4 The Greedy Algorithm and Computational Complexity

4.1 Matroid

- 1935, matroid theory founded by H. Whitney;
- 1965, J. Edmonds pointed out the significance of matroid theory to combinatorial optimization (CO).

Importance: 1) Many CO problems can be formulated as matroid problems, and solved by the same algorithm;

2) We can detect the insight of the CO problems;

3) A special tool for CO.

Definition 4.1 *Suppose we have a finite ground set S, $|S| < \infty$, and a collection, Ξ, of subsets of S. Then $H := (S, \Xi)$ is said to be an independent system if the empty set is in Ξ and Ξ is closed under inclusion; that is*

i) $\emptyset \in \Xi$;

ii) $X \subseteq Y \in \Xi \Longrightarrow X \in \Xi$.

Elements in Ξ are called independent sets, and subsets of S not in Ξ are called dependent sets.

Example: Matching system. $G = (V, E)$,

$$\Xi = \{\text{all matchings in G}\}.$$

[A matching M of a graph $G = (V, E)$ is a subset of the edges with the property that no two edges of M share the same node. A matching M is a piecewise disjoint edge set]

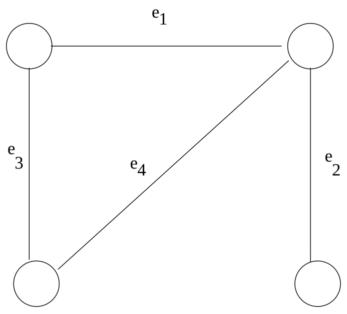

Figure 4.1: A Matching Example

In Figure 4.1,

$$\boxed{S = \{e_1, e_2, e_3, e_4\},\ \Xi = \{\emptyset, \{e_1\}, \{e_2\}, \{e_3\}, \{e_4\}, \{e_2, e_3\}\}.}$$

Definition 4.2 *If $H = (S, \Xi)$ is an independent system such that*

$$X, Y \in \Xi,\ |X| = |Y| + 1 \implies$$

$$\text{there exists } e \in X \backslash Y \text{ such that } Y + e \in \Xi,$$

then H (or the pair (S, Ξ)) is called a matroid.

Examples: i) Matric matroid: A matrix $A = (a_1, \ldots, a_n)_{m \times n}$, $S = \{a_1, \ldots, a_n\}$,

$$X \in \Xi \iff X = \{a_{i_1}, \ldots, a_{i_k}\} \text{ is independent.}$$

ii) Graphic matroid: $G = (V, E)$, $S = E$,

$$X \in \Xi \iff X \subseteq E,\ X \text{ has no cycle.}$$

ii) is a special case of i) with $A =$ the vertex-edge incidence matrix.

4.2 The Greedy Algorithm

Suppose that $H = (S, \Xi)$ is an independent system and $W : S \to \Re_+$ is a weight function with $W(e) \geq 0\ \forall e \in S$. For $X \subseteq S$, let

$$W(X) := \sum_{e \in X} W(e).$$

Then the matroid problem is

$$\begin{aligned} \max \quad & W(X) \\ \text{s.t.} \quad & X \in \Xi. \end{aligned}$$

Greedy Algorithm:

Suppose $W(e_1) \geq W(e_2) \geq \ldots \geq W(e_n)$.

Step 0. Let $X = \emptyset$.

Step k. If $X + e_k \in \Xi$, let $X := X + e_k$, where $k = 1, \ldots, n$.

Theorem 4.1 *(Rado, Edmonds) The above algorithm works if and only if H is a matroid.*

Applications:

1) The Maximal Spanning Tree Problem.

Suppose that there is a television network leasing video links so that its stations in various places can be formed into a connected network. Each link (i, j) has a different rental cost c_{ij}. The question is how the network can be constructed to have the minimum cost? Obviously, what is wanted is a minimum cost spanning tree of video links. Replacing c_{ij} by $M - c_{ij}$, where M is a larger number, we can see that it then turns into a *maximum spanning tree* (MST). Kruskal has already proposed the following solution: *Choose the edges one at a time in order of their weights, largest first, rejecting an arc only if it forms a cycle with edges already chosen.*

2) A Sequencing Problem.

Suppose that there are a number of jobs which are to be processed by a single machine. All jobs require the same processing time. Each job j has assigned to it a deadline d_j, and a penalty p_j, which must be paid if the job is not completed by its deadline. What ordering of the jobs minimizes the total penalty costs? It can be easily seen that there exists an optimal sequence in which all jobs completed on time appear at the beginning of the sequence in order of deadlines, earliest deadline first. The late jobs follow, in arbitrary order. Thus, the problem is to choose an optimal set of jobs which can be completed on time. The following procedure can be shown to accomplish that objective.

Choose the jobs one at a time in order of penalties, largest first, rejecting a job only if its choice would mean that it, or one of the jobs already chosen, cannot be completed on time. [This requires checking to see that the total amount of processing to be completed by a particular deadline does not exceed the deadline in question.]

For example, consider the set of jobs below, where the processing time of each job is one hour, and the deadlines are expressed in hours of elapsed time.

Job	Deadline	Penalty
j	d_j	p_j
1	1	10
2	1	9
3	3	7
4	2	6
5	3	4
6	6	2

Job 1 is chosen, but job 2 is discarded, because the two together require two hours of processing time and the deadline for job 2 is at the end of the first hour. Jobs 3 and Jobs 4 are chosen, job 5 is discarded, and job 6 is chosen. An optimal sequence is jobs 1, 4,3, and 6, followed by the late jobs 2 and 5.

3) A Semimatching Problem.

Let W be an $m \times n$ nonnegative matrix. Suppose we wish to choose a maximum weight subset of elements, subject to the constraint that no two elements are from the same row of the matrix. Or, in other words, the problem is to

$$\begin{aligned} \text{maximize} \quad & \sum_{i,j} w_{ij}x_{ij} \\ \text{subject to} \quad & \sum_{j} x_{ij} \leq 1, \;\; i = 1, ..., m \\ & x_{ij} \in \{0, 1\}. \end{aligned}$$

This semimatching problem can be solved by choosing the largest element in each row of W. Or alternatively: *choose the elements one at a time in order of size, largest first, rejecting an element only if an element in the same row has already been chosen.*

4.3 General Introduction on Computational Complexity

Initiated in large measure by the seminal papers of S. A. Cook (1971) and R. M. Karp (1972) in the area of discrete optimization.

Definition 4.3 *An* **instance** *of an optimization problem consists of a feasible set F and a cost function $c : F \to \Re$. An optimization* **problem** *is defined as a collection of instances.*

For example, "linear programming" is a problem and $\min\{x : x \geq 3\}$ is an "instance".

Instances of a problem need to be described according to a common format. For example, instances of linear programming in standard form can be described by listing the entries of $\mathbf{A}, \mathbf{b}$, and $\mathbf{c}$. Note that some instances are "larger" than others, and it is convenient to define the notion of the "size" of an instance.

Definition 4.4 *The* **size** *of an instance is defined as the number of bits used to describe the instance, according to a prescribed format.*

Given that arbitrary numbers cannot be represented in binary, this definition is geared towards instances involving integer (or rational) numbers. Note that any nonnegative integer r smaller or equal to U can be written in binary as follows:

$$r = a_k 2^k + a_{k-1} 2^{k-1} + \ldots + a_1 2^1 + a_0,$$

where the scalars $a_0, \ldots, a_k$, are 0 or 1. The number k is clearly at most $\lfloor \log_2 U \rfloor$, since $r \leq U$. We can then represent r by the binary vector $(a_0, a_1, \ldots, a_k)$. With an extra bit for sign, we can aslo represent negative numbers. In other words, we can represent any integer with absolute value less than or equal to U using at most $\lfloor \log_2 U \rfloor + 2$ bits.

Consider now an instance of a linear programming problem in standrad form, i.e., an $m \times n$ matrix $\mathbf{A}$, an m-vector $\mathbf{b}$, and an $n-$vector $\mathbf{c}$, and assume that the magnitude of the largest element of $\{\mathbf{A}, \mathbf{b}, \mathbf{c}\}$ is equal to U. Since there are $(mn + m + n)$ entries in $\mathbf{A}, \mathbf{b}$, and $\mathbf{c}$, the size of such an instance is at most

$$(mn + m + n)(\lfloor \log_2 U \rfloor + 2).$$

In fact, this count is not exactly correct: more bits will be needed to encode "flags" that indicate where a number ends, and another starts. However, our count is right as far as the order of magnitude is concerned. To avoid details of this kind, we will be using instead the order-of-magnitude notation, and we will simply say that the size of such an instance is $O(mn\log U)$.

Optimization problems are solved by algorithms. The running time of an algorithm will, in general, depend on the instance to which it is applied. Let $T(n)$ be the *worst-case* running time of some algorithm over all instances of size n, under the bit model.

Definition 4.5 *An algorithm runs in* **polynomial time** *if there exists an integer k such that $T(n) = O(n^k)$.*

Fact: Suppose that an algorithm takes polynomial time under the arithmetic model. Furthermore, suppose that on instances of size n, any integer produced in the course of execution of the algorithm has size bounded by a polynomial in n. Then, the algorithm runs in polynomial time under the bit model as well.

The class $\mathcal{P}$: A combinatorial optimization (CO) problem is in $\mathcal{P}$ if it admits algorithms of polynomial complexity.

The class $\mathcal{NP}$: A combinatorial problem is in $\mathcal{NP}$ if for all YES instances, there exists a polynomial length "certificate" that can be used to verify in polynomial time that the answer is indeed yes.

$\mathcal{NP}$: e.g., verify the optimality of an LP solution.

Obviously, $\mathcal{P} \subseteq \mathcal{NP}$. But,

$$\mathcal{P} = \mathcal{NP}?$$

Definition 4.6 *Suppose that there exists an algorithm for some problem A that consists of a polynomial time computation in addition of polynomial number of subroutine calls to an algorithm for problem B. We then say that problem A* **reduces** *(in polynomial time) to problem B. For short, $A \overset{R}{\Longrightarrow} B$.*

In the above definition, all references to polynomiality are with respect to the size of an instance of problem A.

Theorem 4.2 *If $A \overset{R}{\Longrightarrow} B$ and $B \in \mathcal{P}$, then $A \in \mathcal{P}$.*

The above theorem says that if $A \overset{R}{\Longrightarrow} B$, then problem A is not much more difficult than problem B.

For example, let us consider the following scheduling problem: a set of jobs are to be processed on two machines where no job requires in excess of three operations. A job may require, for example, processing on machine one first, followed by machine two, and finally back on machine one. Our objective is to minimize *makespan*, i.e., complete the set of jobs in minimum time. Let us refer to this problem as (P_J).

Now, take the one-row integer program or knapsack problem that we state in the equality form: given integers $a_1, a_2, \ldots, a_n$ and b, does there exist a subset $S \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{j \in S} a_j = b$? Calling the later problem (P_K), our objective is to show that (P_K) polynomially reduces to (P_J).

For a given (P_K) we construct an instance of (P_J) wherein the first n jobs require only one operation, this being on machine one. Each has processing time a_j for $j = 1, 2, \ldots, n$. Job $n + 1$ possesses three operations constrained in such a way that the first is on machine two, the second on machine one, and the last on machine two again. The first such operation has duration b, the second duration 1, and the third duration $\sum_{j=1}^{n} a_j - b$.

Clearly, one lower bound on the completion of processing time of all jobs in this instance of (P_J) is the sum of processing times for job $n + 1$, i.e., $\sum_{j=1}^{n} a_j + 1$. Any feasible schedule for all jobs achieving this makespan value must be optimal. Suppose a subset S exists such that the knapsack problem is solvable. For (P_J) we can schedule jobs implies by S first on machine one, followed by the second operation of job $n + 1$, and complete with the remaining jobs (those not given by S). The first and last operations for job $n + 1$ (on machine two) finish at times b and $\sum_{j=1}^{n} a_j + 1$, respectively. Thus, the completion time of this schedule is $\sum_{j=1}^{n} a_j + 1$.

If, conversely, there is no subset $S \subseteq \{1, 2, \dots, n\}$ with $\sum_{j \in S} a_j = b$ our scheduling instance would be forced to a solution like: either job $n+1$ waits before it obtains the needed unit of time on machine one or some of jobs $1, 2, \dots, n$ wait to keep job $n+1$ progressing. Either way the last job will complete after time $\sum_{j=1}^{n} a_j + 1$.

We can conclude that the question of whether (P_K) has a solution can be reduced to asking whether the corresponding (P_J) has makespan no greater than $\sum_{j=1}^{n} a_j + 1$. Since (as is usually the case) the size of the required (P_J) instance is a simple polynomial (in fact linear) function of the size of (P_K), we have a polynomial reduction. Problem (P_K) indeed reduces polynomially to (P_J).

4.4 Three Forms of a CO Problem

A CO problem: F is the feasible solution set and $c : F \to \Re$ is a cost function,

$$\begin{aligned} \min \quad & c(f) \\ \text{s.t.} \quad & f \in F. \end{aligned}$$

The above CO problem has three versions:

a) Optimization version: Find the optimal solution.

b) The evaluation version: Find the optimal value of $c(f)$, $f \in F$.

c) The recognition version: Given an integer L, is there a feasible solution $f \in F$ such that $c(f) \leq L$?.

These three type of problems are closely related in terms of algorithmic difficulty. In particular, the difficulty of the recognition problem is usually a very good indicator of the difficulty of the corresponding evaluation and optimization problems. For this reason, we can focus, without loss of generality, on recognition problems.

Consider the following combinatorial optimization problem, called the **maximum clique problem**:

Given a graph $G = (V, E)$ find the largest subset $C \subseteq V$ such that for all distinct $u, v \in C$, $(v, u) \in E$.

The maximum clique problem is in $\mathcal{NP}$ or in short, Clique $\in \mathcal{NP}$.

Assume that we have a procedure *cliquesize* which, given any graph G, will evaluate the size of the maximum clique of G. In other words cliquesize solves the evaluation version of the maximum clique problem. We can then make efficient use of this routine in order to solve the optimization version.

Step 0 . $X = \emptyset$.

Step 1. Find $v \in V$ such that cliquesize$(G(v))$ = cliquesize(G), where $G(v)$ is the subgraph of G consisting of v and all its adjacent nodes.

Step 2. $X = X + v$. $G = G(v) \backslash v$. If $G = \emptyset$, stop; otherwise, go to

Step 1.

We now discuss the relation between the three variants in general. Let us assume that the cost $c(f)$ of any feasible $f \in F$ can be computed in polynomial time. It is then clear a polynomial time algorithm for the optimization problem leads to a polynomial time algorithm for the optimization problem. (Once an optimal solution is found, use it to evaluate - in polynomial time, the optimal cost.) Similarly, a polynomial time for the evaluation problem immediately translates to a polynomial time algorithm for the recognition problem. For many interesting problems, the converse is also true: namely a polynomial time algorithm for the recognition problem often leads to polynomial time algorithms for the evaluation and optimization problems.

Suppose that the optimal cost is known to take one of M values. We can then perform binary search and solve the evaluation problem using $\lceil \log M \rceil$ calls to an algorithm for the recognition problem. If $\log M$ is bounded by a polynomial function of the instance size (which is often the case), and if the recognition algorithm runs in polynomial time, we obtain a polynomial time algorithm for the evaluation problem.

We will now give another example to show how a polynomial time evaluation algorithm can lead to a polynomial time optimization algorithm by using the zero-one integer programming problem (ZOIP). Given an instance I of ZOIP, let us consider a particular component of the vector x to be optimized, say x_1, and let us form a new instance I' by adding the constraint $x_1 = 0$. We run an evaluation algorithm on instances I and I'. If the outcome is the same for both instances, we can set x_1 to zero without any loss of optimality. If the outcome is different, we conclude that x_1 should be set to 1. In either case, we have arrived at an instance involving one less variable to be optimized. Continuing the same way, fixing the value of one variable at a time, we obtain an optimization algorithm whose running time is roughly equal to the running time of the evaluation algorithm times the number of variables.

4.5 $\mathcal{NPC}$

The class co–$\mathcal{NP}$: A combinatorial problem is in co–$\mathcal{NP}$ if for all "NO" instances, there exists a polynomial length "certificate" that can be used to verify in polynomial time that the answer is indeed no.

Obviously, $\mathcal{P} \subseteq$ co–$\mathcal{NP}$. But,

$$\mathcal{P} = \text{co–}\mathcal{NP}?$$

The next definition deals with the simplest type of a reduction, where an instance of problem A is replaced by an "equivalent" instance of problem B. Rather than developing a general definition of "equivalence", it is more convenient to focus on the recognition problems, that is, problems that have a binary answer (e.g., YES or NO).

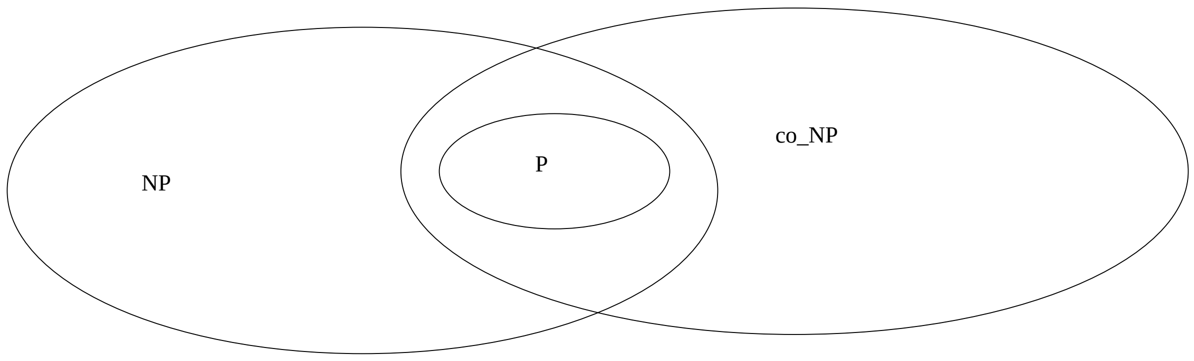

Figure 4.2: Relationships among $\mathcal{P}$, $\mathcal{NP}$ and co-$\mathcal{NP}$

Definition 4.7 *Let A and B be two recognition problems. We say that problem A* **transforms** *to problem B (in polynomial time) if there exists a polynomial time algorithm which given an instance I_1 of problem A, outputs an instance I_2 of B, with the property that I_1 is a YES instance of A if and only if I_2 is a YES instance of B. [$A \overset{R}{\Longrightarrow} B$.]*

The class $\mathcal{NP}$-hard: A problem A is $\mathcal{NP}-$hard if for any problem $B \in \mathcal{NP}$, $B \overset{R}{\Longrightarrow} A$.

Theorem 4.3 *Suppose that a problem C is $\mathcal{NP}$-hard and that C can be transformed (in polynomial time) to another problem D. Then D is $\mathcal{NP}$-hard.*

Define a set of Boolean variables $\{x_1, x_2, \ldots, x_n\}$ and let the complement of any of these variables x_i be denoted by $\bar{x}_i$. In the language of logic, these variables are referred to as *literals*. To each literal we assign a label of *true* or *false* such that x_i is *true* if and only if $\bar{x}_i$ is *false*.

Let the symbol $\vee$ denote *or* and the symbol $\wedge$ denote *and*. We then can write any Boolean expression in which is referred to as *conjunctive normal form*, i.e., as a finite conjunction of disjunctions using each literal once at most. For example, with the set of variables $\{x_1, x_2, x_3, x_4\}$ one might encounter the following conjunctive normal form expression

$$(x_1 \vee x_2 \vee x_4) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_4).$$

Each disjunctive grouping in parenthesis is referred to as a *clause*. The *satisfiability problem* is

Given a set of literals and a conjunction of clauses defined over the literals, is there an assignment of values to the literals for which the Boolean expression is *true*?

If so, then the expression is said to be satisfiable. The Boolean expression above is satisfiable via the following assignment: $x_1 = x_2 = x_3 = true$ and $x_4 = false$. Let SAT denote the satisfiability problem and Q be any member of $\mathcal{NP}$.

Theorem 4.4 *(Cook (1971)) Every problem $Q \in \mathcal{NP}$ polynomially reduces to SAT.*

Karp (1972) showed that SAT polynomially reduces to many combinatorial problems.

The class $\mathcal{NPC}$: A recognition problem A is $\in \mathcal{NPC}$ if

i) $A \in \mathcal{NP}$ and

ii) for any problem $B \in \mathcal{NP}$, $B \overset{R}{\Longrightarrow} A$.

Cook's Theorem shows SAT $\in \mathcal{NPC}$ because it can be checked easily that SAT $\in \mathcal{NP}$.

Examples of $\mathcal{NPC}$ problems: ILP, ZOIP, Clique, Vertex Packing, TSP, ΔTSP, 3-Index Assignment, Knapsack, etc.

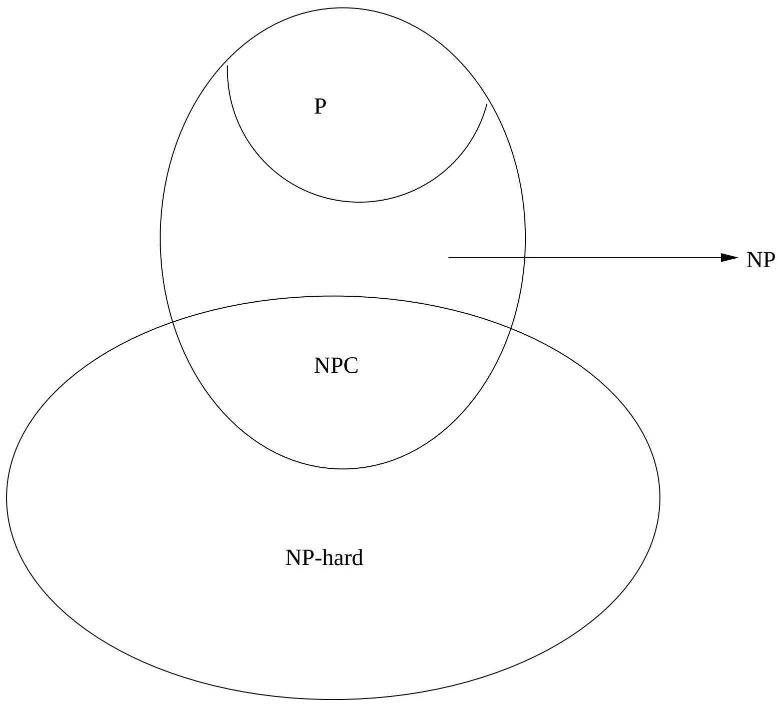

Figure 4.3: Relationships among $\mathcal{P}$, $\mathcal{NP}$, $\mathcal{NPC}$, and $\mathcal{NP}$-hard

$\mathcal{NP}$-hardness is not a definite proof that no polynomial time algorithm exists. For all we know, it is always possible that ZIOP belongs to $\mathcal{P}$, and $\mathcal{P} = \mathcal{NP}$. Nevertheless, $\mathcal{NP}$-hardness suggests that we should stop searching for a polynomial time algorithm, unless we are willing to tackle the $\mathcal{P} = \mathcal{NP}$ question.

For a good guide to the theory of $\mathcal{NPC}$, see

1979, M. R. Garey and D. S. Johnson, "Computers and Intractability: a Guide to the Theory of $\mathcal{NP}$-Completeness".

1995, C.H. Papadimitriou, "Computational Complexity".