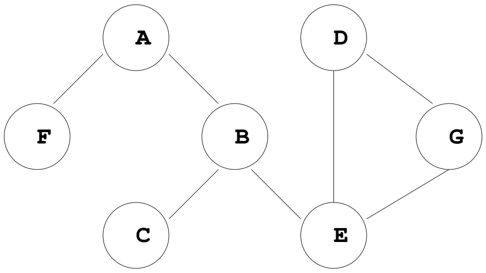
Lagrange relaxation, relax (i):
$L(\vec{\pi}) = \min \sum_{1\le i<j\le n} c_{ij}x_{ij} + \sum_{i=2}^{n-1} \pi_i(2 - [\sum_{j>i} x_{ij} + \sum_{j<i} x_{ji}])$
subject to: (ii) - (v)
$\pi_i$ unrestricted, $\sum_{j>i} x_{ij} + \sum_{j<i} x_{ji} = d_i$, the degree of node $i$.
$L(\vec{\pi}) \le TSP*$
Rewrite $L(\vec{\pi})$ as:
$L(\vec{\pi}) = \min \sum_{1\le i<j\le n} c_{ij}x_{ij} - 2\sum_{i=2}^{n-1}\pi_i - \sum_{i=2}^{n-1}\pi_i d_i$
subject to: (ii) - (v)
i.e.,

$$\begin{aligned}
L(\vec{\pi}) = 2\sum_{i=2}^{n-1}\pi_i + \min \sum_{1\le i<j\le n} (c_{ij} - \pi_i - \pi_j)x_{ij}\\
= 2\sum_{i=2}^{n-1}\pi_i + \min\{\overline{c_{ij}}, x_{ij}\}
\end{aligned}$$

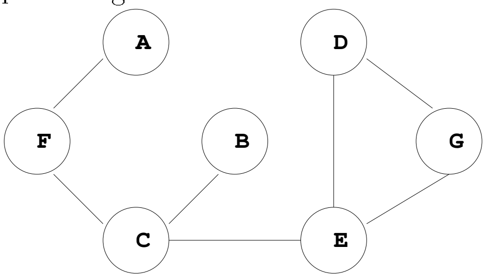
subject to: (ii) - (v)
*Example:*




Edges not present have cost = 20.
One tour found by inspection, with total weight 28, is: GDBAFCEG
Let node 1 = $G$:




1-tree = 21, $L((A, B, C, E, F) = (0, 0, 0, 0, 0)) = 21$
$L(\vec{\pi}*) = \max_{\pi} L(\vec{\pi}$. The search for $\pi*$ is done using the subgradient method:
$\pi_B \searrow, \pi_C \nearrow, \pi_E \searrow, \pi_F \nearrow, \pi_A$ unchanged.




1-tree = 23.
$\pi_A \nearrow, \pi_B \nearrow, \pi_C \searrow, \pi_E \searrow$
Check that the following set of multipliers is optimal:
$\pi_A = 2, \pi_B = -1, \pi_C = 0, \pi_D = -4, \pi_E = -4, \pi_F = 2$

**Lecturer:** Dorit Hochbaum
**Scribe:** Paul Farias

**Agenda**

In the last class the subgradient method of solving the relaxed Lagrangian problem was discussed. The subgradient method is needed because the objective function of the relaxed problem is generally not differentiable. Thus a gradient function is not generally available.

Two examples were shown to demonstrate the subgradient method. The first example was the fixed plant location problem and the second was the Set Union Knapsack first relaxation.

Today the Set Union Knapsack second relaxation will be covered followed by a discussion of the duality gap and the integrality property.

# 1 Set Union Knapsack Relaxation 2

## 1.1 Formulation of the integer programming problem

**Example**: *The Set Union Knapsack second constraint relaxed.*
Let S be the set containing the collections of elements (or baskets of goods) that may be chosen. Let $|S| = P$. Let J be the set of items that comprise the elements (the baskets of goods) of S. Let $|J| = q$.

For $1 \leq i \leq P$ let $S_i$ be the $i^{th}$ element of S. For $1 \leq i \leq P$ let

$$y_i = \begin{cases} 1, & \text{if all elements of } S_i \text{ are selected} \\ 0, & \text{otherwise.} \end{cases}$$

For $1 \leq j \leq q$ let

$$x_j = \begin{cases} 1, & \text{if element j is selected} \\ 0, & \text{otherwise.} \end{cases}$$

Now the Set Union Knapsack problem is formulated as follows:

$$\begin{aligned} z_D(u) = max \quad & \textstyle\sum_{i=1}^{P} y_i + u[b + \sum_{j=1}^{q} x_j] \\ s.t. \quad For\ 1 \leq i \leq P\ and\ j \in S_i \quad y_i \ & \leq \ x_j \\ x_i, y_i \in \ & \{0,1\} \end{aligned}$$

The constraint matrix of the above problem is totally unimodular (TU). Which means the subdeterminants are all -1 , 0 or 1. When this occurs the problem can be solved with a min cut-max flow algorithm as is done with the selection problem.

### 1.2 Selection Problem

The selection problem is formulated as follows:

$$\begin{aligned} max\ [\sum_{i=1}^{P} B_i y_i \ - \ \sum_{j=1}^{q} c_j x_j] \\ st \quad For\ 1 \leq i \leq P,\ \ j \in S_i \ \ y_i \ \leq \ x_j \\ x_j,\ y_i \in \{0,1\} \end{aligned}$$

In this formulation the $y_i$ are a collection of sets (or baskets) and the $x_i$ are a collection of objects in the baskets. Some objects may be in more than one basket. We select a basket by choosing all of it's elements. The cost of selecting element j is $c_j$. We gain a benefit of $B_i$ if all the objects of set i are chosen. To solve the selection problem construct a bipartite graph with sets on one side and elements on the other. Add a node S with an arc to each of the sets, $y_j$, with a cost of $B_j$ and a node t which receives an arc from each of the elements, $x_j$, with a cost of $c_j$. For each set, $y_i$, add an arc, with infinite cost, to each of its elements.

Any finite S-T cut in this graph corresponds to a feasible solution of the selection problem; give the variables corresponding to nodes in the S section of the cut the value 1. Now $\sum_{i=1}^{P} B_i + max_{S(a\ selection)}[-\sum_{\notin S} B_i - \sum_{j\in S} C_j]$ = B − min $\sum_{i\notin S} B_i + \sum_{j\in S} C_j$ = B − min cut in the bipartite graph. Thus the selection problem reduces to solving a minimum cut.

However, we really need to solve $z_D(u*) = min_u$. This can be done by solving the min cut-max flow problem parametrically. An algorithm to do this was developed by Gallo et all in 1991. This problem can be solved in the same time required to solve one max flow problem, O($n^3$). In our problem $n = p + q$. We are lucky because generally $z_D(u)$ cannot be solved in polynomial time. Another example is the Held-Karp 1-tree bound. This can be solved by applying the ellipsoid method on an LP with an exponential number of constraints: for $S \subset \{2,\ldots,n\}$ $\sum_{i,j\in S, i<j} x_{ij} \ <= |S|-1$. Which can be solved, in polynomial time, using a separation algorithm. It has been shown by Shmoys and Williamson (Info Processing Letters 92) that the Held & Karp lower bound is $\geq \frac{2}{3}$ TSP*.

## 2 Duality Gap

### 2.1 Assignment Formulation

Let z* be the optimal solution to an ILP and $z_D(\pi^*)$ be the best lower bound for a Lagrangian relaxation of the ILP. Then $z^* - z_D(\pi^*)$ is called the Duality gap for the Lagrangian relaxation. Suppose for some graph G, G has a TSP*= 100. Further suppose the best 1-tree relaxation lower bound is at least 66 2/3 then the duality gap for this relaxation is no more than 33 1/3. If an optimal solution to the Lagrangian relaxation is integer, whether specified or not, the Lagrangian relaxation is said to have the integrality property. Examples include the 1-tree relaxation for TSP, set-union knapsack (relaxation 2) and the fixed cost plant location problem.

**THEOREM:**

When the integrality property holds then the duality gap = Lagrangian relaxation solution − LP relaxation solution.

Proof: See page 13 of the Fischer paper.

**Lecturer:** Dorit Hochbaum
**Scribe:** Paul Farias

**Agenda**

During the last lecture we discussed why the Dual Ascent method is better then the subgradient method for the fixed plant location problem. Prof. Hochbaum then made some comments regarding Beasley's paper on Lagrangian relaxation. Next, we were shown a 3/2 approximation for the TSP – the algorithm used an MST plus a matching on the odd degree nodes of the MST. The lecture was completed by showing that getting a better than 3/2 approximation is an NP-complete problem

Today a Fully Polynomial Approximation Scheme (FPAS) for the 0/1 knapsack problem will be presented. Then it will be shown that a FPAS for a strongly NP-complete problem is impossible.

# 1 FPAS for the 0/1 knapsack problem

For the details of how to develop a Fully Polynomial Approximation Scheme for the 0/1 knapsack problem see the handout "A Fully Polynomial Approximation Scheme for the Knapsack Problem". Equation (1.2) of the handout should read:

$\mathrm{P}^* = \sum_{j\in S^*} p_j \;\geq\; \sum_{j\in S(k)} p_j \;\geq\; \sum_{j\in S(k)} k\lfloor p_j/k\rfloor \;\geq\; \sum_{j\in S^*} k\lfloor p_j/k\rfloor \;\geq\; k\; \sum_{j\in S^*}\lfloor \frac{p_j}{k}\rfloor \;\geq\; \sum_{j\in S^*}(p_j - k) \;\geq\; \sum_{j\in S^*} p_j - k|S^*|$.

The best scaling factor k to use is the GCD of $\{\, p_j \mid j \in \{1,\ldots,N\}\,\}$.

Section 3 of the handout describes a greedy algorithm for the 0/1 knapsack problem that adds the item with the largest "profit ratio" $(\frac{p_j}{a_j})$ as long as it fits.

**Lemma:**
Greedy is a 2-approximation algorithm (OPT $\leq$ 2 Greedy).

The handout provides a reference to a proof of this lemma.

# 2 Greedy Algorithm

Section 3 of the handout describes a greedy algorithm for solving the knapsack problem.

**Lemma:** Greedy is a 2-approximation algorithm.

## 3 Strongly NP-complete Problems

**Definition:**

The running time of an algorithm for a strongly NP-complete problem is independent of the size of the numbers in the input. Alternatively, strongly NP-complete problems remain NP-complete even if the input is given in unary.

An NP-complete problem that is not strongly NP-complete is called weakly NP complete.

An example of a weakly NP-complete problem is the 0/1 Knapsack problem which has a running time of O(nB).

**THEOREM:** (Garey and Johnson)

If there exists a FPAS scheme for a strongly NP-complete problem then NP=P.

**Lecturer:** Dorit Hochbaum
**Scribe:** Derek Bennett

**Review of Last Class:**

Last class focused on the idea of a fully polynomial approximation scheme (FPAS). This is in contrast to the dynamic programming algorithm for the Knapsack problem we saw before, where the running time was pseudopolynomial. Specifically, we discussed:

1. FPAS for 0/1 KNAPSACK, where the running time was polynomial in $n$ and $\frac{1}{\epsilon}$.
2. Relationship of FPAS to the theory on NP-completeness
3. Concept of strongly NP-complete problems

We stated a theorem, by Garey & Johnson: If there is a FPAS for a strongly NP-complete problem then $\mathcal{NP} = \mathcal{P}$. Running times for strongly NP-complete problems are independent of the size of the numbers in the input. Alternatively, a strongly NP-complete problem remains NP-complete even if the input is represented in unary (e.g. K-CENTER). This will not be true for a weakly NP-complete problem such as 0/1 KNAPSACK.

**Today:**

The problem discussed today will be a scheduling problem called MINIMUM MAKESPAN, which is a strongly NP-complete problem.

1. Problem description and formulation
2. Discussion of LIST and LIST DECREASING heuristics
3. Relationship to BINPACKING, and the use of dual approximation schemes

# 1 Problem description and formulation

Problems that arise frequently in planning and scheduling provide us with a rich collection of problems and formulations to study. There is a wide range of difficulty in these problems, that can occur due to deadlines, priority jobs, precedence constraints, and resource constraints. Some of you may be familiar with PERT and CPM tools used in project planning situations.

Unfortunately, many of the problems in this area are known to be NP-complete. Today, we will look at an example of such a problem, the MINIMUM MAKESPAN problem.

**Problem Formulation:**

Suppose that we have $m$ identical machines. There are $n$ jobs, each with a processing time of $p_j, j = 1, \ldots, n$, which must be processed on these machines. We are given that $n > m$, and no preemption of jobs is allowed, which means that each job must be processed the same machine from start to finish. The idea is to partition the $n$ jobs into $m$ sets and create a "schedule". Our objective in the schedule is to minimize the maximum machine makespan. In other words, minimize the latest finish time of the last job completed. The decision version of this problem is as follows:

*Is there a schedule (partition of $n$ jobs into $m$ subsets, $S_i$) such that:*

$$\sum_{j \epsilon S_i} p_j \leq K \quad \forall S_i \ ?$$

This problem is known to be strongly NP-complete. A reduction from BINPACKING is possible.

# 2 Heuristics for solving the problem

## 2.1 An example using the LIST heuristic

We will illustrate the list heuristic with the following example. Suppose we have 6 machines, and the following list of jobs: 5 jobs of length 5, 5 jobs of length 1, and 1 job of length 6. The list heuristic works as follows:

**List heuristic :** Given a list of jobs, take each job one at at time, and place it in the machine with the current earliest finish time.

Applying this rule results in a schedule with a makespan of 11, using the list as given above. The optimal schedule has a makespan of 6: put the job of length 6 on one machine, and the other five machines have one job of length 5, and one of length 1. So the list heuristic gives us a ratio of $\frac{11}{6}$. Although this isn't great, one advantage of the list heuristic is that it is an *on-line* algorithm. That is, it doesn't look ahead in the data before choosing an allocation. There is a theorem (Graham, 1966) providing a bound on the error of the list heuristic:

**Theorem 2.1** *The list heuristic is a $(2 - \frac{1}{m})$-approximation algorithm.*

**Proof:** Let $\mathcal{OPT}$ be the optimal solution value, and $\mathcal{L}$ be the makespan returned by the LIST heuristic. First, observe that the best makespan that could be achieved with preemption is the sum of the job lengths, divided by $m$, the number of machines. The optimal solution must be greater than that solution obtained with preemption, and must also be greater than the longest job. Thus,

$$\mathcal{OPT} \geq \max\{\frac{1}{m}\sum_{j=1}^{n} p_j, \max_j \ p_j\}$$

Now, let $t$ be the processing time of the last job on the list. Then,

$$\mathcal{L} - t \leq \frac{1}{m}(\sum_{j=1}^{n} p_j - t)$$

$$\Rightarrow m(\mathcal{L} - t) \leq m \cdot \mathcal{OPT} - t$$

$$\Rightarrow \mathcal{L} \leq \frac{m \cdot \mathcal{OPT} + (m-1)t}{m}$$

Since $t \leq \mathcal{OPT}$, then $\mathcal{L} \leq \frac{(2m-1)\cdot\mathcal{OPT}}{m} = (2 - \frac{1}{m})\mathcal{OPT}$. Thus, the heuristic is a $(2 - \frac{1}{m})$-approximation. ■

## 2.2 An improvement: the LIST DECREASING heuristic

To improve upon the list heuristic, we can sort the list in nonincreasing order prior to assigning jobs to machines. This gives us the list decreasing heuristic:

**List Decreasing heuristic :** Sort list of jobs in nonincreasing order, then apply list heuristic.

The example in class had 6 machines, and a list of 13 jobs. The times for the jobs,in nonincreasing order are: 11,11,10,10,9,9,8,8,7,7,6,6,6. When we applied the heuristic, we obtained a makespan of 23. The optimal value is actually 18, so our ratio for the heuristic is $\frac{23}{18}$.

Graham has another theorem, stating the quality of solution obtained with this heuristic:

**Theorem 2.2** *List Decreasing is a* $(\frac{4}{3} - \frac{1}{3m})$*-approximation.*

**Proof:** We will show that either this magnitude of error is valid, or the heuristic is optimal. Our jobs are sorted so that $p_1 \geq p_2 \geq \ldots \geq p_n$. Again, let $\mathcal{OPT}$ be the optimal makespan, and $\mathcal{LD}$ be the makespan returned by the heuristic. Suppose that the above theorem is false. This implies that $\frac{4}{3} - \frac{1}{3m} < \frac{\mathcal{LD}}{\mathcal{OPT}}$ for some problem instance. Since list decreasing is a list heuristic, theorem 1 applies, and $\frac{\mathcal{LD}}{\mathcal{OPT}} \leq 1 + \frac{(m-1)t}{m\cdot\mathcal{OPT}}$. For list decreasing, $t = p_n$. Then,

$$\frac{4}{3} - \frac{1}{3m} < 1 + \frac{(m-1)p_n}{m \cdot \mathcal{OPT}}$$

After some algebra, we discover that $\mathcal{OPT} < 3p_n$, which implies that at most two jobs are assigned to each machine, and $n \leq 2m$. If $n < 2m$, then add $2m - n$ jobs of length 0, for $2m$ total jobs. Then, we claim that for $2m$ jobs, list decreasing is an optimal heuristic. Consider this solution: simply assign the first $m$ jobs, one to each machine. Then, assign the remaining $m$ jobs to the machines, starting with the last machine assigned a job in the first pass, and ending with the first machine. We can show that this produces an optimal solution:

Compare the "pairing up" solution(described above) produced to an optimal solution. Suppose that these solutions differ first at the $i$th machine, where job $i$ is matched with job $2m - i + 1$ in the "pairing up" solution, but matched with job $j$ in the optimal solution. Likewise, machine $k$, $k > i$, is paired up with machine $j$ in the heuristic solution, but paired up with $2m - i + 1$ in the optimal solution. In summary:

| Optimal: | | List Decreasing: |
|---|---|---|
| $p_k + p_{2m-i+1}$ | $\rightarrow$ | $p_k + p_j$ |
| $p_i + p_j$ | $\rightarrow$ | $p_i + p_{2m-i+1}$ |

Then,

$p_k + p_j \leq p_i + p_j$ because $p_k \leq p_i$

$p_i + p_{2m-i+1} \leq p_i + p_j$ because $p_i \leq p_{2m-i+1}$

If $p_i + p_j$ were the optimal makespan length, then the pairing up solution delivered by the heuristic would improve the solution, by the above inequalities. However, this would contradict our initial assumption that the optimal solution differs at the $i$th position. If $p_i + p_j$ is less than the optimal makespan length, then the heuristic has simply delivered a different optimal solution.

There may be instances where the list decreasing solution is different than the "pairing up" solution given above. However, we can show that the list decreasing solution is at least as good as "pairing up". For example, find a job $i$ in the list decreasing assignment that is in a machine by itself (which means another machine contains three jobs). Now, we know that $\mathcal{LD} - p_{2m} \leq p_i$, because otherwise, job $2m$ would be on the same machine as job $i$. Using this inequality, and the fact that $p_{2m} \leq p_{2m-i+1}$, we have:

$$\mathcal{LD} \leq p_i + p_{2m} \leq p_i + p_{2m-i+1} \leq \text{Pairing Up Makespan} .$$

Now, since list decreasing is at least as good as "pairing up", then for $2m$ machines, the list decreasing heuristic is optimal.

We have reached a contradiction, since we know the list decreasing does not always produce an optimal solution. So, this heuristic must be a $(\frac{4}{3} - \frac{1}{3m})$-approximation. ∎

# 3 Relationship to BINPACKING, and Dual Approximation Schemes

A strong relationship exists between the MAKESPAN problem and the BINPACKING problem: suppose we knew the optimal makespan, call it $D$. This quantity becomes the size of the bins, into which we must pack n items of varying sizes $p_j$. The number of bins used must be at most $m$, the number of available machines. If the value of $D$ is unknown, it can first be guessed as the midpoint of a known interval. Then, if more than $m$ bins were used, $D$ must lie in the larger half of the interval. If less than $m$ bins used, then the smaller half of the interval must contain $D$. This procedure can be repeated, leading to a binary search approach.

From above, an optimal solution to the BINPACKING problem leads to an optimal solution to the MINIMUM MAKESPAN problem. The complexity of this algorithm for the makespan problem is the number of calls times the complexity of the BINPACKING problem. How many calls are necessary? Consider this interval which contains $D$:

$$[\max\{\frac{1}{m}\sum_{j=1}^{n} p_j, \max_j \ p_j\}, \frac{1}{m}\sum_{j=1}^{n} p_j + \min_j \ p_j]$$

The upper limit of this interval is an upper bound on the optimal solution, as it is achieved by the list heuristic. So, the number of calls is at most logarithmic in the length of the above interval, and hence, polynomial. Unfortunately, BINPACKING is also known to be NP-complete. So,instead of solving it optimally, we solve it with a *dual approximation scheme*.

If one uses a dual algorithm for linear programming, primal feasibility is often violated in choosing the solutions to the dual problem. Likewise, we will "violate" feasibility by allowing our bins to be slightly larger than $D$. Basically, our dual approximation solution will have two properties:

1. Solution is superoptimal. Number of bins used $\leq$ optimal number of bins

2. There is an $\epsilon$ violation of feasibility: we will allow bins of size $\leq (1+\epsilon)D$.

This approximation scheme will be the subject of the next class, where the concept of scaling and grouping will be introduced.

**Lecturer:** Dorit Hochbaum
**Scribe:** Ilya Vinarsky

Last class, we discussed a $(2-\frac{1}{m})$- and a $(\frac{4}{3}-\frac{1}{3m})$- approximation algorithm to the makespan problem due to Graham, which were some of the first nontrivial approximation algorithms invented. We also mentioned the dual relationship between the makespan problem and the bin packing problem. Today, we present a polynomial approximation scheme for the makespan problem that makes use of scaling and grouping, and introduce the powers of graph approach to bottleneck problems.

# 1 Polynomial Approximation Scheme to Makespan

Consider two similar problems: makespan and bin packing. In the makespan problem, we are to pack several objects of given size into a given number of bins, and we have to minimize the bins' maximum necessary capacity. The original formulation had to do with scheduling jobs of different lengths to run on a given number of machines so that the whole system will finish as early is possible; these formulations are equivalent. In the bin packing problem, the number of bins is variable, but their capacities are constant. The two problems are dual to each other. Both are invariant to scaling, that is, feasibility, optimality etc. do not depend on the units of measurement. Both a"re known to be strongly NP-hard.

Suppose we have a *superoptimal* solution to the bin packing problem, that is, an instance that is known to use at most as many bins as the optimal, but may "overhang" a little bit, by a factor of $\epsilon$, making it infeasible. Then this solution can be used to approximate the corresponding makespan problem the following way:

Let $L = \max\{\max_j t_j, \frac{1}{m}\Sigma_i t_i\}$.
Guess a median $d \epsilon [L, 2L]$.
Find an $\epsilon$-dual approximation for packing $m$ bins of size $d$.
If less than $m$ bins were used, search the smaller half of the interval.
If $m$ bins proved to be too few, search the larger half of the interval.
Repeat until length of the interval is less than 1.

We are looking for a scheme to find $\epsilon$-approximations to bin-packing in time polynomial in $n$ (number of items) for each given $\epsilon$. Our algorithm will run as follows: 1. Pack all items larger than $\epsilon$ (assuming bin sizes are normalized to 1). 2. Pack the remaining items.

In stage 1, no more than $\frac{1}{\epsilon}$ items can go in a single bin. The problem would be easier if we had a fixed number of possible sizes because then we could use dynamic programming. Instead, we use a technique called *scaling and grouping*. Divide the interval $(0, \epsilon)$ into subintervals of size $\epsilon^2$. Divide every item's size by $\frac{1}{\epsilon}$, rounding down to the nearest multiple of $\epsilon^2$. Then solve the problem optimally using dynamic programming. The reason we are rounding down is to ensure superoptimality, since if item sizes are reduced or kept the same, it would require fewer or as many bins to accommodate them than before.

A feasible configuration is a collection of items that can go feasibly into one bin. There are $s = \frac{1}{\epsilon^2}$ possible sizes, and for each size, at most $\frac{1}{\epsilon}$ can go into a bin.
So, a (loose) upper bound on the total number of feasible configurations is $\lfloor\frac{1}{\epsilon}\rfloor^{\lfloor\frac{1}{\epsilon^2}\rfloor}$. For $\epsilon$ fixed, this number is fixed.

For the purposes of dynamic programming, let $BINS(n_1, n_2 \ldots n_s)$ be the smallest number of bins necessary to accomodate $n_1$ items of size $\epsilon^2$, $n_2$ items of size $2\epsilon^2$ etc. Obviously, $BINS(0, 0 \ldots 0) = 0$. Also, $BINS(n_1, n_2 \ldots n_s) = \min_{(b_1,b_2 \ldots b_s) feasible}\{1 + BINS(n_1 - b_1, n_2 - b_2 \ldots n_s - b_s)\}$.

What is the complexity of the dynamic programming algorithm? There are at most $n^s$ different $(n_1, n_2 \ldots n_s)$ configurations for BINS, and at most $\lfloor\frac{1}{\epsilon}\rfloor^s$ feasible $(b_1, b_2 \ldots b_s)$ configuration. So the complexity is, $O(\lfloor\frac{n}{\epsilon}\rfloor^{\lfloor\frac{1}{\epsilon^2}\rfloor})$. Note that, unlike the situation with knapsack, $\epsilon$ is now in the exponent. This is understandable: the problem is strongly NP-complete.

After scaling the solution back, we are done with stage 1. The solution we've gotten is superoptimal because we were rounding down the normalized sizes, which made the solution to the scaled problem superoptimal, and because the problem is invariant under scaling. Because we undid the rounding down, feasibility may have been violated. To prove that it has not been violated too much, consider the rounded-down configuration $(x_1, x_2 \ldots x_s)$ that required $m$ bins:

$$\Sigma_i \epsilon t_i \leq \Sigma_i x_i (i+1)\epsilon^2 = \Sigma_i x_i i \epsilon^2 + \Sigma_i x_i \epsilon^2 = m + \epsilon^2 \frac{m}{\epsilon} = m(1+\epsilon)$$

Stage 2, packing small items, is really easy: we throw them in any nonempty bin until the total size of items in the bin becomes greater than 1. When all the bins used so far are more then full, we can start filling a new bin. The solution will remain superoptimal because all bins except possibly the last one are more than full. This concludes the algorithm.

## 2 Introduction to the Powers of Graphs Approach

The Powers of Graphs approach to bottleneck problems is the topic of the next lecture. Here is a short introduction illustrated by the weighted $k$-center problem:

Given a graph $G = (V, E)$ with edge weights $w_{e_1} \leq w_{e_2} \leq \ldots w_{e_m}$ satisfying the Triangle inequality, do the following:
Bring in one edge at a time.
If we brought in $i$ edges, and there is a $k$-center solution with maximum distance from the center $w_{e_i}$, it uses only these edges. To check whether the solution is feasible, we need to find the dominating set of the resulting graph. If it has $k$ or fewer vertices, we are done. Otherwise we should repeat the procedure, bringing in $e_{i+1}$ next.

The problem with the above algorithm is that Dominating Set is an NP-complete problem. However, if there is a good approximation algorithm to Dominating Set, we could use it above to get a good approximation to $k$-center.

Given a graph $G = (V, E)$ and $k \epsilon \mathbf{N}$, $G^k$ is defined to be $(V, E^k)$, where an edge $(u, v) \epsilon E^k$ if there is a path of length $k$ or less between $u$ and $v$ in $G$. If we assume the Triangle inequality, the length of the longest edge in $G^2$ is at most twice the length of the longest edge in $G$. This proved to be very useful in approximation algorithms.

**Lecturer:** Dorit Hochbaum
**Scribe:** Howard Thompson

**Today's class:** Vertex and Set Cover approximation:
Primal-Dual approach
Linear Programming algorithm
Bar-Yehuda and Even's feasible dual algorithm

# 1 The LP algorithm for Set cover

An approximation algorithm - which is duality based - was devised for the set cover problems. This was motivated by an approximation to the *unweighted* vertex cover problem by Gavril. Gavril's algorithm is based on the idea of solving for a maximal (not necessarily maximum) matching, and taking both endpoints of the edges in the matching. The number of edges in the maximal matching $|M|$, is a lower bound on the optimum, $|VC^*|$. This is because a single vertex cannot cover two edges in $M$. On the other hand, if we pick both endpoints of each matched edge we get a feasible cover, $VC^M$, as otherwise there would be an edge with both endpoints unmatched, and therefore this edge could be added to the matching $M$ - contradicting its maximality. These two statements lead to the inequalities,

$$|VC^M| = 2|M| \le 2|VC^*|. \tag{4}$$

Hence this cover is at most twice the optimal cover.

The extension of this idea to the weighted case and to the set cover problem was inspired by an alternative way of viewing the maximal matching algorithm as a feasible dual solution. To see that, consider the linear programming relaxation of the unweighted vertex cover problem.

$$(VC) \qquad \begin{array}{ll} \min & \sum_{j \in V} x_j \\ \text{subject to} & x_i + x_j \ge 1 \qquad (\text{for } (i,j) \in E\ ) \\ & 0 \le x_j \le 1, \quad \forall\ j \in V. \end{array}$$

And its corresponding dual.

$$(VC-dual) \qquad \begin{array}{ll} \max & \sum_{(i,j)\in E} y_{ij} \\ \text{subject to} & \sum_{(i,j)\in E} y_{ij} \le 1 \qquad (\text{for } j = 1,\ldots,n\ ) \\ & y_{ij} \ge 0, \quad \forall\ (i,j) \in E. \end{array}$$

An integer feasible solution to $(VC - dual)$ is a matching in the graph. Consider a feasible solution to the dual, $\bar{\boldsymbol{y}}$, and let $\bar{x}_j = 1$ whenever the dual constraint is binding, $\sum_{(i,j)\in E} \bar{y}_{ij} = 1$. We show that the solution $\bar{\boldsymbol{x}}$ is a feasible vertex cover. To that end, we introduce the concept of maximality: a feasible solution to $(VC - dual)$ $\bar{\boldsymbol{y}}$ is said to be *maximal* if there is no feasible solution $\boldsymbol{y}$ such that $y_{ij} \ge \bar{y}_{ij}$ and $\sum_{(i,j)\in E} y_{ij} > \sum_{(i,j)\in E} \bar{y}_{ij}$.

**Lemma 1.1** *Let $\bar{\boldsymbol{y}}$ be a maximal feasible solution to $(VC - dual)$. Then the set*

$$VC = \{i | \sum_{(i,j)\in E} \bar{y}_{ij} = 1\}$$

*is a feasible solution to $(VC)$.*

**Proof:** Suppose that $VC$ is not a feasible cover. Then there is an edge $(u, v)$ which is uncovered, i.e., $\sum_{(u,j)\in E} y_{uj} < 1$ and, $\sum_{(v,j)\in E} y_{vj} < 1$.
Let $\delta = \min\{1 - \sum_{(u,j)\in E} y_{uj}, 1 - \sum_{(v,j)\in E} y_{vj}\}$. Then the vector,

$$\boldsymbol{y} = \bar{\boldsymbol{y}} + \delta \cdot e_{uv}$$

(for $e_{uv}$ denoting the vector of all zeros except for a 1 in the $uv$ entry), is a feasible solution satisfying, $y_{ij} \geq \bar{y}_{ij}$ and $\sum_{(i,j)\in E} y_{ij} > \sum_{(i,j)\in E} \bar{y}_{ij}$. This contradicts the maximality of $\bar{\boldsymbol{y}}$. Hence every edge must be covered by $VC$ and VC is therefore a feasible cover. ■

Consider now the generalization of this approach for the set cover problem.

$$(SC) \qquad \begin{array}{lll} \min & \sum_{j=1}^{n} w_j x_j & \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} x_j \geq 1 & (\text{for } i = 1, \ldots, n\ ) \\ & 0 \leq x_j \leq 1, \quad \forall\, j. & \end{array}$$

And its corresponding dual.

$$(SC - dual) \qquad \begin{array}{lll} \max & \sum_{i=1}^{m} y_i & \\ \text{subject to} & \sum_{i=1}^{m} a_{ij} y_i \leq w_j & (\text{for } j = 1, \ldots, m\ ) \\ & y_i \geq 0. & \end{array}$$

First, a feasible solution to $(SC - dual)$ $\bar{\boldsymbol{y}}$ is said to be *maximal* if there is no feasible solution $\boldsymbol{y}$ such that $y_i \geq \bar{y}_i$ and $\sum_{i=1}^{n} y_i > \sum_{i=1}^{n} \bar{y}_i$.

**Lemma 1.2** *Let $\bar{\boldsymbol{y}}$ be a maximal feasible solution to $(SC - dual)$. Then the set*

$$SC = \{j | \sum_{i=1}^{m} a_{ij} \bar{\boldsymbol{y}}_i = w_{ij}\}$$

*is a feasible cover. That is, $x_j = 1$ for $j \in SC$ is a feasible solution to (SC).*

**Proof:** Similar to lemma 1. Suppose that $SC$ is not a feasible cover, and $q$ is the uncovered element. Let $\delta = \min_{q\in S_j}\{w_j - \sum_{i=1}^{m} a_{ij}\bar{\boldsymbol{y}}_i\}$. Then $\boldsymbol{y}$ defined as:

$$\boldsymbol{y}_q = \bar{\boldsymbol{y}}_q + \delta$$

$$\boldsymbol{y}_i = \bar{\boldsymbol{y}}_i,\ i \neq q$$

is a feasible solution contradicting the maximality of $\bar{y}$. Therefore all elements are covered by $SC$ and it is a feasible cover. ■

**The LP- algorithm**
**Step 1:** Find a maximal dual feasible solution for $(SC)$, $\bar{\boldsymbol{y}}$.
**Step 2:** Output the cover $C^H = \{j | \sum_{i=1}^{n} \bar{y}_i = w_j\}$.

**Lemma 1.3** *$C^H$ is a feasible solution to $(SC)$.*

**Proof:** The proof is an obvious extension of Lemma 1.1: Consider an element $p$ that is not covered. Let $\delta = \min_{j|p\in S_j}\{w_j - \sum_{j=1}^n a_{ij}\bar{y}_i\}$. Then the vector,

$$\boldsymbol{y} = \bar{\boldsymbol{y}} + \delta \cdot e_p$$

is a feasible solution contradicting the maximality of $\bar{\boldsymbol{y}}$. ∎

Let $C^*$ be the optimal cover. Define for any set cover $C$, $w(C) = \sum_{j\in C} w_j$. The following lemma shows that the the LP- algorithm is a $\max_i\{\sum_j a_{ij}\}$-approximation algorithm.

**Lemma 1.4** $w(C^H) \leq \max_i\{\sum_j a_{ij}\}w(C^*)$.

**Proof:**

$$w(C^H) = \sum_{j\in C^H} w_j = \sum_{j\in C^H}(\sum_{i=1}^m a_{ij}y_i).$$

Using the weak duality theorem we get that for any solution to the linear programming relaxation, and in particular for the optimal solution $\boldsymbol{x}^*$,

$$\sum_{j\in C^H}(\sum_{i=1}^m a_{ij}y_i) \leq \max_i\{\sum_{j\in C^H} a_{ij}\}\sum_{i=1}^m y_i \leq \max_i\{\sum_{j\in C^H} a_{ij}\}\sum_{j=1}^n w_j x_j^* \leq \max_i\{\sum_{j\in C^H} a_{ij}\}w(C^*).$$

Now the value of the optimal solution to the linear programming relaxation is a lower bound to the optimal integer solution. It follows that, $w(C^H) \leq \max_i\{\sum_{j\in C^H} a_{ij}\}w(C^*)$. ∎

This proves a slightly stronger approximation factor as we can consider the row sums restricted only to those sets in the cover (alternatively the row sums are calculated in the submatrix of the columns in the cover). For instance, if the cover $C^H$ has only one set covering each element, then it is optimal. Let the maximum number of sets covering an element, $\max_i\{\sum_j a_{ij}\}$ be denoted by $p$.

An immediate corollary of the $p$-approximation is that the LP- algorithm is a 2-approximation algorithm for the (weighted) vertex cover problem.

**Corollary 1.1** *The LP- algorithm is a 2-approximation algorithm for the vertex cover problem.*

**Proof:** $(VC)$ is a special case of $(SC)$ with each element - an edge - belonging to precisely two "sets" representing its endpoints. Hence, $\sum_j a_{ej} = 2$ for all edges $e \in E$. ∎

The implementation of the LP-algorithm used the optimal dual solution as a maximal feasible solution. Still, the role of the dual solution is only to aid the analysis of the algorithm. It need not be generated explicitly by the algorithm:

**The rounding algorithm**

**Step 1:** Solve optimally the linear programming relaxation of (SC). Let an optimal solution be $\{x_j^*\}$

**Step 2:** Output the cover $C^H = \{j|x_j^* > 0\}$. Equivalently, set $x_j^H = \lceil x_j^* \rceil$.

The rounding algorithm is indeed a special case of the LP-algorithm as $x^*_j > 0$ implies that the corresponding dual constraint is binding by complementary slackness optimality conditions, $\sum_{i=1}^m a_{ij} y_i = w_j$.

A minor variation of the rounding algorithm is still a $p$-approximation algorithm. We replace step 2 by:

**Step 2':** Output the cover $C^H = \{j | x^*_j \geq \frac{1}{p}\}$.

The feasibility of this cover is obvious, as in any fractional solution $\boldsymbol{x}$ corresponding to a cover $C$, $\sum_{j=1}^n x_j \geq 1$, and there are at most $p$ positive entries per such inequality. So at least one must be at least as large as the average $\frac{1}{p}$. This rounding algorithm (rounding II) will always produce a cover no larger than the rounding algorithm. It does not offer any advantage though in terms of worst case analysis. Moreover, for any cover produced by an approximation algorithm, it is easy to prune it of unnecessary extra sets, and leave a "prime" cover, which is a *minimal* cover. None of these approaches has provided tighter approximation factors.

In the next section we see that it is not necessary to compute the optimal solution to the linear programming. Rather, there are more efficient ways of finding a dual maximal solution, two of which are described in the next section. On the other hand, as discussed in the section on the preprocessing algorithm, the linear programming solution carries some extra valuable information for the vertex cover problem. In addition, for the vertex cover problem it is possible to solve the linear programming relaxation by applying a max-flow min-cut algorithm which is more efficient than solving the respective linear program.

## 2 The feasible dual approach

Bar-Yehuda and Even [2] devised an efficient algorithm for identifying a maximal feasible dual solution to be used in the LP-algorithm. As before, the derivation of the dual solution is implicit and exists only in order to analyze the approximation factor. This dual information is placed in square brackets in the description of the algorithm to stress that it is not an integral part of the procedure.

**The Dual-feasible Algorithm [Bar-Yehuda and Even]**
**Step 0:** Set $C = \emptyset$; $I = \{1, \dots, m\}$; $[\boldsymbol{y} = 0]$.
**Step 1:** Let $i \in I$. Let $w_{j(i)} = \min_{a_{ij}=1} w_j$. $[y_i = w_{j(i)}]$; $C \leftarrow C \cup \{j(i)\}$.
**Step 2:** {update} For all $j$ such that $a_{ij} = 1$, $w_j \leftarrow w_j - w_{j(i)}$, $I \leftarrow I \setminus S_j(i)$.
**Step 3:** If $I = \emptyset$, stop and output cover $C$. Else go to step 1.

The vector $\boldsymbol{w}$ represents in fact the reduced costs corresponding to the solution $\boldsymbol{y}$, and at each iteration quantify the amount of slack in the dual constraint. The strategy is to choose one of the remaining uncovered elements, identified by the set $I$, and among the sets covering that element, add the one with the lowest reduced cost, represented by $w_j$.

Throughout this procedure the updated $w_j$'s remain nonnegative. In this sense, the algorithm is dual-feasible - it maintains throughout the feasibility of the dual vector $\boldsymbol{y}$. For any set added to the cover, the updated value of the reduced cost is 0, i.e. the corresponding dual constraint is binding. The resulting dual vector is maximal since each element belongs to some set in the cover, and therefore to some set with a binding dual constraint. Therefore, there is no vector that is larger or equal to $\boldsymbol{y}$ in all its components which is feasible, unless it is equal to $\boldsymbol{y}$.

**Lemma 2.1** *The cover $C$ from the Bar-Yehuda and Even algorithm is a $p$-approximation for $(SC)$, where $p = \max_i\{\sum_j a_{ij}\}$.*

**Proof:** Throughout the algorithm $w_j - \sum_{i=1}^{m} a_{ij}y_i \geq 0$, so $\boldsymbol{y}$ is dual feasible. It is also maximal since, at each step the minimum residual weight is chosen and the corresponding constraint is made tight, so it can not be incremented as in Lemma 1.1. The result then follows from Lemma 1.4. ■

The complexity of the Bar-Yehuda and Even dual-feasible algorithm is $O(mn)$, which is linear in the size of the input matrix, and hence a considerable improvement to the complexity of the respective linear program. For the vertex cover problem the complexity is $O(m)$. As in corollary 1.1, this algorithm is a 2-approximation to the vertex cover problem.

There is no proof of this approximation bound that does not use this idea of duality, although linear programming is not used in the algorithm itself. As mentioned before, the implicit use of duality is included only to facilitate analysis. In the next class we will see another algorithm of Chvátal [1] that also uses duality in this manner and results in a different approximation bound.

# References

[1] V. Chvátal,"A Greedy Heuristic for the Set-Covering Problem,"*Math. of Oper. Res.* Vol. 4, 3, 233-235 (1979).

[2] R. Bar-Yehuda and S. Even, "A linear time approximation algorithm for the weighted vertex cover problem," *J. of Algorithms* **2** (1981) 198–203.

[3] D. S. Hochbaum, "Approximation algorithms for the set covering and vertex cover problems," SIAM J. Comput. 11(3) (1982), an extended version: W.P. #64-79-80, GSIA, Carnegie-Mellon University, April (1980).

**Lecturer:** Dorit Hochbaum
**Scribe:** Derek Bennett

**Last Class:**

Last class was devoted to approximations for the set cover problem. In particular, there were three main approaches:

- LP algorithm
- Rounding algorithm
- Maximal feasible dual algorithm – general purpose

All of these were $p$-approximation algorithms, where $p$ is the maximum row sum of the matrix $A$ in the set cover problem. Note that in vertex cover, $p = 2$.

**Today's Agenda:**

Two main topics in today's lecture:

- Review and reprove approximation bound for maximal feasible dual algorithm
- Present greedy algorithm for set cover problem

The first item is a continuation from the last class. The greedy algorithm is due to Chvátal [1].

# 1 The maximal dual feasible algorithm for set cover

This section gives an additional proof of the bound on the error for the maximal feasible dual algorithm (by Bar Yehuda and Even) for set cover. Recall the weighted set cover problem: we are to choose sets (columns), at minimum cost, such that all elements (rows) are covered. The following notation was used in describing the problem:

$$
\begin{aligned}
x_j &= \begin{cases} 1, & \text{if set } j \text{ is part of the cover} \\ 0, & \text{otherwise} \end{cases} \\
y_i &= \text{value of dual variable } i \\
w_j &= \text{weight of set } j \\
a_{ij} &= \begin{cases} 1, & \text{if element } i \text{ is in set } j \\ 0, & \text{otherwise} \end{cases}
\end{aligned}
$$

Recall that the dual feasible algorithm proceeds by choosing a set to be in the cover, and then updates the *reduced costs* after choosing the set. The LP dual is not solved, and the dual variables are not an explicit part of the algorithm, however, they will be referred to in the proof of the following theorem.

**Theorem 1.1** *The maximal feasible dual algorithm is a p-approximation, where p is the maximum row sum in the matrix A:* $p = \max_j \sum_j a_{ij}$.

**Proof:** Let $C$ be the collection of sets chosen by the algorithm. Recall that when choosing sets, the reduced costs are greater than zero. In fact, for $j \in C$, this constraint is binding, and the reduced cost (equivalently, dual slack) equals zero. More formally, for all $j$:

$$w_j - \sum_i a_{ij} y_i \geq 0$$

where the inequality becomes an equality for those sets in the cover, $C$. Rearranging, summing over $j \epsilon C$, and reversing the summation order on the RHS leads to:

$$\begin{aligned} \sum_{j \epsilon C} w_j &= \sum_i \sum_{j \epsilon C} a_{ij} y_i \\ &\leq \max_i \sum_{j \epsilon C} a_{ij} \cdot \sum_{i=1}^{m} y_i \\ &\leq p \cdot \sum_{j=1}^{n} w_j x_j^* \quad \text{by weak duality} \end{aligned}$$

where $x_j^*$ is the optimal LP solution. From the last inequality, $\sum_{j \epsilon C} w_j \leq p \cdot \mathcal{OPT}$, where $\mathcal{OPT}$ is the optimal solution. Thus, we have a p-approximation. ∎

## 2 The greedy algorithm for the Set Cover problem

Johnson and Lovàsz ([2], [3]) were the first to demonstrate that a greedy algorithm is a $H(d)$-approximation algorithm for the unweighted set cover problem, where $H(d) = \sum_{i=1}^{d} \frac{1}{i}$ (harmonic series) and $d$ is the size of the largest set. $H(d)$ is bounded by $1 + \log d$. The greedy algorithm works by selecting one set at a time that covers the most elements among the uncovered ones.

Chvátal [1] extended the applicability of the greedy to the weighted set cover. This greedy selects a set with the minimum ratio of weight to remaining coverage. Chvátal proved that this greedy algorithm is still a $H(d)$-approximation algorithm.

**The Greedy Algorithm [Chvátal]**
**Step 0:** Set $C^G = \emptyset$; $S_j^1 = S_j, j \in J$; $I = \{1, \ldots, m\}$; $k = 0$.
**Step 1:** $k \leftarrow k + 1$. Select a set $S_{j_k}$, such that $\frac{w_{j_k}}{|S_{j_k}^k| = \min_{j \in J} \frac{w_j}{|S_j^k|}}$.
**Step 2:** Set $C^G \leftarrow C^G \cup \{j_k\}$ and $S_j^{k+1} = S_j^k \setminus S_{j_k}^k, j \in J$, $I \leftarrow I \setminus S_{j_k}^k$.
**Step 3:** If $I = \emptyset$, stop and output cover $C^G$. Else, go to step 1.

Chvátal's proof of the heuristic bound is remarkable in its innovative use of duality as an analysis tool.

Consider the linear programming relaxation of (SC), with the upper bound $x_j \leq 1$ constraints omitted (an optimal solution will satisfy those constraints anyway). The dual problem is

$$
\begin{array}{lll}
 & \text{Max} & \sum_{i=1}^m y_i \\
(SC-dual) & \text{subject to} & \sum_{i=1}^m a_{ij} y_i \leq w_j \quad (\text{for } j=1,\ldots,n\ ) \\
 & & y_i \geq 0 \quad (i=1,\ldots,m)
\end{array}
$$

The proof relies on allocating the weights of the set selected by the greedy heuristic to the elements covered and interpreting those as form of dual, not quite feasible, solution.

**Theorem 2.1** *The greedy heuristic is a $H(d)$-approximation algorithm.*

**Proof:** To prove the desired result, it suffices to show that for *any* cover $C$, indicated by the characteristic vector $\{x_j\}$, and a cover delivered by the greedy $C^G$,

$$\sum_{j\in C} H(d) w_j = \sum_{j=1}^n H(d) w_j x_j \geq \sum_{j \in C^G} w_j \ . \tag{5}$$

Applying this inequality to $C^*$, the optimal cover, yields that the value of the solution delivered by the greedy is at most $H(d)$ times the value of the optimal solution. To prove (5), it is sufficient to find an "almost feasible" dual solution $\boldsymbol{y}$ such that,

$$\sum_{i=1}^m a_{ij} y_i \leq H(|S_j|) w_j \quad j=1,\ldots,n \tag{6}$$

and so that the weight of the sets selected is accounted for by $\boldsymbol{y}$,

$$\sum_{i=1}^m y_i = \sum_{j\in C^G} w_j. \tag{7}$$

Such $\boldsymbol{y}$ satisfying these inequalities is feasible within a factor of $H(d)$, and it satisfies (5) since,

$$\sum_{j=1}^n H(d) w_j x_j \overset{(6)}{\geq} \sum_{j=1}^n \left(\sum_{i=1}^m a_{ij} y_i\right) x_j = \sum_{i=1}^m \left(\sum_{j=1}^n a_{ij} x_j\right) y_i \geq \sum_{i=1}^m y_i \overset{(7)}{=} \sum_{j\in C^G} w_j. \tag{8}$$

Let $S_j^k$ be the set $S_j$ with the remaining elements at the beginning of iteration $k$, and its size, $|S_j^k| = s_j^k$. The dual vector $\boldsymbol{y}$ that will satisfy (6) and (7) has for $y_i$ the average price paid by the greedy to cover an element $i$. Whenever a set is selected, its weight is divided evenly among the elements it has newly covered, $y_i = \frac{w_k}{s_k^k}$.

Let the sets greedy selects in the first $k$ iterations be $\{1,2,\ldots,k\}$. Since $k$ is the index for which the ratio is minimum

$$\frac{w_k}{s_k^k} \leq \frac{w_j}{s_j^k} \qquad \forall j. \tag{9}$$

Assume that there are $t$ iterations altogether. Then $\sum_{j\in C^G} w_j = \sum_{j=1}^{t} w_j$. Each element $i \in I$ belongs to one set $S_k^k, k = 1, \ldots, t$, so for $i \in S_k^k$, $y_i = \frac{w_k}{s_k^k}$ . (7) now follows,

$$\sum_{i=1}^{m} y_i = \sum_{k=1}^{t} \sum_{i\in S_k^k} y_i = \sum_{k=1}^{t} s_k^k (\frac{w_k}{s_k^k}) = \sum_{k=1}^{t} w_k \ .$$

To prove (6) observe that, $S_j \cap S_k^k = S_j^k \setminus S_j^{k+1}$ and $I = \bigcup_{k=1}^{t} S_k^k$. Hence,

$$\sum_{i=1}^{m} a_{ij} y_i = \sum_{k=1}^{t} \sum_{i\in S_j\cap S_k^k} y_i = \sum_{k=1}^{t} \sum_{i\in S_j^k\setminus S_j^{k+1}} y_i = \sum_{k=1}^{t} \left(s_j^k - s_j^{k+1}\right) \frac{w_k}{s_k^k} \ .$$

For a given set $S_j$, let $p$ be the largest index such that $s_j^p > 0$, then

$$\sum_{i=1}^{m} a_{ij} y_i = \sum_{k=1}^{p} (s_j^k - s_j^{k+1}) \frac{w_k}{s_k^k} \overset{(9)}{\leq} w_j \sum_{k=1}^{p} \frac{s_j^k - s_j^{k+1}}{s_j^k} \ .$$

We now use the inequality $\frac{s_j^k - s_j^{k+1}}{s_j^k} \leq H(s_j^k) - H(s_j^{k+1})$ to establish,

$$\sum_{i=1}^{m} a_{ij} y_i \leq w_j \sum_{k=1}^{p} \left(H(s_j^k) - H(s_j^{k+1})\right) \leq w_j H(s_j^1) \ .$$

■

# References

[1] V. Chvátal,"A Greedy Heuristic for the Set-Covering Problem,"*Math. of Oper. Res.* Vol. 4, 3, 233-235 (1979).

[2] D. S. Johnson, "Approximation Algorithms for Combinatorial Problems," *J. Comput. System Sci.*, 9, 256-278 (1974).

[3] L. Lovàsz, "On the Ratio of Optimal Integral and Fractional Covers," *Discrete Math.* 13 383-390 (1975).

---

**Lecturer:** Dorit Hochbaum
**Scribe:** Wooseung Jang

---

Last class we saw that the greedy is an $H(d)$-approximation for set cover. Today we will study about preprocessing in approximating vertex cover and independent set, cuts, cutting planes and facets generation.
( Ch 14 [P&S], pp 209-275 Nemhauser & Wolsey )

## Preprocessing

We know that there is an optimal LP–relaxation solution to $VC$ is $\{x_j^*\}$, $x_j^* \in \{0, \frac{1}{2}, 1\}$.
Nemhauser and Trotter proved that there exists optimal $VC$ solution $\boldsymbol{z}^*$ such that $z_j^* = x_j^*$ for $x_j^* \in \{0, 1\}$.

Now let's do preprocessing .
Let $H_{\frac{1}{2}} = (V_{H_{\frac{1}{2}}}, E(V_{H_{\frac{1}{2}}}))$ be a subgraph with $V_{H_{\frac{1}{2}}} = \{v | x_v^* = \frac{1}{2}\}$.
Fix $x_v = 1$ if $x_v^* = 1 \longrightarrow$ say $H_1$
$x_v = 0$ if $x_v^* = 0 \longrightarrow$ say $H_2$.
Given a heuristic for $VC$ , apply it to $H_{\frac{1}{2}}$ and get a cover $VC^{H_{\frac{1}{2}}}$ then add it to $H_1$ to create $VC^H$. Then $VC^H$ is a vertex cover delivered by the heuristic.

$$w(VC^*) \geq w(H_1) + \frac{1}{2} w(H_{\frac{1}{2}})$$

$$\Longrightarrow \frac{w(VC^H)}{w(VC^*)} \leq \frac{w(VC^{H_{\frac{1}{2}}}) + w(H_1)}{\frac{1}{2} w(H_{\frac{1}{2}}) + w(H_1)} \leq \frac{2w(VC^{H_{\frac{1}{2}}})}{w(H_{\frac{1}{2}})} \leq 2.$$

For some graphs, we can get $< 2$ approximation:
Consider a $k$-coloration of a graph.
Let $V = V_1 \cup V_2 \cup \cdots \cup V_k$ where all the nodes in $V_i$ have same color. $V \setminus V_i$ is a feasible vertex cover. Let $V_i$ be the heaviest one , i.e. $\max_{j=1 \cdots k} w(V_j) = w(V_i)$. So $V \setminus V_i$ is a vertex cover such that $w(V \setminus V_i) \leq (1 - \frac{1}{k}) w(V)$. $(w(V_i) \geq \frac{1}{k} w(V)$ )
Apply this as a heuristic to $H_{\frac{1}{2}}$ (that is also $k$-colorable). $(H_{\frac{1}{2}} - V_i) \cup H_1$ is a cover, and

$$\frac{w(VC^H)}{w(VC^*)} \leq \frac{2w(VC^{H_{\frac{1}{2}}})}{w(H_{\frac{1}{2}})} \leq \frac{2(1 - \frac{1}{k}) w(H_{\frac{1}{2}})}{w(H_{\frac{1}{2}})} = 2(1 - \frac{1}{k}) = 2 - \frac{2}{k}.$$

### Gomory Cuts

We next describe an algebraic method for generating cuts due to R. E. Gomory. Suppose we are given an instance of ILP and begin by solving the LP–relaxation, producing an optimal basic feasible solution $x^{\mathcal{B}}$ associated with basis $\mathcal{B}$. If solution is integer then stop. Otherwise add a cutting plane such that $x^*$ is not feasible and all feasible integer points remain feasible. And then, repeat above processes.

Consider any equation in the final tableau, say the $i$th equation

$$x_{B(i)} + \sum_{j \notin B} y_{ij} x_j = y_{i0} \tag{1}$$

for some $i, 0 \le i \le m$.
Due to nonnegativity of $x$

$$\sum_{j \notin B} \lfloor y_{ij} \rfloor x_j \le \sum_{j \notin B} y_{ij} x_j$$

Therefore (1) becomes

$$x_{B(i)} + \sum_{j \notin B} \lfloor y_{ij} \rfloor x_j \le y_{i0}$$

Since left hand side is integer, we also get

$$x_{B(i)} + \sum_{j \notin B} \lfloor y_{ij} \rfloor x_j \le \lfloor y_{i0} \rfloor \tag{2}$$

Subtracting (2) from (1), we get the constraint

$$\sum_{j \notin B} (y_{ij} - \lfloor y_{ij} \rfloor) x_j \ge y_{i0} - \lfloor y_{i0} \rfloor$$

called the *Gomory cut* corresponding to the *source* or *generating row i.*

**Lecturer:** Dorit Hochbaum
**Scribe:** Thomas R. Nitsch

## Lenstra's Algorithm

$$Ax \le b$$
$$x \ \ integer$$
$$x \ \in Z^n$$

Consider n fixed:

1. Decision Problem: Is there a feasible solution?
2. Optimization Problem: Do a binary search on the following interval:

$$LB = -P(A,b,c), UB = P(A,b,c)$$

where P(A,b,c) = P(log A, log b, log c) $\Rightarrow$ can also accomplish the optimization problem

## Overview of the algorithm

Polytope K = $\{x \in R^n \ | \ Ax \le b\}$
Find the transformation matrix $\tau$: $B(P,r) \subset \tau K \subset B(P,R)$
Find $\tau$ st

$$\frac{R}{r} \le \ C(n) \qquad (FACT3)$$

If P integer $\rightarrow$ done
else round to nearest integer Y
  if Y $\in$ Polytope $\rightarrow$ done
  else for $b_1 ... b_n$ the reduced basis for the lattice

$$L = \bigcup_{k=1}^{t} (H + Kb_n) \quad ; t \le C(n)$$

*Idea:* Show that the diameter of the big sphere does not contain too many of these hyperplanes.
In $H + k\ b_{n-1}$

$$X = \sum_{i=1}^{n-1} \alpha_i b_i + k b_n \quad where\ \alpha_i\ integer$$

Given a lattice defined by $b'_1 ... b'_n$,
Let $b_1 ... b_n$ be a it reduced basis
Property: $d(L) \overset{Fact(1)}{\leq} \prod_{i=1}^{n} |\ b_i\ | \overset{Fact(2)}{\leq} c_2(n)\ d(L)$
Note: Fact(1) holds for any basis.
Fact(2) holds for reduced basis.
*Question:* Is there an integer point in k?
We can always assume:
(i) k is bounded.
(ii) k has positive volume (if fully dimensional - meaning it cannot be embedded in one dimension less)
if not, reduce to lower dimensional polytype using LP.
For k with properties (i) and (ii) one can find, using the Ellipsoid method, $\tau$ st
$B(P, r) \subset \tau K \subset B(P, R)$
Rename, such that $|b_n| \geq |b_{n-1} \geq \cdots \geq |b_1|$

**Lemma 0.1**

$$\forall x \in R^n, \exists \gamma \in L \qquad L = L(b_1 \cdots b_n)$$

$$st|X - Y|^2 \leq \frac{1}{4}(|b_1|^2 + \cdots + |b_n|^2)$$

$$\Rightarrow |X - Y| \leq \tfrac{1}{2}\sqrt{n}\ |b_n|$$

Let $H = \sum_{i=1}^{n-1} R\ b_i = \left\{\sum_{i=1}^{n-1} \lambda_i b_i \mid \lambda_1 \cdots \lambda_{n-1} \in R\right\}$

Fact 4: h $\cdots$ distance $(b_n, H) \leq |b_n|$
L' is the lattice generated by $(b_1 \cdots b_{n-1}$.
Fact 5: $d(L) = h\ d(L)$

$$\text{Now } \prod_{i=1}^{n} |b_i| \overset{F2}{\leq} c_2\ d(L) = c_2\ h\ d(L') \overset{F1}{\leq} c_2\ h\ \prod_{i=1}^{n-1} |b_i|$$

$\Rightarrow$ Fact 6: $|b_n| \leq c_2\ h$ (within a constant of the orthogonal basis)
Consider P (the center of the concentric spheres)

$$\Rightarrow \exists y \in L\ st\ \ |P - \gamma| \leq \tfrac{1}{2}\sqrt{n}\ |b_n| \qquad (Lemma)$$

Either $\gamma \in \tau\ k$, hence it is the desired integer solution (after transformation).
Or $|P - \gamma| > r \quad \Rightarrow \quad (Fact7) : r < \frac{1}{2}\sqrt{n}\ |b_n|$
$L = (L' + Z\ b_n) \subset (H + Z\ b_n) = \bigcup_{k \in Z} (H + k\ b_n)$

Distance$(H + k\ b_n;\ H + (k-1)\ b_n) =$
$=$ distance $(H + b_n, H) =$ h
Recall: $\tau k \subset B(P, R)$
t = number of hyperspaces intersecting $\tau k$ satisfies

$$t \leq \lceil \tfrac{2R}{h} \rceil \qquad \text{(want to show that t is "small enough").}$$

$2R \overset{F3}{\leq} 2rc_1 \overset{F7}{<} c_1\sqrt{n}b_n \overset{F6}{\leq} c_1c_2\sqrt{n}h \quad \Rightarrow \quad t \leq \lceil c_1c_2\sqrt{n}\rceil$
Solve the problem on each of the t hyperplanes by substituting

$$x = \sum_{i=11}^{n-1} y_i b_i + kb_n$$

for k assuming up to t values.
Let T(n) be the time to solve n-dimensional problem

$T(n) = c_1c_2\sqrt{n}T(n-1)$

Complexity: O($c_1c_2\sqrt{n}$)

More precisely: O($n^{2n}\ 2^{O(n^3)}$ $+$ complexity of ellipsoid)

**NOTE:** Result applies also for fixed number of integers by reducing A to its *Hermite Normal Form*

## Diophantine Equations

(covered by handout)

**Definition:** Matrix of full rank is in *Hermite Normal Form* (HNF) if it is of the form [B,0] with B nonsingular, lower triangular, and nonnegative matrix in which each row has a unique maximum located on the main diagonal of B.

ElementaryColumn Operations:

(i) exchange two columns
(ii) multiply a column by -1;
(iii) adding an integral multiple of one column to another column.

**Theorem 4.1:** (HNF Theorem)
Each rational matrix of full rank can be brought into HNF by a polynomial series of elementary column operations.