

**Revision: Matroids and Greedy algorithms**

- Many problems can be solved using a greedy strategy, although we do not know exactly which greedy strategy works.
- Some problems can be formulated as a “maximum weight independent subset problem in matroid”.
- A known greedy algorithm works for all such problems.
- A matroid is a combinatorial structure containing a set  $S$  and a set of subsets  $\mathcal{I}$  of  $S$ .
- $\mathcal{I}$  needs to satisfy three properties to be a matroid: non-empty and finite; hereditary; and exchange.

Today: One more example of matroid, and start about NP completeness.

ALG/TITCS L04-1

**Unit-time job scheduling**

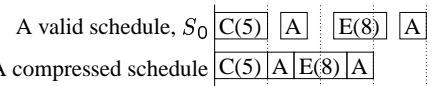
Problem:

We are given a set of jobs  $S = \{1, 2, \dots, n\}$ . Each job  $i$  is of length 1, has an integer deadline  $d_i$ , and has a value  $v_i$ . That is, each job can execute between time 0 and  $d_i$ , and need 1 unit of time to complete. At any time we can execute only one job. If we can complete a job  $i$  before its deadline, we get a value of  $v_i$ . We want to schedule the jobs so as to maximize the total value obtained.

In general, a solution is a mapping from time to  $S$ , representing the job we execute at that time.

**Simplifications**

- It seems that the problem is complicated. It is not an optimization problem to select some elements so that they have a particular property (independent), i.e. not suitable as a special case of matroid.
- But after some investigations, the problem can be made much simpler, and matroid can actually be applied.
- Suppose there is a schedule in which the set of jobs meeting deadline is  $S_0$ .
- We can always “compress out” any idle time of  $S_0$ . This never delay the time when any job is completed, so all jobs still meet deadlines.

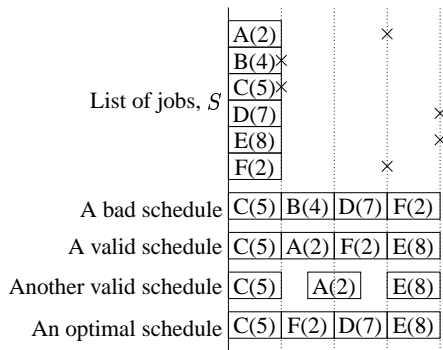


- Therefore, we need considering only schedules without idle time (until at the end of scheduling).

ALG/TITCS L04-3

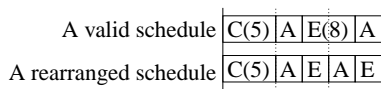
ALG/TITCS L04-4

**Example**



**Canonical schedule**

- Furthermore, we can always arrange the jobs so that jobs with earlier deadlines are always done before jobs with later deadlines.
- This involves exchanging the time when a job with early deadline is done, and the time when a job with later deadline is done.



- It might delay the time when the late-dead job is completed, but it never delay it so much that it is later than the original completion time of the early-dead job. Therefore, all jobs still meet deadlines.
- After compression and rearrangements, jobs are done in the order of deadlines. This is called the canonical schedule.

ALG/TITCS L04-5

**The selection problem**

- Given any schedule, we can construct the canonical schedule.
- For any subset of jobs to complete, it is trivial to check whether the canonical schedule meets deadline of all the jobs in the subset, and hence whether it is possible to schedule all jobs without missing deadlines.
- The problem thus becomes to find a subset of jobs to complete, so that the canonical schedule completes all jobs within deadlines.
- Now it is a problem in the form of maximum weight subset with a particular property.
- We define  $\mathcal{I}$  to be the set of subsets of  $S$  that has a canonical schedule meeting all deadlines. It remains to see whether  $(S, \mathcal{I})$  is a matroid.

ALG/TITCS L04-6

### Matroid, and feasibility of a job set

To show  $\mathcal{I}$  is a matroid, we need non-empty and finite, hereditary, and the exchange property.

- $\mathcal{I}$  is nonempty and finite. This is obvious:  $\mathcal{I}$  contains at least the empty set. It is a set of subset of a finite set  $S$ , so it must also be finite.
- $\mathcal{I}$  is hereditary. Again obvious: if a subset  $S_0 \in \mathcal{I}$ , i.e. it can be scheduled without missing deadline, a subset of  $S_0$  can also be scheduled without missing deadline, and hence the canonical schedule of the subset.

It remains to show that  $\mathcal{I}$  satisfy the exchange property. This is a bit more tricky. Suppose  $S_1$  and  $S_2$  are in  $\mathcal{I}$ , with  $|S_1| < |S_2|$ , i.e.  $|S_1| + 1 \leq |S_2|$ . We need to find a job  $J$  in  $S_2 - S_1$  such that  $S_1 \cup \{J\}$  is also in  $\mathcal{I}$ .

Idea: set  $J$  to be the latest deadline job in  $S_2 - S_1$ .

ALG/TITCS L04-7

### The exchange property

- We want to show that  $S'_1 = S_1 \cup \{J\}$  is in  $\mathcal{I}$ .
- We partition the jobs in  $S'_1$  into two sets. One includes  $J$ , and all jobs in  $S_2$  with deadline later than  $J$ . The other includes the other jobs. Suppose the first set contains  $k$  jobs, and the second set contains  $|S_1| - k + 1$  jobs.
- The second set is a subset of  $S_1 \in \mathcal{I}$ . So its canonical schedule meets all deadlines.
- Now we go back to the first set. In the canonical schedule of  $S_2$ , they are all scheduled after time  $|S_2| - k > |S_1| - k \geq |S_1| - k + 1$ . Therefore, they can all be completed after the last job completes in the canonical schedule of the second set.
- So there is a schedule for  $S'_1$  that meets all deadlines, and thus the canonical schedule for  $S'_1$  meets all deadline.  $S'_1 \in \mathcal{I}$ , as needed.

ALG/TITCS L04-8

### NP completeness: an overview

- A problem is said to be “trackable”, i.e. can be solved efficiently, if there is an algorithm to solve it in polynomial time.
- Many problem is known to be trackable. The set of trackable problems is known as  $P$ , the set of polynomial-time solvable problems.
- If we extend the definition of “computations”, we might end up with more problems that can be solved in polynomial time.
- One such extension is the allowance of “non-determinism”. This notion of computation allows an algorithm to make “guesses”, and settle on algorithms which output the correct solution only when the guess are right.
- If a problem can be solved in polynomial-time under this (strange) definition of computation, it is said to be a non-deterministically polynomial-time solvable problem. The set  $NP$  is the set of these problems.

ALG/TITCS L04-9

### NP completeness: cont'd

- We don't really know whether this extension of computation extends the number of problems that is polynomially-time solvable (in the normal computational model). In other words, we don't really know whether  $P = NP$ .
- However, there are some problems in  $NP$  that seems very difficult, and no polynomial-time algorithm is known after being posed a long time.
- Some problems in  $NP$  are very characteristic problems in  $NP$ . It has the property that if one can be solved in polynomial time, **all** problems in  $NP$  can be solved in polynomial time. We call th  $NP$ -complete problems.
- Since it is really unexpected that suddenly all problems in  $NP$  are polynomial-time solvable, most people expect that these problems are not in  $P$ .
- We show problems are  $NP$ -complete by showing that it is in  $NP$ , and it can be used to solve another problem known to be  $NP$ -complete (reduction).

ALG/TITCS L04-10

### Abstract problems

- To argue about the difficulty of problems, we need to formally define what is a problem.
- We define an abstract problem to be a binary relation on a set  $I$  of problem instances and a set  $S$  of problem solutions.
- For example, consider the MST (Minimum Spanning Tree) problem. Each problem instance is a connected graph  $G = (V, E)$ , together with a function  $w : E \rightarrow \mathbb{R}^+$ . A solution to the problem is a subset of  $E$  which forms a tree and has the minimum weight.
- Note that under this formulation, “solving a problem” is a one-step process. The whole input is given, and then the problem is solved. There is no concept of interactions, future information, etc.

ALG/TITCS L04-11

### Decision problems vs. optimization problems

- When we study  $NP$  completeness, we usually focus on decision problems.
- The solution of such problem is always either “yes” or “no” (but not both).
- For example, the decision-version of MST looks like this. Each problem instance specify a connected graph  $G = (V, E)$ , together with a function  $w : E \rightarrow \mathbb{R}^+$ , **and a threshold**  $t$ . The solution is “yes” if there is a minimum spanning tree of weight no more than  $t$ , and “no” if otherwise.
- An optimization problem asks for the “best possible threshold”. For example, the optimization version of MST is to find the minimum  $t$  so that the decision version of the problem answers “yes”, given  $G$  and  $w$ .
- Usually, if the decision problem problem can be solved in polynomial time, so does the optimization problem. (By using binary search for  $t$ .)

ALG/TITCS L04-12

### Encoding of a problem

- In order to formally define a problem and to argue about its difficulty, the encoding of a problem must be specified.
- An encoding is the representation of a problem in terms of a fixed “alphabet”.
- The problem instance is thus encoded as a string over the alphabet.
- For example, a number might be encoded as a binary string over  $\{0, 1\}$ . Other mathematical objects like pairs, (finite) sets, graphs, etc. are similarly encoded.
- We use  $\langle obj \rangle$  to denote the encoded version of an object  $obj$ .
- Discounting some very poor encoding (e.g. an unary string of 1’s for a number), most encoding performs well, even under  $\{0, 1\}$ .

ALG/TITCS L04-13

### Decision problems vs. Language

- Since the solution set of decision problems is always “yes” or “no”, we can formulate decision problems by using set notations.
- Suppose we have an abstract decision problem  $Q$ . Then the set of problem instances that has the solution “yes” forms a set  $L$ .
- We say  $L$  is the language corresponding to  $Q$ . (Therefore, a language in computational theory is just a set.)
- This makes it very convenient to formally specify a decision problem. Instead of specifying the problem, we specify the set of problem instances that gives “yes” solution.
- e.g. the decision version of MST is correspond to the (infinite) set  $L = \{ \langle (V, E), w, t \rangle : \exists T \subseteq E \text{ s.t. } T \text{ forms a tree, and } \sum_{e \in T} w(e) \leq t \}$ .

ALG/TITCS L04-14

### Some terminology about algorithms and languages

- An algorithm for a language  $L$  should thus process the input  $x$ , answer “yes” (i.e. accepts  $x$ ) if it find the input is in  $L$ , and “no” (i.e. rejects  $x$ ) if not.
- We say an algorithm decides a language  $L$  if, for each input  $x$ , it accepts  $x$  if  $x \in L$ , and rejects  $x$  otherwise, after a finite number of steps.
- If an algorithm always accepts an input  $x \in L$  and never accept an input  $x \in L$  (but it may never answer), we say the algorithm accepts  $L$ .
- We say an algorithm decide a language in polynomial time if there is a polynomial  $f(n)$ , such that the algorithm always accept or reject an input of  $n$  bits within  $f(n)$  steps.
- Note that the algorithm is allowed to take more time for longer input (larger problem). But it must be bounded by some fixed polynomial. The algorithm is not allowed to use larger polynomial when the input is larger.

ALG/TITCS L04-15

### Reduction

- We say a problem  $A$  is reducible to another problem  $B$  if, given an instance  $I_A$  of  $A$ , we can compute an instance  $I_B$  of  $B$ , so that they always has the same solution.
- If this computation can be done by a polynomial time algorithm, we say that  $A$  is polynomial-time reducible to  $B$ .
- This indicates that  $A$  is no more difficult than  $B$  when polynomial-time algorithms are in concern.
- **Be careful!** Since we actually do *more* things to solve  $A$  than to solve  $B$ , it is easy to think that  $A$  is more difficult. But the fact that we can solve  $A$  this way means that  $A$  is **no harder** than  $B$ , just the other way round! Don’t forget that there might be other ways to solve  $A$  without using  $B$ , so  $A$  might be much easier than  $B$ .

ALG/TITCS L04-16

### Example of reduction

- CYCLIC-SHIFT: Given two strings  $w$  and  $u$  of the same length, answer yes iff  $u$  is a cyclic-shifted version of  $w$ .
- SUBSTRING: Given two strings  $w$  and  $u$ , answer yes iff  $u$  is a substring of  $w$ .

Given an instance  $(w, u)$  of CYCLIC-SHIFT (say,  $(aabca, bcaaa)$ ), we can construct an instance  $(ww, u)$  of SUBSTRING (i.e.  $(aabcaabca, bcaaa)$ ).

- They always have the same solution. If the instance of SUBSTRING answers “yes”, there is a substring  $u$  within  $ww$ , the part in the second  $w$  followed by the part in the first  $w$  is exactly  $w$ , so  $u$  is a cyclic shift of  $w$ . This also works the other way round.
- Since the construction needs polynomial (in fact linear) time, we say CYCLIC-SHIFT is polynomial-time reducible to SUBSTRING.

ALG/TITCS L04-17

### Non-determinism

Now we comes to the most difficult part of the theory, since it changes the way we think what is a computation.

In a non-deterministic algorithm, we can use an expression **Guess()** which always returns either 1 or 0. The behaviour of the algorithm then depends on the returned value.

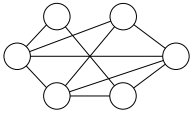
- Given an  $x \in L$ , the algorithm must answer “yes” for at least one sequence of values returned by **Guess()**. I.e. there must be some way that the algorithm accepts  $x$ .
- Given an  $x \notin L$ , the algorithm must not answer “yes” for any sequence of values returned by **Guess()**. I.e. there must be no way that the algorithm accepts  $x$ .

Note that the definition is asymmetric: it allows the algorithm to be less strict when dealing with  $x \in L$ .

ALG/TITCS L04-18

### Example problem: Hamiltonian cycle

Problem: Given a graph  $(V, E)$ , answer “yes” iff there is a permutation  $P = (v_1, v_2, \dots, v_{|V|})$  of  $V$ , such that  $(v_1, v_2), (v_2, v_3), \dots, (v_{|V|}, v_1)$  are all in  $E$ .



The non-deterministic algorithm on the right is a correct non-deterministic algorithm for this problem, requiring only  $O(n^2)$  time.

```

for i = 1 to |V|
do  $u_i \leftarrow 0$ 
for i = 1 to |V|
do  $v_i = -1$ 
  for j = 1 to |V|
  do if Guess()=1
    then if  $v_i \neq -1$ 
      then reject()
      if  $u_j \neq 0$ 
      then reject()
       $v_i \leftarrow j$ 
       $u_j = 1$ 
for i = 1 to |V| - 1
do if  $(v_i, v_{i+1}) \notin E$ 
then reject()
else accept()

```

### Another formulation: polynomial-time verification

There is another way to define the set  $NP$ . We can define them as problems that can be “verified” in polynomial time given a “certificate”:

A language  $L$  is in  $NP$  if there is an algorithm  $A$  taking the problem instance  $I$  and a string  $w$  over  $\{0, 1\}$  as input, so that

- if  $I \in L$ , there is a  $w$  with length polynomial to  $|I|$ , so that  $(I, w)$  is accepted by the algorithm;
- if  $I \notin L$ , there is no  $w$  that  $(I, w)$  is accepted by the algorithm.

Note the similarity with the other definition. Again, the algorithm is allowed to be much looser if  $I \in L$ .

ALG/TITCS L04-19

ALG/TITCS L04-20

### Example problem: Composite

As an example, consider the problem COMPOSITE, which answer “yes” if the input is a composite number, and “no” otherwise.

The verification algorithm:

```

Input:  $N$  (the number to be tested),  $c$  (the certificate)
if  $c \leq 1$  or  $c \geq N$ 
then reject()
if  $N \bmod c \neq 0$ 
then reject()
accept()

```

If  $N$  is composite, there must be at least one non-trivial factor  $f$ . Thus the verification algorithm would accept  $(N, f)$ . If  $N$  is prime, then there is no non-trivial factor, so the verification algorithm reject all  $(N, f)$ . The verification algorithm runs in linear time (i.e. linear to the number of bits of  $N$  and  $c$ ).

ALG/TITCS L04-21

ALG/TITCS L04-22

### The equivalence of the two formulations

If there is a verification algorithm, we can construct a non-deterministic algorithm in this way.

- Run Guess() some times to “generate” the certificate.
- Run the verification algorithm to check the input with the certificate.

On the other hand, if there is a non-deterministic algorithm, we can make a verification algorithm as follows.

- Replace every Guess() by a routine which shift out 1 bit of the certificate and return that bit.
- If the certificate runs out of bits, the algorithm rejects.

In either case, it can easily be shown that the meaning of non-deterministic algorithm is preserved.

### Problems in $NP$ is exponential-time solvable

It should be noted that although non-determinism gives extra power, it does not give so much power that it can solve problems that we cannot solve at all.

We can always solve a problem in  $NP$  in exponential time, as follows:

```

for  $k = 1$  to  $f(|I|)$  { Where  $f(|I|)$  is the max length of the certificate}
do for each certificate  $c$  in  $\{0, 1\}^k$ 
  do if the verification algorithm accept  $(I, c)$ 
    accept  $I$ 
reject  $I$ 

```

Since the verification algorithm runs in polynomial time (say  $g(|I|)$ ), and there are only  $2^{f(|I|)+1}$  inputs, the algorithm runs in  $2^{f(|I|)+1}g(|I|)$  time, i.e. exponential time.

Slow, but still possible.

ALG/TITCS L04-23

ALG/TITCS L04-24

### Class announcement

Next lecture will be delayed for 15 minutes due to JUPAS open day admission talks.

Assignment 1 is posted on the web. Due day is 14 days after today, at Oct 21.