CS491I Approximation Algorithms
Lecture Notes

Lan Guo

Turing Machine

A Turing machine M can be viewed as a computing device (Figure 1) provided with:

1. A set Q of internal states, including a *start* state S and an *accepting* state q_A.
2. An infinite memory, represented by an (semi-) infinite *tape* consisting of *cells*, each of which contains either a symbol in a work alphabet Γ or the special *blank* symbol λ.
3. A *tape head* that spans over the tape cells and at any moment identifies the current cell.
4. A finite control (program) δ whose elements are called *transition rules:* any such rule $((q_i, a_k), (q_j, a_l, r))$ specifies that if q_i is the current state and a_k is the symbol in the cell currently under the tape head, then a computing step can be performed that makes q_j the new current state, writes a_l in the cell, and either moves the tape head to the cell immediately to the right (if r = 1) or to the left (if r = -1) or leaves the tape head on the same cell (if r= 0). [1]

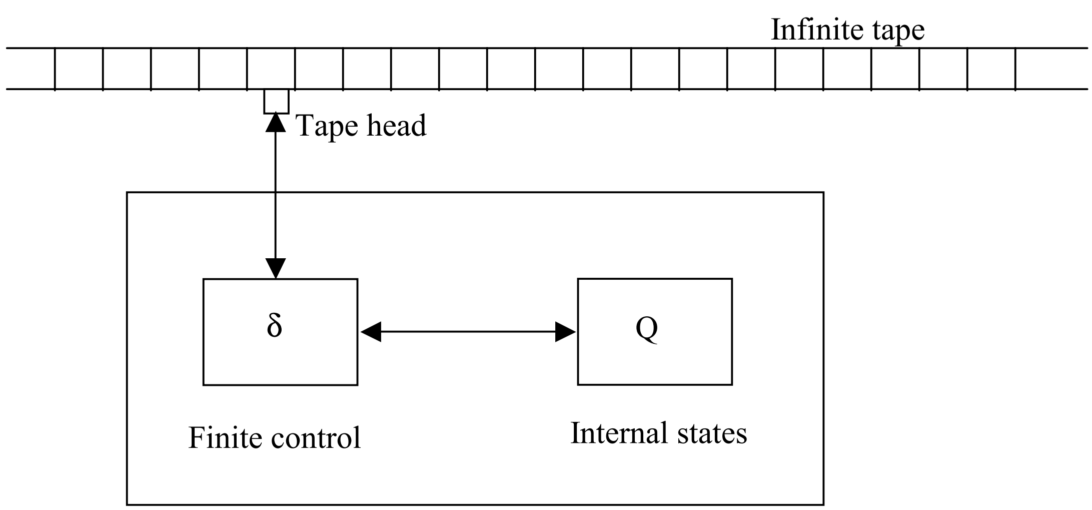

Figure 1. A Turing Machine

Turing machine: A Turing machine M is a 6-tuple M = (Q, Σ, Γ, δ, S, F) where:

1. Q is a finite set of internal states.
2. The input alphabet Σ is a finite set of symbols (not including the special symbol λ).
3. The work alphabet Γ is a finite set of symbols that includes all symbols in Σ and does not include λ.
4. The set of transition rules δ is a subset of $(Q \times (\Gamma \cup \{\lambda\})) \times (Q \times (\Gamma \cup \{\lambda\}) \times \{0, 1, -1\})$.
5. F includes $\{q_Y, q_N\}$. $S \subseteq Q$ and $F \subseteq Q$, which are the starting and the final states, respectively. [1]

Language L is *decided* by a Turing machine M if $L \subseteq \Sigma^*$, i.e. if

1. For all strings $x \in L$, M halts in q_Y
2. For all strings $x \notin L$, M halts in q_N

Language L is *accepted* by a Turing machine M, if for all strings $\in L$, M halts in q_Y.

Complement of language L, $\overline{L}$, is defined as $\overline{L} = \Sigma^* - L$, where $\Sigma^0 = \Phi$, $\Sigma^1 = \Sigma \ldots \Sigma^i = \Sigma^{i-1} \cup \{\Sigma\}$, $\Sigma^* = \cup_{i=0}^{\infty} \Sigma^i$. For example, if L is the set of graphs that contain Hamilton Path, $\overline{L}$ is the set of graphs that do not contain Hamilton Path.

Types of Problems

(D) Decision problem: the problem with answer Yes/No. For example, Does there exist $\vec{x}$, such that $A\vec{x} \leq \vec{b}$?

(S) Search problem: can you give me that $\vec{x}$?

(O) Optimization problem: maximize/minimize a function, i.e. give me the $\vec{x}$ that maximize $f(\vec{x})$.

We can use (D) for (S) and (O). One such example is SAT: given a Boolean expression with conjunction of disjunctions and an oracle to decide if this Boolean formula is satisfiable, can we find an assignment such that the result of this Boolean expression is true? It is a search problem, whose solution is based on the decision problem. We can use the oracle for decision to produce the actual assignment. If the oracle returns "yes" for the SAT instance, we know that it is satisfiable. Then, we can get the truth assignment as following. First, we can decide the value of x_1. We can put $x_1 = 0$, and $\overline{x_1} = 1$. If the oracle return "yes" indicating that it is a truth assignment, we get the value for x_1; otherwise, $x_1 = 1$ and $\overline{x_1} = 0$. We can substitute x_1 value in the original formula and get a new one. Similarly, we can get the assignment for the rest of the variables. This algorithm can be finished in polynomial time. Following procedure can solve the search problem:

```
Function Search_SAT_Assignment (F)
   If oracle (F) = "yes" then
      For each variable xi in the formula F loop
          Assign (xi = 0; x̄1 = 1) in F and get new formula F'
          If oracle (F') = "yes" then F = F'
          Else
               Assign  (xi = 1; x̄1 = 0) in F and get new formula F'
               F = F'
          End if;
       End loop;
       Return F;
     Else
         Return "not satisfiable";
     End if;
End;
```

Non-determinism

Non-deterministic Turing Machine (NDTM): NDTM is a Turing machine that an arbitrary finite number of computing steps can be applicable to a given configuration C, i.e. for transition rule δ: $(Q \times (\Gamma \cup \{\lambda\})) \rightarrow (Q \times (\Gamma \cup \{\lambda\}) \times \{0, 1, -1\})$ is a relation, instead of a (partial) function. NDTM has a witness, and we can guess a computing path and check its result in polynomial time.

We say that a string $\sigma \in \Sigma^*$ is *accepted* by a NDTM if at least one such path leads the Turing machine to halt in state q_A. One the other hand, σ is *rejected* by this NDTM if all computation paths starting from the initial configuration are rejecting [1]. Such computing paths form a tree. At the level of the leaves, it is easy to verify if this computing path is accepted or rejected.

It is generally believed that Deterministic Turing Machine (DTM) is less powerful than NDTM. Whether or not DTM is strictly less powerful is an open problem.

Time and Space Complexity

There are two ways to determine the execution cost of a Turing machine:

1. The number of computing steps performed by the machine (time complexity).
2. The amount of different tape cells visited during the computation (space complexity).

P and PSPACE:

1. The class of all problems solvable in time proportional to a polynomial of the input size: P = $\bigcup_{k=0}^{\infty} Time(n^k)$;
2. The class of all problems solvable in space proportional to a polynomial of the input size: PSPACE = $\bigcup_{k=0}^{\infty} Space(n^k)$;

NP: Set of the problems that can be decided in polynomial time by using a NDTM.

Co-NP: Set of problems whose complement can be decided in polynomial time by using a NDTM.

NP-Complete: a problem L is NP-Complete if:

1. $L \in NP$

2. $L_0 \leq L$, for any $L_0 \in$ NP. ($\leq$ is reduction relationship.)

It is generally believed that NP $\neq$ Co-NP, and P $\subseteq$ NP. The relationship between complexity classes can be pictured as Figure 2.

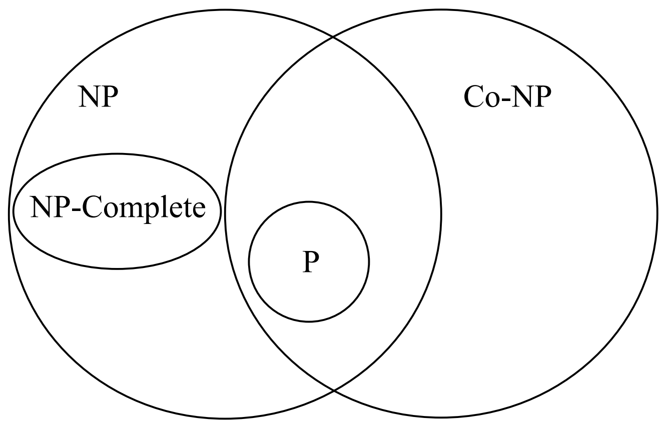

Figure 2. Relationship between complexity classes

It is easy to prove that CoP = P.

Proof: For the CoP problem, we can run the verifier that determines the P problem in polynomial time, if the verifier returns "yes" for the P problem, then the answer for the CoP should be "no"; if verifier returns "no" for the P problem, then the answer for the CoP should be "yes".

Problem and Language are interchangeable. A Turing machine decides a language.

Instance: An instantiation of parameters for a problem.

Generally, Co-NP is not easy. For instance, NON-Hamilton Path is in Co-NP. It is not easy since you need to prove that EVERY path in the given graph is not a Hamilton Path. Therefore, to verify a NON-Hamilton Path instance needs exponential time in computation. Generally speaking, "No" certificate is easy for a Co-NP problem, while "Yes" certificate is easy for a NP problem. For example, it is easy to verify that a given graph is NOT a NON-Hamilton Path instance by a Hamilton Path as a witness. In contrast, it is easy to verify that a graph is a Hamilton Path instance by a Hamilton Path as a witness.

Reducibility and Reduction

Ordering:

Given two numbers, we can compare them based on ordering as $a \leq b$, or $b \leq a$.

Given two languages, the ordering is based "hardness" or "complexity". $L_1 \leq L_2$, if L_2 is as least as hard as L_1.

Reduction: $L_1 \leq L_2$ if there exists a function f computable in polynomial time or log space, such that $x \in L_1$ iff $f(x) \in L_2$.

Note: the reduction function f has to be computable in polynomial time or log space, otherwise, we can derive an erroneous conclusion. One such example is Hamilton Path problem $\leq$ graph reachability. We can generate path for all reachable pairs, and test if there is such HM path (exponential time). Hence, graph reachability is at least as hard as HM problem. This conclusion is obviously wrong, since HM Path is hard, while graph reachability is easy (We can use either BFS or DFS in poly time). Why we reached such conclusion? The reason is that function f is not computable in polynomial time or log space. Therefore, we should have restrictions on f. It should be computable in either log space (denoted as $\leq^L$) or polynomial time (denoted as $\leq^P$).

Given $L_1 \leq^L L_2$, we know that:

1. If $L_2 \in$ P, then $L_1 \in$ P
2. If $L_1 \in$ NP, then $L_2 \in$ NP.

Closure: For a given reduction γ (polynomial time or log space), complexity class C is said to be closed with regard to γ, if $L_1 \leq^{\gamma} L_2$ and $L_2 \in C$, then $L_1 \in C$.

Hardness: A language L is said to be "hard" for complexity class C or "C-hard", if for every $L' \in C$, $L' \leq^{\gamma} L$.

Completeness: L is C-Complete, if it is C-hard and $L \in C$ (C-easy).

NP-Complete (NPC): L is NP-Complete if

1. $L \in NP$
2. For every $L' \in NP$, $L' \leq^{\gamma} L$.

Theorem: SAT is NPC.

Given conjunction of disjunctions, $(x_1 \vee x_2 \vee x_3 \ldots \vee x_k) \wedge (\overline{x_1} \vee x_2 \vee x_3 \ldots \vee x_k) \ldots$ to decide if there is a truth assignment for this Boolean formula is NP-Complete.

Proof: See [1] for detail.

Theorem: 3-SAT is NP-Complete.

3-SAT is a SAT instance that every clause contains 3 variables.

If we can reduce SAT to 3-SAT, i.e. 3-SAT ≤ SAT, we prove 3-SAT is NP-Complete (3-SAT in NP is trivial).

Proof: Let C_i be any clause of the instance of SATISFIABILITY. Then C_i is transformed into the following subformula C_i', where the y variables are new ones:

1. If $C_i = x_i$, then $C_i' = (x_i \vee y_{i,1} \vee y_{i,2}) \wedge (x_i \vee y_{i,1} \vee \overline{y_{i,2}}) \wedge (x_i \vee \overline{y_{i,1}} \vee y_{i,2}) \wedge (x_i \vee \overline{y_{i,1}} \vee \overline{y_{i,2}})$.
2. If $C_i = x_{i,1} \vee x_{i,2}$, then $C_i' = (x_{i,1} \vee x_{i,2} \vee \overline{y_i}) \wedge (x_{i,1} \vee x_{i,2} \vee y_i)$.
3. If $C_i = x_{i1} \vee x_{i,2} \vee \ldots \vee x_{i,k}$ with $k>3$, then $C_i' = (x_{i,1} \vee x_{i,2} \vee y_{i,1}) \wedge (\overline{y_{i,1}} \vee x_{i,3} \vee y_{i,2}) \wedge \ldots \wedge (\overline{y_{i,k-4}} \vee x_{i,k-2} \vee y_{i,k-3}) \wedge (\overline{y_{i,k-3}} \vee x_{i,k-1} \vee x_{i,k})$.

Specifically, if $C_i = x_{i,1} \vee x_{i,2} \vee x_{i,3} \vee x_{i,4}$, we can transform C_i to $C_i' = (x_{i,1} \vee x_{i,2} \vee \overline{y_i}) \wedge (x_{i,3} \vee x_{i,4} \vee y_i)$.

Clearly, this reduction can be done in polynomial time. In addition, it is easy to prove that the original formula is satisfiable iff the transformed formula is satisfiable. [1]

Theorem: Vertex cover (VC) is NPC.

We already proved that 3-SAT is NPC. If 3-SAT $\leq$ VC, we prove that vertex cover is NPC (VC in NP is trivial).

Vertex cover is the problem that given a graph G = (V, E) and a number k, is there a subset $V' \subseteq V$, such that $|V'| \leq k$, and for every edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$.

Proof: Let I be an instance of 3-SAT with n variables and m clauses. We can transform I to an instance S of vertex cover with 2n+3m vertices and n+6m edges as following: for each variable x_i in I, we create two vertices x_i and $\overline{x_i}$ in graph G, and put an edge between them; for each clause C_i in I, we create a triangle $a_{i1}a_{i2}a_{i3}$ in graph G, and connect each vertex in this triangle with one variable in clause C_i. For example, if $C_1 = x_1 \vee \overline{x_2} \vee x_n$, we connect a_1 with x_1, a_2 with $\overline{x_2}$, and a_3 with x_n. Figure 3 is the graph we constructed for vertex cover from 3-SAT. For every clause in 3-SAT, we pick 2 vertices in the triangle, and for every variable, we pick 1 vertex according to their form in the clause for vertex cover. Every edge is covered in Figure 3. We get k=n+2m. 3-SAT is satisfiable iff there is a vertex cover of size k = n+2m in graph G (See [2] for detail.). Obviously, this construction can be done in polynomial time.

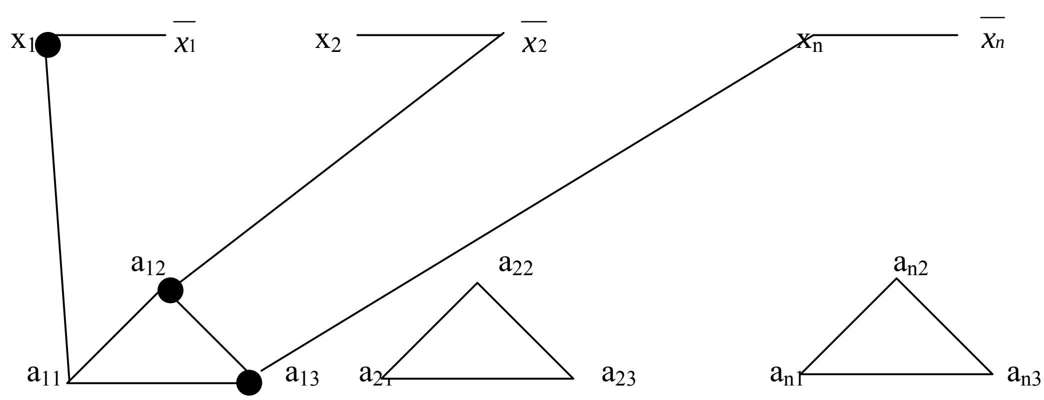

Figure 3. Vertex Cover Transformed from 3-SAT

Theorem: Integer Programming is NPC.

Proof: Let's reduce 3-SAT to Integer Programming (IP), i.e. 3-SAT $\leq^P$ IP. Once again, IP in NP is trivial.

Let C_i be a clause in 3-SAT, if $C_i = (x_1, x_2, x_3)$ in disjunction form, we construct an inequality equation as $x_1 + x_2 + x_3 \geq 1$. If variable x_i is in the negate form in clause C_i, we represent it as $1\text{-}x_i$ in the inequality equation. Therefore, we can transform an instance of 3-SAT with n variables and m clauses to an instance of IP, $A\vec{x} \geq \vec{b}$, with A (m x n), and $\vec{b}$ an integral vector, $x \in \{0, 1\}$. This transformation can be done in polynomial time.

3-SAT is satisfiable iff IP has a feasible solution.

1. If 3-SAT is satisfiable, IP is feasible. If 3-SAT is satisfiable, for each clause, there has to be at least one variable x_i is true, i.e. $x_i = 1$, or $\overline{x_i} = 1$ (if the truth assignment is in negate form). In the second case, $1\text{-}\ x_i = 1$. Without loss of generality, each clause of 3-SAT can be represented as $x_{i1} + x_{i2} + x_{i3} \geq 1$, or $1\text{-}x_{i1} + x_{i2} + x_{i3} \geq 1$, which is $-x_{i1} + x_{i2} + x_{i3} \geq 0$. In either case, it is one feasible inequality equation in the IP model. Therefore, IP is feasible if 3-SAT is satisfiable.
2. If IP is feasible, 3-SAT is satisfiable. If IP is feasible, for each inequality equation, we have $x_{i1} + x_{i2} + x_{i3} \geq 1$, corresponding to a clause C_i in 3-SAT (If one of the variable x_i in 3-SAT is in the negate form, we represent it as $1 - x_i$ in the inequality equation in the IP model.). Since $x_i \in \{0, 1\}$, there has to be at least one variable $x_i = 1$ in each inequality equation ($\overline{x_i} = 1$ if it is in negate form in 3-SAT formula). We can assign the corresponding variable x_i (or $\overline{x_i}$) in 3-SAT formula to true for each clause C_i. That satisfies each clause C_i in 3-SAT instance. Therefore, 3-SAT is satisfiable if IP is feasible.

Question: Is Linear Program $A\vec{x} \geq \vec{b}$ in NP?

We know that LP is P. So it should be in NP ($P \subseteq NP$). However, we can’t just jump to say that we can guess a solution and verify it in poly-time, and conclude that LP is NP. For continuous problem, it is hard to show that it is in NP. A way to solve it is that if A is rational, the extreme points are always rational and small. Hence, we can guess an extreme point and verify it in poly-time.

References

[1] Combinatorial Optimization, B. Korte and J. Vygen, Springer –Verlag, 2000

[2] Computers and intractability-A guide of the Theory of NP-Completeness, M. Garey and D. Johnson, Bell Laboratories Murray Hill, New Jersey, 1979