yz709-CT-sup1

1. Algorithms and problems

Question 1

1. a) Explain the formal connections between the notions of *characteristic function*, *predicate*, *decision problem*, *subset* and *language*.

   b) What is the difference between a Turing machine *accepting* vs. *deciding* a language L? How does this distinction relate to the difference between *recursively enumerable* and *decidable* languages? (Note: instead of *accepting*, *decidable* and *recursively enumerable*, you will also often see the terms *recognising*, *recursive* and *semidecidable*, respectively).

   c) What set-theoretic object (what kind of function or relation) is implemented by a Turing machine accepting vs. deciding a language?

- (a):
    - A decision problem is a problem that outputs yes or no for one for more input values. The input values can be numbers and strings over an alphabet.
    - The subset of strings for which the decision problem returns yes is a language.
    - Using Gödel numbering, we can encode any strings into a natural number, hence the input to the decision problem can be defined as natural numbers, and the set of strings that return yes will be encoded as a subset of natural

numbers S. Therefore, the decision problem algorithm is to compute the characteristic function of a set $S \subseteq \mathbb{N}$.

    - A predicate in a logical system is a statement that contains unknown variables. A logical system is decidable if there is an effective algorithm for determining whether arbitrary formulas are theorems of the logical system.

- (b):
    - Turing machine M decides a language L if, for all possible inputs x, regardless of whether $x \in L$ or not, the TM M would start from the configuration $(s, \triangleright, x)$ and reach $(\mathbf{acc}, w, u)$ if $x \in S$, and $(\mathbf{rej}, w, u)$ if $x \in S$, that is the TM M would halt on every possible input, essentially computes the characteristic function of the set/language L
        - Language L is **decidable/recursive** if it is $L(M)$ for some machine M which halts on every input, i.e., there exists a Turing machine M that decides the language L
    - Turing machine M accepts a language L if, for all $x \in L$, the TM M would start from the configuration $(s, \triangleright, x) \rightarrow_M^* (\mathbf{acc}, w, u)$ and reaches the accepting state.
        - Language L is **recursively enumerable/semi-decidable** if it is $L(M)$ for some M, i.e., there exists a machine that confirms membership for strings in L but may not halt on strings $\notin L$.
- (c):
    - A Turing machine deciding a language L: there is a total computable function $g(x)$ such that $g(x) = \begin{cases} 0 & \text{if } x \notin S \\ 1 & \text{if } x \in S \end{cases}$
    - A Turing machine accepting a language L: there is a partial computable function $g(x)$ such that $L = \{x \in \mathbb{N} | g(x) \downarrow\}$,

Comments:

Question 2

2. Say we are given a set $V = \{v_1, \ldots, v_n\}$ of vertices and a cost matrix $c : V \times V \to \mathbb{N}$. For an index $i \in [1..n]$ and a subset $S \subseteq V$, let $T(S, i)$ denote the cost of the shortest path that starts at v_1, and visits all vertices in S, with the last stop being $v_i \in S$. Describe a dynamic programming algorithm that computes $T(S, i)$ for all sets $S \subseteq V$ and all $i \leq |V|$. Show that your algorithm can be used to solve the Travelling Salesman Problem in time $O(n^2 2^n)$.

- The travelling salesman problem is given a set of vertices and the distance between each pair of vertices, find the shortest possible route that visits every city exactly once and returns to the starting point (i.e., minimum of the Hamiltonian cycle).
- Arbitrarily choose one vertex, e.g., v_1 as the starting point, and v_i as the final vertex to visit, where $i \neq 1$ and $i \leq n$, let the cost of this path $\text{cost}(i)$. Then the cycle cost would be $\text{cost}(i) + \text{dist}(i, 1)$, hence we are trying to find the minimum cost of all cycles $\min_{(i \leq n) \wedge (i \neq 1)} (\text{cost}(i) + \text{dist}(i, 1))$
- Define $\text{cost}(i) = T(V, i)$ and use the dynamic algorithm with the following inductive definition:
  - (1) Base case when we have two cities, which must be v_1 and v_i, then we output distance $\text{dist}(1, i)$;
  - (2) Inductive case when we have a subset of cities S and the ending vertex i, we choose a city $j \in S \wedge j \neq i \wedge j \neq 1$ to visit from i, then recursively calculate the cost when j is the ending point $T(S - \{i\}, j)$, for all such j, find the minimum cost $\min(T(S - \{i\}, j) + \text{dist}(j, i))$

$$\text{Base case: } T(\{1, i\}, i) = \text{dist}(1, i)$$
$$\text{Inductive case: } T(S, i) = \min_{j \in S / \{1, i\}} (T(S - \{i\}, j) + \text{dist}(j, i))$$

- Complexity analysis:
  - We have n vertices that can be chosen as the ending point
  - For each such choice, run the dynamic programming algorithm, for a j-element set $S \subseteq \{2, 3, ..., n\}$, computing one value of $T(S, i)$ requires finding the shortest of j possible paths, there are in total $\binom{n-1}{j}$ j-element subsets of $\{2, 3, ..., n\}$ and each subset gives $n - j - 1$ possible values, hence we have $j(n - j - 1)\binom{n-1}{j}$
  - Hence the total time complexity is $n \times j(n - j - 1)\binom{n-1}{j} = O(n^2 2^n)$

💡 Comments:

Question 3

3. The lectures define Turing machines to have a transition function of type $\delta : (Q \times \Sigma) \to (Q \cup \{\mathtt{acc}, \mathtt{rej}\}) \times \Sigma \times D$, where $D = \{L, R, S\}$ is the set of directions that the tape head can move in (left, right, stationary). In other literature you might find definitions that have $D = \{L, R\}$, allowing only two directions and requiring that the tape head move at each transition. Can such a Turing machine simulate the one described in lectures? Is the complexity class P affected by this distinction?

- The stationary action of the TM in the lecture $(q, w, u) \to_M (q', w', u')$:
  - Assume $w = va, \text{then } \delta(q, a) = (q', a', S)$ means stay stationary, change a to a' and $w' = va', u' = u$
  - We can simulate this using the TM in the literature with two transitions, $\delta(q, a) = (q', a', L)$ and $\delta(q', b) = (q', b, R)$, hence the tape head moves to left and back, leaving the value unchanged, configuration changes from $(q, va, u) \to (q', v, a'u) \to (q', va', u)$
- No, the complexity class P is not affected because it is independent of computation models, “the language is decidable in polynomial time” is a property of the language.

💡 Comments:

2. Polynomial-time problems

Question 1

1. Consider the language Unary-Prime in the one-letter alphabet $\{a\}$ defined by Unary-Prime $= \{a^n \mid n \text{ is prime}\}$. Show that this language is in P.

- Input: number a^n encoded in the unary alphabet a

- Algorithm: get the length of the string a^n by scanning it, i.e., $|a^n| = n$. For k from 2 to $\sqrt{n}$, check whether $k|n$, if so n is not prime, otherwise continue until the end of the loop, output n is prime.
- Time complexity: the length of a^n in unary alphabet a is n bits. Scanning the input to get the length takes $O(n)$, we then have a for loop running at most $n^{1/2}$ times for a length n, each mod operation takes $O(k)$, giving us $O(n)$ in total.
- This algorithm shows Unary-Prime can be solved in polynomial time in the length of the input, hence it is in P.

```
def unary_prime(a^n):
  n = |a^n|
  if n < 2: return False
  if n == 2: return True
  for k in range(2,sqrt(n)):
    if n % k == 0: return False
  return True
```

Comments:

Question 2

2. Suppose $S \subseteq \mathbb{N}$ is a subset of natural numbers and consider the language Unary-S in the one-letter alphabet $\{a\}$ defined by Unary-$S = \{a^n \mid n \in S\}$, and the language Binary-S in the two-letter alphabet $\{0,1\}$ consisting of those strings starting with a 1 which are the binary representation of a number in S. Show that if Unary-S is in P, then Binary-S is in TIME(2^{cn}) for some constant c.

- The language $\text{Binary-S} = \{1\{0,1\}^{n-1} | \exists m \in S. n \equiv \text{binary}(1\{0,1\}^{n-1})\}$
- Assume Unary-S is in P, there exists an algorithm that for any input x, can decide whether $x \in \text{Unary-S}$ in polynomial time $\text{Time}(|x|^c)$, where $|x|$ is the length of the input and c is some positive exponent.
- For an input $1\{0,1\}^{n-1}$, we can first scan the input from least significant bit to most significant bit, calculate the decimal representation m of binary number, e.g., $1001_2 \equiv 2^0 + 2^3 = 9_{10}$
- Then feed the input a^m into the algorithm deciding the set Unary-S, if $m \in S$, then $a^m \in \text{Unary-S}$, and $1\{0,1\}^{n-1} \in \text{Binary-S}$

- Time complexity: for an input $1\{0,1\}^{n-1}$ of length (n), scan and calculate the decimal number representation m takes $\mathrm{Time}(n)$, and in the worst-case the decimal m requires $(\log_2 m)+1$ bits in its binary representation, that is, $n=(\log_2 m)+1$, hence we have $m=2^{(n-1)}$. Thus, the time complexity of the algorithm for $\mathrm{Unary\text{-}S}$ is $\mathrm{Time}(m^c)$, and for the algorithm deciding $\mathrm{Binary\text{-}S}$, we have $\mathrm{Time}(2^{cn})$.

Comments:

Question 3

3. We say that a propositional formula φ is in 2CNF if it is a conjunction of clauses, each of which contains exactly two literals. The point of this problem is to show that the satisfiability problem for formulas in 2CNF can be solved by a polynomial time algorithm.

   First note that any clause with two literals can be written as an implication in exactly two ways. For instance $(P \vee \neg Q)$ is equivalent to $(Q \Longrightarrow P)$ and $(\neg P \Longrightarrow \neg Q)$, and $(P \vee Q)$ is equivalent to $(\neg P \Longrightarrow Q)$ and $(\neg Q \Longrightarrow P)$. For any formula φ, define the directed graph G_φ to be the graph whose set of vertices is the set of all literals that occur in φ, and in which there is an edge from literal P to literal Q if, and only if, the implication $P \Longrightarrow Q$ is equivalent to one of the clauses in φ.

   a) If φ has n variables and m clauses, give an upper bound on the number of vertices and edges in G_φ.

   b) Show that φ is *unsatisfiable* if, and only if, there is a literal P such that there is a path in G_φ from P to $\neg P$ *and* a path from $\neg P$ to P.

   c) Give an algorithm for verifying that a graph G_φ satisfies the property stated in (b) above. What is the complexity of your algorithm?

   d) From (c) deduce that 2CNF-SAT is in P.

   e) Why does this idea not work if we have three literals per clause?

- (a):
  - The set of vertices is the set of all literals that occur in φ, so the upper bound on the number of vertices would be n if all variables are distinct literals.
  - There will be an edge between vertex P and vertex Q if and only if the implication $P \Longrightarrow Q$ is equivalent to one of the clauses in φ. Hence the

upper bound on the number of edges would be m if all clauses are distinct implications between two literals.

- (b):
  - $\Leftarrow$: if there is a path in G_φ from P to $\neg P$ and a path from $\neg P$ to P, then we have two implications $P \Rightarrow \neg P$ and $\neg P \Rightarrow P$, and thus two clauses in the formula φ, which are $\{P\}$ and $\{\neg P\}$, since there are no possible assignment of P that satisfies both clauses, the formula φ is not satisfiable.
  - $\Rightarrow$: if the formula φ is unsatisfiable, then there is a literal P such that any assignment of this literal will not satisfy all clauses, and there will be a conflict between two or more clauses. Any attempt to change the variable assignment would leave some new clauses unsatisfied, this infinite failure loop with a finite number of literals indicates we have changed the value of a variable to its negation and back, i.e., we have a circular implication from P to $\neg P$ and back.
- (c):
  - Algorithm: (1) construct the implication graph G_φ from the formula φ, (2) find all pairs of vertices that are strongly connected, i.e., P is strongly connected to Q if there is an edge from P to Q and Q to P, (3) iterate through all such pairs, if both literal and its negation are in one of the pairs, then return False, if no such pairs, then return True.
    - Kosaraju's algorithm for finding strongly connected pairs of vertices: (1) use two passes of DFS on the graph, the first path adds vertices into a stack in a topologically sorted order, (2) reverse the graph edges (in the following pseudocode, we used `in_neighbour` to represent a reversed version), if we have a pair of vertices that are strongly connected, then they would be labelled or assigned by the same `root`

```
# find all strongly connected pairs of vertices
# using the Kosaraju's algorithm
def find_pairs(G):
  stack = []
  for u in G.vertices:
    visit(u,stack) # vertices in a topologically sorted order
  while stack != []:
    u = stack.pop() # the vertice added in last is popped first
    assign(u,u)

def visit(s,stack):
  if !s.visited:
    s.visited = True
```

```
    for v in s.out_neighbour:
      visit(v)
    stack.add(s)

def assign(u,root):
  if (u.assigned == None):
    u.assigned = root
    for v in u.in_neighbour:
      assign(v,root)
```

    - Complexity analysis: (1) scanning through the set of m clauses, we can construct the implication graph in $O(m)$; (2) we can find all strongly connected pairs in $O(|V| + |E|)$ using the Kosaraju's algorithm; (3) iterate through all such pairs takes $O(n/2)$ for n vertices.
        - Hence overall the algorithm runs in polynomial time.
- (d):
    - The contrapositive statement of (b) says for all literal P, G does not contain a path from P to $\neg P$ and a path from $\neg P$ to P if and only if φ is satisfiable.
    - Hence we can run the algorithm in (c) and if it returns True, then the set of clauses in φ are satisfiable, otherwise not. Since the algorithm runs in polynomial time, the problem 2CNF-SAT is in class P.
- (e):
    - Analogous to the resolution procedure: if we have two literals in a clause, then resolving these two clauses would leave us with another clause with two literals and the total number of clauses decreased by 1; but for a clause with three literals, resolving these two clauses leave us with another clause with four literals and this will grow exponentially with longer and longer clauses being build up.
    - For an implication graph in 2CNF-SAT, longer implications grow linearly, so we can string implications together in a linear chain.
    - But for 3CNF-SAT, take a clause $\{a, b, c\}$, one of the possible implication can be $\neg a \wedge \neg b \rightarrow c$, but each $\neg a$ and $\neg b$ can have another clause leading to it, e.g., $p \wedge q \rightarrow \neg a$ and $r \wedge s \rightarrow \neg b$, this leads to an exponential explosion.
    - The idea of finding a contradiction by looking for circularity in the graph contains a literal and its negation does not work in 3CNF-SAT as well, e.g.,

$\neg a \wedge b \rightarrow a$ and $a \wedge b \rightarrow \neg a$ does not lead to a contradiction if we set b as False.

Comments:
Reference

Question 4

4. A clause (i.e. a disjunction of literals) is called a *Horn clause* if it contains at most one positive literal. Such a clause can be written as an implication: $X \vee \neg Y \vee \neg W \vee \neg Z$ is equivalent to $(Y \wedge W \wedge Z \Longrightarrow X)$. HORNSAT is the problem of deciding whether a given Boolean expression that is a conjunction of Horn clauses is satisfiable.

   Show that there is a polynomial time algorithm for solving $\mathrm{HORNSAT}$.

- DPLL algorithm:
  - Unit propagation then Case split: If there is a clause containing a single positive variable literal, then set this literal to true, remove the negation of this literal from all other clauses (can achieve this using a map that stores the values of variables).
  - If at the end there is an empty clause (i.e., if the assignment of variables cannot satisfy a clause), then output False;
  - If at the end there is no clauses (i.e., if the assignment of variables can satisfy all clauses), then output True;

```
# Perform the DPLL algorithm
Input: F := A conjunction of Horn clauses
Let assignment I := {}
while I not satisfy F:
  pick an unsatisfied clause p1 ⋀ p2 ... ⋀ pk => G
  if G is a variable: I[G] = 1
  else return False
return True
Output: Boolean value indicating satisfiability of F
```

- Time complexity analysis:
  - For an input with n variables, in the worst case we don't do any unit propagation, but have a set of clauses each containing ≥ 2 literals and at least one of them is a positive literal. Every time the case split is applied, the

current formula is satisfiable. Every time when the wrong decision is made, this will be immediately detected. Hence the search tree for n variables can only contain a maximum of n nodes.

- Since the size of the search tree is polynomial in n, the running time is also polynomial in n

💡 Comments:

3. Reductions

Question 1

1. We define the complexity class of *quasi-polynomial-time* problems Quasi-P by:

$$\text{Quasi-P} = \bigcup_{k=1}^{\infty} \text{Time}\left(n^{(\log n)^k}\right)$$

Show that if $L_1 \leq_P L_2$ and $L_2 \in$ Quasi-P, then $L_1 \in$ Quasi-P.

- Since $L_1 \leq_p L_2$, we have a reduction function f that reduces a language L_1 into a language L_2, and f is computable by an algorithm running in polynomial time $\text{Time}(n^{k_1})$ for some constant k.
- The decision procedure g_2 for L_2 is in quasi-polynomial-time $\text{Time}(n^{(\log n)^{k_2}})$, hence we can compose a decision procedure for L_1 on input w: (1) first compute $f(w)$; (2) run decision procedure g_2 on $f(w)$, the total running time is $O(n^{k_1} + (n^{k_1})^{(\log n)^{k_2}}) = O(n^{(\log n)^{k_3}})$

💡 Comments:

Question 2

2. In general, k-colourability is the problem of deciding, given a graph $G = (V, E)$, whether there is a colouring $\chi : V \to \{1, \ldots, k\}$ of the vertices such that if $(u, v) \in E$, then $\chi(u) \neq \chi(v)$. That is, adjacent vertices do not have the same colour.
    a) Show that there is a polynomial time algorithm for solving 2-colourability.
    b) Show that, for each k, k-colourability is reducible to $(k + 1)$-colourability. Does this, together with part (a), mean that 3-colourability is also in P?

- (a):
    - 2-colourability decision procedure with BFS search: (1) pick one arbitrary vertex, give it a colour, (2) colour the neighbours of this vertex a different colour until we coloured all or encountered a conflict, (3) if all vertices are coloured without a conflict, then we are done.
        - This decision procedure also forms a bipartite graph where vertices are classified into two different groups and vertices inside the same group should not be connected by an edge.

```
def colour_graph(s):
  q = []
  s.colour = 0
  s.visited = True
  q.enqueue(s)
  while (q != []):
    v = q.dequeue()
    for u in v.neighbours:
      if (u.visited && u.colour == v.colour): return False
      if (!u.visited):
        u.colour = 1 - v.colour
        u.visited = True
      q.enqueue(u)
  return True
```

    - The time complexity analysis: input n vertices, we will only colour each of them once and visit each edge once, hence the time complexity would be $O(|V| + |E|)$ where $|V| = n$, since $E \leq n^2$, we have a polynomial-time algorithm.
- (b):
    - If $L_1 \leq_p L_2$, i.e., L_1 is reducible to L_2, then there is a function f that can reduce a language L_1 into L_2, and f is computable by an algorithm running in polynomial time $\text{Time}(n^{k_1})$ for some constant k.
        - If L_2 belongs to a class C, then L_1 also belongs to a class C.

- For a k-colour graph, we can add a vertex that connects to every other vertex via an edge, hence this new vertex needs to have a different colour, i.e., a $(k+1)^{th}$ colour, thus turning the k-colour graph into a $(k+1)$-colour graph. This algorithm can run in a polynomial-time of the number of vertices n.
- No, the 3-colourability decision problem is not in P class. Because 2-colourability can be reduced to 3-colourability, then if 3-colourability is in class C, then 2-colourability is in class C, not the converse.

Comments:
I have a question: 3-colourability is in class NP-complete, but 2-colourability is in class P, but from the above lemma (L_2 is in class C, then L_1 is in class C), 2-colourability is in NP-complete, then what is the relationship between P and NP-complete, does it relates with $P = NP$ million-dollar problem?