CSE541 Advanced Algorithm Note

1 Background

During my exchange in WashU, I've typed some of my notes from lectures and collaborated some with online resources and my understanding. This is mainly for the course - Advanced Algorithm, as the professor doesn't have a typed note. Since I typed the notes quite rush, there maybe some mistakes or typos. I've typed the note mainly for my revision and probably future review. But I am also glad if this note can help anyone need it.

2 Overview of designing efficient algorithms

Our first goal:

1. Design a polynomial time algorithm
2. If fail, some problem is NP-hard
3. Design a pseudo-polynomial time algorithm
4. If fail, some problem is NP-hard in strong sense
5. Solve using SAT-solver
6. If fail, some problem is Σ_2^p or Π_2^p
7. Design an exponential-time algorithm

Our second goal: Design an algorithm that solve the problem approximately by using Fixed Parameter Tractable algorithm and algorithm using prediction.

2.1 Algorithm efficiency

1. Divide and conquer
2. Substitution method
3. Recursion tree
4. Master method
5. Dynamic programming
6. Use ILP, SAT solver
7. FPT, using Predictions

2.2 Lower bound

1. NPC is a lower bound
2. $\Omega(nlogn)$ for sorting
3. P, NP, coNP, etc.
4. Strong NP-hardness
5. PSPACE "and above"
6. Σ_2^p, Π_2^p, PH

2.3 Tools

1. Pseudo-polynomial is better than truly exponential
2. Idea of reduction
3. Relationship amongst classes
4. polynomial-time reduciblility
5. pseudo-polynomial transformation
6. Savitch's theorem, time and space hierarchy theorems
7. Oracles, P^{NP} and $\triangle_2^p$

3 NP-completeness

3.1 Reduction

When we reduce A to B, we are saying "if I could solve B in some models, then I could solve A in that model as well". It follows that B is at least as hard as A but A maybe easier. Hence we have A reduces to B $\Leftrightarrow A \leq_p B \Leftrightarrow$ B is at least as hard as A.

3.2 Definition of NP, NPC and NPI

P, NP, NP-Hard and NP-Complete are sets of problems, defined as follows: Let A be a decision problem.

1. P: can be solved by polynomial time algorithm
2. NP: can be verified by polynomial time algorithm
3. NP-hard: Every problem in NP is reducible to A in polynomial time

4. NP-complete: A is in NP and NP-hard

5. NP-intermediate: If P $\neq$ NP, then there exists some decision problems between NP and NP-complete problems. Otherwise, NP-intermediate problems do not exist because in this case every NP-complete problem would fall in P, and by definition, every problem in NP can be reduced to an NP-complete problem.

3.3 Ladner's theorem

Theorem 1. *If $P \neq NP$, there are problems in $NP \backslash P$ that are not NPC.*

3.4 coNP and coNP-complete

Definition 1. *$coNP := \{L | \bar{L} \in NP\}$*

Definition 2. *A language L is coNP-complete if $L \in coNP$ and $\forall A \in coNP$, $A \leq_p L$, i.e. every coNP problem is reducible to L in polynomial time.*

Remark. *We can also write $\bar{L} \in coNP$ if $\bar{L}$ has a polynomial time verifiers. i.e. given a false instance of L, we can return true.*

Theorem 2. *If L is NP-complete, then $\bar{L}$ is coNP-complete.*

The proof for the theorem is easy.

3.5 Strongly NP-completeness

Any NP-complete problem without numerical data is strongly NP-complete.

Definition 3. *NP-complete problem which remains NP-complete when all numbers in the input are bounded by some polynomial in the length of the input is called strongly NP-complete.*

3.6 Working examples

Example 1. *The COUNTFACTORS problem is defined as follows:*
Instance: *positive integers N and k*
Question: *Does N have at least k prime factors?*

Observe that the prime factorization problem is in NP since a factor $p < N$ such that $p|n$ serves as a witness of a yes instance. Set count = 0 and check if i is a prime factor from $i = 1$ to $i = N$. If count $\geq k$, return yes. Otherwise return false. Hence, COUNTFACTORS is in NP.

Similarly, if count $< k$, we can return true for the complement of the problem. Hence, COUNTFACTORS is also in coNP.

Example 2. *The SUBSET-SUM problem (Chapter 34.5.5) was shown to be NP-hard by a polynomial-time reduction from 3-CNF-SAT.*

SUBSET-SUM $= \{\langle S, t\rangle : \text{there exists a subset } S' \subset S \text{ such that } t = \sum_{s \in S'} s \}$

Remark. *As with any arithmetic problem, it is important to recall that our standard encoding assumes that the input integers are coded in binary. With this assumption in mind, we can show that the subset-sum problem is unlikely to have a fast algorithm.*

4 Time and Space complexity (PSPACE and NPSPACE)

4.1 Overview of time and space hierarchy

1. PSPACE - decision problems that can be solved by algorithms that use a polynomial amount of space
2. NPSPACE - decision problems that can be verified by algorithms that use a polynomial amount of space
3. EXPTIME - decision problems that can be solved by algorithms with exponential running time
4. NEXPTIME - decision problems that can be verified by algorithms with exponential running time
5. EXPSPACE - decision problems that can be solved by algorithms that use an exponential amount of space
6. NEXPSPACE - decision problems that can be verified by algorithms that use an exponential amount of space

4.2 More Classes

4.2.1 P, NP, coNP $\subseteq$ PSPACE

Recall that PSPACE = problems that are solved by algorithms requiring polynomial space. Clearly, P $\subseteq$ PSPACE. Consider a Turing Machine that need unit time to process single instructions. If we can solve an algorithm in polynomial time, that means we only visit polynomial space of cells. Hence, the result follows.

Moreover, NP $\subseteq$ PSPACE. Given a boolean formula with n variables. Think of a truth-value assignment as a sequence of 0's and 1's, where 0 indicates false and 1 indicates true. If there are n variables, then there are n bits in the sequence. But a sequence of bits is just a binary number. We can try all assignments by starting with 0 and adding 1 each time around a loop, since adding 1 moves to the next collection of values to try. Since SAT is in

PSPACE and SAT is NP-complete, every problem in NP is in PSPACE.

PSPACE is defined in terms of deterministic algorithms. So it is closed under complementation. Consequently, coNP $\subseteq$ PSPACE.

4.2.2 Relationship between TIME and SPACE classes

Definition 4. *TIME(f(n)) is the class of all decision (yes/no) problems that can be solved (deterministically) in time O(f(n)).*

Definition 5. *SPACE(f(n)) is the class of all decision (yes/no) problems that can be solved (deterministically) in space O(f(n)).*

By definition, a program that loops forever uses infinitely much space. Notice that we can define P and PSPACE in terms of TIME and SPACE classes.

Definition 6. $P = \cup_{k>0}\ TIME(n^k)$

Definition 7. $PSPACE = \cup_{k>0}\ SPACE(n^k)$

Remark. *We cannot use more space than time. Recall that our official model is a Turing machine. A Turing machine can only move its head to the right one cell per step. So an algorithm that uses time O(f(n)) must also use space O(f(n)). As a consequence, for every function $f(n)$, $TIME(f(n)) \subseteq SPACE(f(n))$*

4.2.3 Savitch's theorem

We need to look at space classes defined in terms of nondeterministic algorithms.

Definition 8. *NSPACE(f(n)) is the class of all decision problems that can be solved by nondeterministic algorithms using space O(f(n)).*

Theorem 3. *Savitch's theorem: $NSPACE(f(n)) \subseteq SPACE(f(n)^2)$.*

Remark. *Define NPSPACE to be the class of all decision problems that can be solved nondeterministically in polynomial space. Since converting a nondeterministic machine to a deterministic machine only squares the space, it is clear that NPSPACE = PSPACE.*

4.2.4 PSPACE-complete

Definition 9. *Decision problem A is PSPACE-complete if both of the following are true.*

1. *$A \in PSPACE$*
2. *For every $X \in PSPACE$, $X \leq_p A$.*

4.2.5 Is NP = PSPACE?

This is still a conjecture that someone believe that NP $\neq$ PSPACE.

4.3 Time and Space Hierarchy Theorem

Recall that we have the following relationship:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{NEXPTIME} \subseteq \text{EXPSPACE} = \text{NEXPSPACE}$$

EXPTIME is a very large class. It is know that P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME, and it is conjectured that all of those inclusions are proper: that is, P $\subset$ NP $\subset$ PSPACE $\subset$ EXPTIME. P and EXPTIME are defined in terms of deterministic algorithms and both in terms of time. That makes it possible to prove that P $\neq$ EXPTIME.

Time and Space hierarchy tell us that

1. P $\subsetneq$ EXPTIME
2. PSPACE $\subsetneq$ EXPSPACE
3. NP $\subsetneq$ NEXPTIME

4.4 Polynomial Hierarchy (PH)

4.4.1 Oracle definition

The polynomial hierarchy is a hierarchy of complexity classes in computational complexity theory. It is a generalization of the concept of the complexity class NP.

For the oracle definition of the polynomial hierarchy, define $\triangle_0^p := \Sigma_0^p := \Pi_0^p = P$,where P is the set of decision problems solvable in polynomial time. Then for i ≥ 0 define

$$\begin{aligned}
\triangle_{i+1}^p &:= P^{\Sigma_i^P} \\
\Sigma_{i+1}^p &:= NP^{\Sigma_i^p} \\
\Pi_{i+1}^p &:= coNP^{\Sigma_i^p}
\end{aligned}$$

where P^A is the set of decision problems solvable in polynomial time by a Turing machine augmented by an oracle for some complete problem in class A; the class NP^A and $coNP^A$ are defined analogously. For example, $\Sigma_1^p = NP$, $\Pi_1^p = coNP$ and $\triangle_2^p = P^{NP}$ is the class of problems solvable in polynomial time by a deterministic Turing machine with an oracle or some NP-complete problem.

4.4.2 Relations between classes in PH

The union of all classes in the polynomial hierarchy is the complexity class **PH**.

The definitions imply the relations:

$$\Sigma_i^p \subseteq \triangle_{i+1}^p \subseteq \Sigma_{i+1}^p$$
$$\Pi_i^p \subseteq \triangle_{i+1}^p \subseteq \Pi_{i+1}^p$$
$$\Sigma_i^p = co\Pi_i^p$$

Remark. *It is an open question whether any of these inclusions are proper, though it is widely believed that they all are.*

5 Fixed Parameter Tractable (FPT)

In the classical study of algorithms and complexity, the running time of an algorithm is stated as a function of the size of its input. In the parameterized study of algorithms and complexity, the running time of an algorithm is stated as a function of both the size of the input and a parameter which is a numerical value that depends on the input and is obtained via a parameterization.

For NP hard problem, we may need $O(2^n)$ running time while the solution maybe of size $O(n^C 2^k)$ where $k << n$. Not all problem can by solved by parameterized algorithm, problems that can be solved by this method is called fixed parameter tractable. If a problem cannot be solved by fpt algorithm, we may classify them as W[1]-HARD or W[2]-HARD. For example, Maximal Clique and Independent Set are in W[1]-HARD.

Definition 10. *A **parameterization** for a problem is a computable function that accepts as input any problem instance and returns an interger:*

$$\kappa : \{problem\ instances\} \longrightarrow \mathbb{N}$$

Remark. *While this definition is very liberal allowing any computable function to be considered a parameterization, some parameterizations are more useful than others from a pratical persepctive.*

5.1 Vertex Cover Problem

This example is from CMU 15-750 note. Consider a real world problem:

We have a set of roads and junctions. We want to place security cameras at specific junctions such that all roads are monitored.

Now, our goal is to minimize the total number of cameras placed. We can see that this is a Vertex Cover problem, which is NP-complete!

1. Approximation algorithms

- We know that we could find a 2-approximate vertex cover. However, cameras might be expensive, so a factor of 2 might still be unsatisfactory.
- If P $\neq$ NP , it is hard to approximate vertex cover to within a factor of 1.3, so we cannot expect to do better than this in general.

2. FPT algorithms

   - If we know that the minimum number of cameras that we need to place is small, we can do better.
   - We fix parameter k, and solve the following problem:
     - If OPT $\leq k$, return OPT // OPT = Optimal Vertex Cover
     - Else return Failure

5.2 Fixed Parameter Tractability

We again follow CMU 15-750 note. I personally think it is easier for beginner to understand.

Definition 11. *A fixed parameter tractable (FPT) algorithm with fixed parameter k and input size n is an algorithm with $O(f(k) \times poly(n))$ time complexity.*

Remark. *Note that:*

1. *$f(k)$ does not depend on n*
2. *$f(k)$ must not be polynomial of k, otherwise we could let $k = n$ and conclude $P = NP$.*

The motivation behind the definition could be illustrated by the vertex cover example. Suppose k = 20 and n = 10000, where k is the fixed parameter, and n is the number of roads and junctions. Consider the computational time for the follow algorithms:

1. Brute force: $2^n = 2^{10000}$
2. Try all set of k junctions: $C_{20}^{10000} = 10^{80}$
3. FPT algorithms: $2^k * n^c = 2^{20} + 10000^c = 10^{10}$

For more detail, please go check CMU 15-750 lecture 38.

5.3 Kernelization algorithms

6 Algorithms using prediction

6.1 Heurstic