CS3052 Practical 3

170011474

2020 May

1 Q1 Computability and Decidability

(a)

A computational problem is a function that maps ASCII strings to sets of ASCII strings, which can be denoted by a letter like F or G. For example, a computational problem F is defined as "find words starting with a". If the input is "she ate an apple" then the output is a set of strings containing "ate" and "an".

(b)

1. Suppose F is a computational problem on the ASCII alphabet, and P is an Python program. P solves or computes F if $P(I) \in F(I)$ for all inputs I. F is computable if there exists some program that computes it. Otherwise, F is uncomputable.
2. A decision problem always returns "yes" or "no". If F is a decision problem solved by P, then P decides F. F is decidable if there exists some program that decides it. Otherwise, F is undecidable.
3. `YesOnString` is uncomputable and undecidable, and `HasShortPath` is computable and decidable. `Multiply` is computable but neither decidable nor undecidable (because it is not a decision problem).

(c)

If `YesOnString` exists, we create `YesOnSelf` as below:

```
from yesOnString import yesOnString
def yesOnSelf(progString):
    return yesOnString(progString, progString)
```

We can also create `NotYesOnSelf`:

```
from yesOnSelf import yesOnSelf
def notYesOnSelf(progString):
    val = yesOnSelf(progString)
    if val == 'yes':
        return 'no'
    else:
        return 'yes'
```

From above, we can know

1. `notYesOnSelf` **accepts** `progString` if `yesOnSelf` rejects `progString`, so `progString` **rejects** itself.

2. `notYesOnSelf` **rejects** `progString` if `yesOnSelf` accepts `progString`, so `progString` **accepts** itself.

If we run `notYesOnSelf` on itself (the `progString` is the description of itself), then

1. it **accepts** if `notYesOnSelf` **rejects** itself.

2. it **rejects** if `notYesOnSelf` **accepts** itself.

This is a contradiction so `notYesOnSelf` does not exist and so does `yesOnSelf`. Therefore, `yesOnString` cannot exist as well, so it is undecidable. Otherwise, `notYesOnSelf` can still be constructed.

(d)

```
def F(inString):
    return "TTT"

def G(inString):
    return "C"
```

Let S be the set of functions that map ASCII strings to ASCII strings and return strings containing a "TTT" on all inputs. Note that S includes at least one computable function (the constant function `F` returning "TTT") and excludes at least one computable function (the constant function `G` returning "C"). Hence the conditions of Rice's theorem are fulfilled since the property is non-trivial. Therefore, we conclude that `ContainsTTTOnAll` is uncomputable.

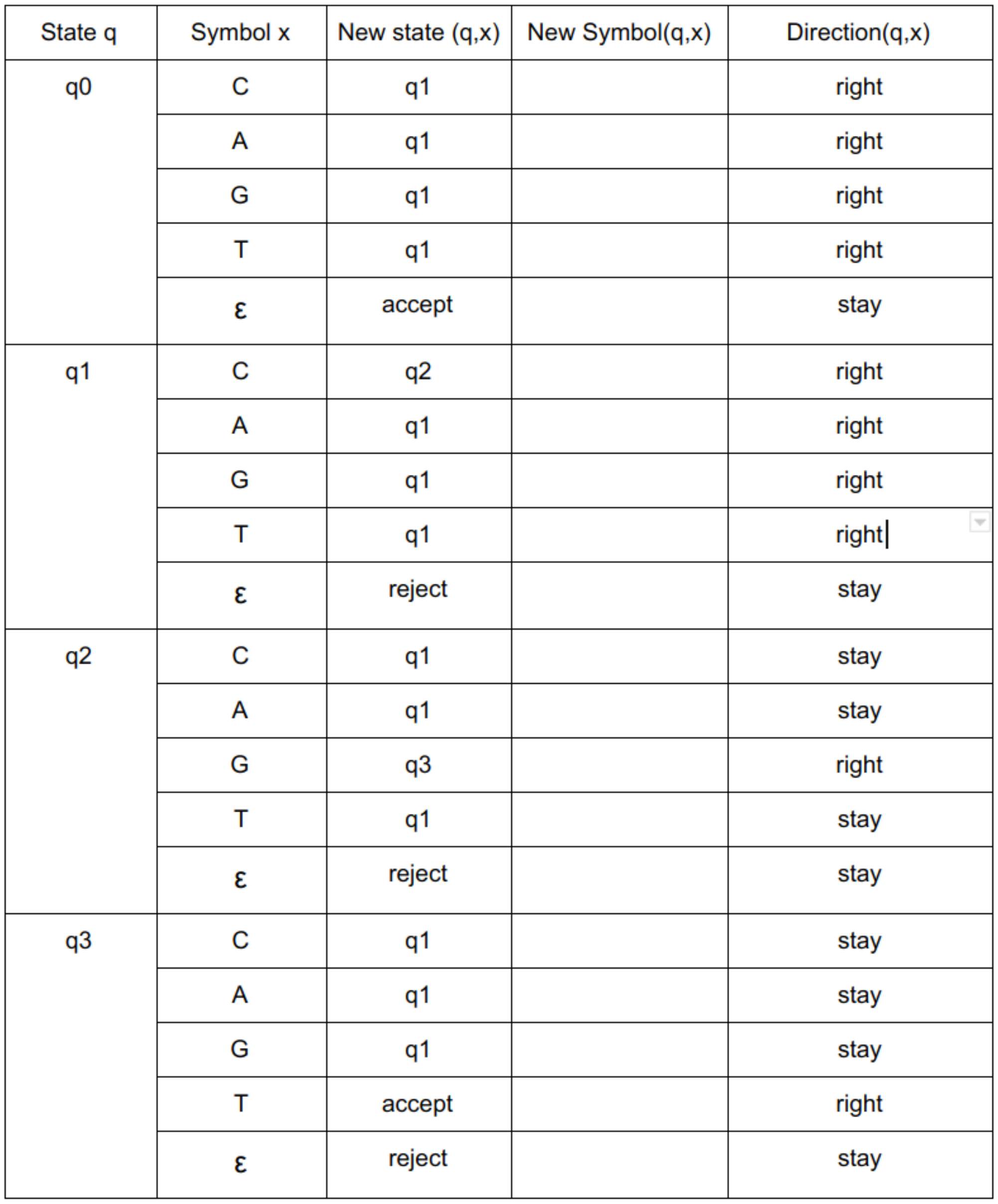
(e)

It is uncomputable.

We will show `YesOnAll` $\leq_P$ `ComputesLength`, which will prove `ComputesLength` is uncomputable because `YesOnAll` is already known to be uncomputable. Let P be an arbitrary input to `YesOnAll`. It is easy to construct P', an altered version of P, that first computes $v = P(I)$ and then returns the length of I (by `len(I)` in python) if and only if $v = "yes"$. By construction, `YesOnAll(P)` can be computed via `ComputesLength(P')`, and the reduction is complete.

2 Q2 Turing Machines

(a)



State q	Symbol x	New state (q,x)	New Symbol(q,x)	Direction(q,x)
q0	C	q1		right
	A	q1		right
	G	q1		right
	T	q1		right
	ε	accept		stay
q1	C	q2		right
	A	q1		right
	G	q1		right
	T	q1		right
	ε	reject		stay
q2	C	q1		stay
	A	q1		stay
	G	q3		right
	T	q1		stay
	ε	reject		stay
q3	C	q1		stay
	A	q1		stay
	G	q1		stay
	T	accept		right
	ε	reject		stay

Figure 1: Transition Table

(b)

Recursion Theorem

Let T be a Turing Machine (TM) that computes a function from $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$. There exists another TM R, where for every w, $R(w) = T(< R >, w)$. In other words, there is an equivalent Turing Machine that computes the same result given just w, but we can program T as if it had access to R's description.

Use Recursion Theorem to Analyse Decidability

Let $A_{TM} = \{< M, w > \mid$ where M is a Turing Machine and M accepts w $\}$. This is undecidable, proved by below:

- Assume A_{TM} is decidable and let H be the TM that decides A_{TM}. So

$$H(< M, w >) = \begin{cases} \text{accepts, if M accepts w} \\ \text{rejects, if M rejects w} \end{cases}$$

- Using H to construct a new TM D, try to run D and find a contradiction so H does not exist. The input to D is $< M >$ which is the description of a TM.
    1. Run H as a function;
    2. Ask whether machine M would accept if given a description of **itself** as an input.
    3. Do the opposite: D rejects when H accepts and D accepts when H rejects as below:

$$D(< M >) = \begin{cases} \text{accepts, if M rejects } < M > \\ \text{rejects, if M accepts } < M > \end{cases}$$

- Now run D on itself

$$D(< D >) = \begin{cases} \text{accepts, if D rejects } < D > \\ \text{rejects, if D accepts } < D > \end{cases}$$

- So we can know that D rejects and accepts at the same time, which is a contradiction.

(c)

When to simulate a single-tape turing machine M by a three-tape turing machine with only one head M' (also called three-track turing machine), all tapes except the first tape are ignored by M'. For example, we can fill all tapes except the first one with blank symbols. The tape alphabet of M is equivalent to M' consists of an ordered pair. The input of M' can be identified as an ordered pair [C,B,B].

In the 3-track turing machine, the transition function $\delta'(q_i, < a_1, a_2, a_3 >) = (q_j, < b_1, b_2, b_3 >, L/R)$, so the dimension is still $16 * 7 = 112$ where each state has 7 kinds of inputs: CBB, ABB, GBB, TBB, xBB, yBB, BBB, and 3 kinds of outputs: xBB, yBB, BBB. Therefore, the transition table of M' is similar to the one of M except the last 2 cells are all blank B.

(d)

The assumption is that the input w is finite. Firstly, define $< M, q, w >$ where M is a TM, q is one of states of M, and w is a input word over the input alphabet of M. Define a problem REACHABLE $= \{< M, q, w >$: M will reach q on input $w\}$. The question is: is REACHABLE decidable?

- Assume REACHABLE is decidable. Assume there exists a TM T that given a TM M, its state q and an input w as its inputs, it will **accept (halt in this case)** the input if and only if M will reach the state q on input w.
- Recall that $A_{TM} = \{< M, w > |$ where M is a Turing Machine and M accepts w $\}$. Construct a new TM T' such that given input $< M, w >$ where M is a TM and w is an input for M, T' will change the input to $w' =< M, q_{halt}, w >$ where q_{halt} is the unique accept state of M, and then simulate T' on input w'. We see that T' is a decider for A_{TM}.
- Since A_{TM} is undecidable as proved above. This contradiction shows that our assumption `REACHABLE is decidable` is wrong, which means this problem cannot be decidable.

(e)

Let $F = \{< M >$: M is a TM which stops for every input after at most 1000 computation steps $\}$. F is decidable.

If M stops after at most 1000 steps, then only the cells 0, 1, ..., 999 on the tape are significant. Then it suffices to simulate the M on all input strings of the form $x \in \Sigma^{1000}$, of which there are a finite number of possible input strings.

- If any of those simulations fail to enter a halting state by the 1000^{th} transitions, this indicates that they do not stop within the first 1000 steps.
- If all of these simulations halt by the 1000^{th} transitions, then M halts within 1000 steps on all inputs of any length (where the substring of length 1000 is all that it operates on).

3 Q3 Finite Automata

(a)

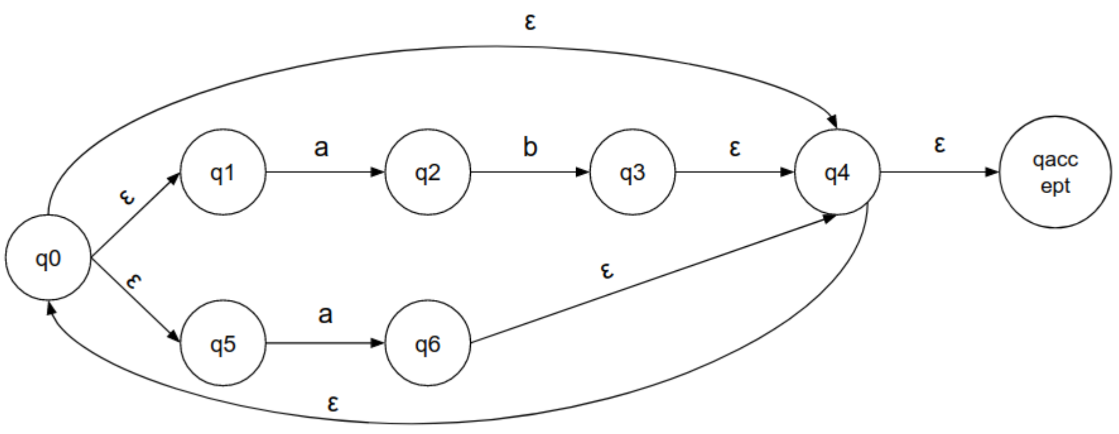


Figure 2: NFA for $(ab \cup a)^*$

(b)

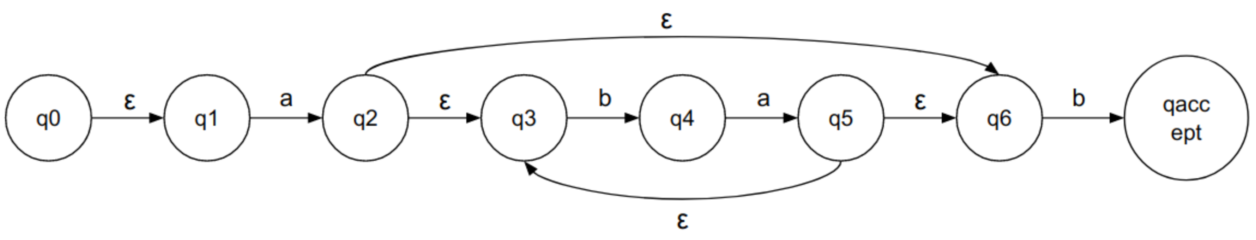


Figure 3: NFA for $a(ba) * b$

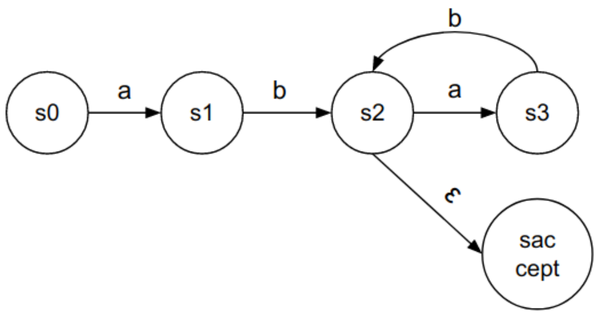


Figure 4: DFA for $a(ba) * b$

The s_0 includes q_0, q_1. s_1 includes q_2, q_3, q_6. s_2 includes q_4, q_accept. s_3 includes q_3, q_5.

(c)

If A is a regular language, then A has a pumping length P such that any string S where $|S| \geq P$ may be divided into 3 parts $S = xyz$ such that the following conditions must be true:

1. $xy^iz \in A$ for every $i \geq 0$
2. $|y| > 0$
3. $|xy| \leq P$

To proof irregularity, using following steps to get a contradiction:

1. Assume that A is regular and it has a pumping length P. So All strings that longer than P can be pumped $|S| \geq P$.
2. Now find a string S in A such that $|S| \geq P$.
3. Divide S into xyz. Show that $xy^iz \notin A$ for some i.
4. Consider all ways to divide S into xyz. Show that none of these can satisfy all the 3 pumping conditions at the same time.
5. So S cannot be pumped == CONTRADICTION.

Proof:

- Assume that F is regular. Then it must have a pumping length P.
- Choose a string S from F, for example, $S = C^PAGTC^PAGT$, which means the P number of Cs followed by AGT and concatenate it with itself. The first half and the second half are exactly the same, so S is in the F. $|S| \geq P$.

$$S = \underbrace{CCCC...C}_{\text{P Cs}}\underbrace{AGT}_{\text{1 AGT}}\underbrace{CCCC...C}_{\text{P Cs}}\underbrace{AGT}_{\text{1 AGT}}$$

- Divide S into xyz. Since condition 3 states that $|xy| \leq P$, we can only have limited ways to divide it. No matter what dividing strategy we choose, a feature of all strategies is that z would have 0 or more than 0 number of letters from the first half. That is, when $|xy| = P$, z is the second half and $|z| = P + 6$. When $|xy| = P - 1$, $|z| = P + 7$. Because the first P letters only contain C, x and y only contains C .
- When $i = 2$, so we have xy^2z. The new String:

$$S' = \underbrace{CCCC...C}_{x=|x|\text{ Cs}}\underbrace{CCCC...C}_{y_1=|y|\text{ Cs}}\underbrace{CCCC...C}_{y_2=|y|\text{ Cs}}\underbrace{\underbrace{AGT}_{\text{1 AGT}}\underbrace{CCCC...C}_{|x|\text{ Cs}}\underbrace{CCCC...C}_{|y|\text{ Cs}}\underbrace{AGT}_{\text{1 AGT}}}_{\text{z}}$$

  where $|xy| = P$.

- We can see that

$$S' = \underbrace{CCCC...CCCCC...CCCCC...CAGT}_{\text{the first half}}\underbrace{CCCC...CCCCC...CAGT}_{\text{the second half}}$$

  and the first half is not same as the second half, so $S' \notin F$ when $i = 2$. Since it fails condition 1, we conclude that F is irregular.

(d)

Using a simple version of definition of pumping from lecture notes: Let L be a language and let S be a string in L. Fix an integer $N > 0$. We say that S can be pumped before N in L if S has a nonempty substring R with the following properties:

1. R finishes before or at the N^{th} symbol of S.
2. Replacing R wit any number of copies of itself(including 0 copies) results in a string that is still a member of L.

Let $S = (GT)^N$. S can be pumped and the pumpable string must be in the first N symbols, so it is $(GT)^k$ for some $k \geq 1$. Pumping an extra copy of that substring gives $S' = (GT)^{k+N}$ and $S' \in L$ in this case. Therefore, this choice would not lead to a contradiction.

A choice $G^N T^N$ would lead a contradiction.

- Assume $G^N T^N$ is regular.
- $G^N T^N$ is infinite so the pumping lemma applies. Thus there exists some cutoff N such that all $S \in G^N T^N$ with $|S| \geq N$ can be pumped before N. This means there is a nonempty substring R in the first N symbols of S, such that S can be pumped with R.
- We do not know exactly what R is, but we know that it must consist of one or more G symbols - let's suppose it consists of kG's, where $k \geq 1$.
- If we pumps S with on extra copy of R, we get the new string $S' = G^{N+k}T^N$. But S' is not a member of $G^N T^N$, contradicting the fact that S can be pumped.

Since $G^N T^N$ is a special case of *sameGT* problem, then the language L is non-regular since not all of its strings can be pumped.

4 Q4 Complexity Classes

(a)

Verifier

Let F be a computational problem. A verifier for F is a program V with the following properties:

1. V receives 3 string parameters: an instance I, a proposed solution S, and a hint H.
2. V halts on all inputs, returning either "correct" or "unsure".
3. Every positive instance can be verified. If I is a positive instance of F, then $V(I, S, H)$ = "correct" for some correct positive solution S and some hint H.
4. Negative instances can never be verified. If I is a negative instance of F, then $V(I, S, H)$ = "unsure" for all values of S and H.
5. Incorrect proposed solutions can never be verified. If S is not a correct solution (i.e. $S \notin F(I)$), then $V(I, S, H)$ = "unsure" for all H.

Polytime Verifier

Suppose $V(I, S, H)$ is a verifier for a computational problem F. We say that V is a polynomial-time verifier (or polytime verifier) if the running time of V is bounded by a polynomial as a function of n, the length of I.

PolyCheck

The complexity class Polycheck consists of all computational problems that have polytime verifiers. It is identical to `NPoly`. Examples: Packing problem and SubsetSum problem.

Nondeterministic Turing Machine

A non-deterministic Turing machine can be formally defined as a 6-tuple $(Q, X, \Sigma, \delta, q_0, B, F)$ where Q is a finite set of states, X is the tape alphabet, Σ is the input alphabet, δ is a transition function, q_0 is the initial state, B is the blank symbol, F is the set of final states.

In a Non-Deterministic Turing Machine, for every state and symbol, there are a group of actions the TM can have. So, here the transitions are not deterministic. The computation of a non-deterministic Turing Machine is a tree of configurations that can be reached from the start configuration.

An input is accepted if there is at least one node of the tree which is an accept configuration, otherwise it is not accepted. If all branches of the computational tree halt on all inputs, the non-deterministic Turing Machine is called a Decider and if for some input, all branches are rejected, the input is also rejected.

NPoly

The complexity class Npoly consists of all computational problems that can be solved by a nondeterministic, polynomial-time program. It is identical to `PolyCheck`. For examples, SubsetSum problem.

(b)

The halting problem has a verifier. If it does, then the verifier program V would take I as input which may be a string ”`progString`; `inString`”. The solution S could be ”yes” or ”no”, and the hint H is the number of steps to halt.

```
def verifyHalting(I, S, H):
    if S == "no":
        return "unsure"
    (progString, inString) = I.split(";")
    h = int(H)

    # pseudo code here
    run P(I) in h steps
    if P(I) halts within those steps:
        return "correct"
    else:
        return "unsure"
```

Since in this function, we can check after a given computational steps if P halts within those steps. If so, then it means the given hint and the given solution is correct. Otherwise, it returns unsure.

(c)

Assume that I is in the format ”n;k”, and S is ”yes” if no factor is found in a range, or a string representation of a factor to be verified. H is used as a list containing prime factors of n. Assume that `isPrime` takes an integer as its input, returns ”yes” if it is a prime number, otherwise ”no”.

```
def verifyNotFactorInRange(I, S, H):
    # reject excessively long solutions or hints
    if len(S) > len(I) or len(H) > len(I):
        return "unsure"
    if S == "no":
        return "unsure"

    # get the problem
    (n, k) = I.split(";")
    n = int(n)
    k = int(k)
    for i in H:
        if isPrime(i) == "no":
            return "unsure"

    if min(H) > k:
        return "correct"
    else:
        return "unsure"
```

Since H is the prime factorization of n, if the least factor is out of the range, then it means that there is no factor in the range. Otherwise, there is so return "unsure" because the hint and solution is wrong. Before that, we use a `for` loop to check that every factor in H is prime. Only in this way, we can make sure that n cannot be divided into smaller integers so the prime factorization is correct and unique.

As we know from lecture slides, `len()`, `min()` and `split()` are all bounded by the length, so they are in P. Besides, the `for` is bounded by the size of the list so it is still in P. The overall verifier is in polynomial time.

(d)

```
import random
def canPartition(nums):
    sum = sum(nums)
    target = 0

    if sum \% 2 == 0:
        target = sum / 2
    else:
        return "no"
    return helper(nums, target)

def helper(nums, target):
    if target == 0:
        return "yes"
    if target < 0 or len(nums) == 0:
        return "no"

    n = len(nums)
    threads = create n threads
    for each thread:
        val = random.choice(nums)
        nums.remove(val)
        target -= val
        thread works with helper(nums, target)
    if there is one thread returns "yes" then returns "yes"
    else returns "no"
```

The function takes a list of integers as inputs and check if the sum is even first. Multiple threads would invoked for different choices of new element. If and only if at least one of them has an accepted condition, ”yes” is returned. Otherwise, no partition pattern is found.

5 Q5 NP Complete Problems

(a)

Let C be a decision problem in NP. We say C is **NP-complete** if, for all NP problem $D \in NP$, $D \leq_P C$.

1. $D \leq_P C$ means D is polynomial-time reducible to C. It means that there exists a **polytime** program C that converts instances of D into instances of C. For example, in SAT $\leq_P$ 3-SAT problem, the converter is bounded. Since the number of splits needed is dependent on the length of the formula. So it is in $O(n)$. Besides, each split needs $O(n)$ to go over the length of the current clause. So altogether $O(n^2)$ which is in polynomial time.

2. It maps positive instances of D to positive instances of C, and negative instances of D to negative instances of C. For example, in 3-SAT $\leq_P$ Clique, assume that the instance of 3-SAT has k clauses. If the instance of 3-SAT is satisfiable, then the converted instance of Clique must have a clique with size k. If the instance of 3-SAT is negative, then the instance of Clique does not have a clique with size k.

(b)

A computational problem F is NP-hard if there exists some NP-complete problem C with $C \leq_P F$. This means F is at least hard as C. To prove a problem is NP-hard, there are 3 main methods.

By Restriction

If a problem Q contains a known NP-Complete problem as its subset, then Q is NP-hard. We restrict the inputs of the problem Q to gain a subset problem of Q. And if the subset problem is NP-Complete, Q is proven to be NP-hard. For example, the `Partition` problem is a subproblem of `SubsetSum` problem, since they are the same when the target subset sum is half of the sum of input integers. So the `Partition` is only a special case of `SubsetSum`.

By Local Replacement

Local replacement means making local changes to the structure. For example, SAT $\leq_P$ 3-SAT requires modification of the original formula structure to split until all clauses contain no more than 3 literals.

By Component Design

Build (often complicated) parts of an instance with certain properties, called components or gadgets. Afterwards, glue them together in such a way that the proof works. For example, in 3-SAT $\leq_P$ 3-Coloring, OR-gadgets can be made for clauses and they are glued together for an entire formula's satisfiability.

(c)

Hamilton Circuit $\leq_P$ Hamilton Path

Given a graph G of which we need to find Hamiltonian Cycle, we go through all the edges one by one. For each edge (u, v) (hoping that it will be the last edge on a Hamiltonian circuit) we create a new graph by deleting this edge and adding vertex x onto u and vertex y onto v.

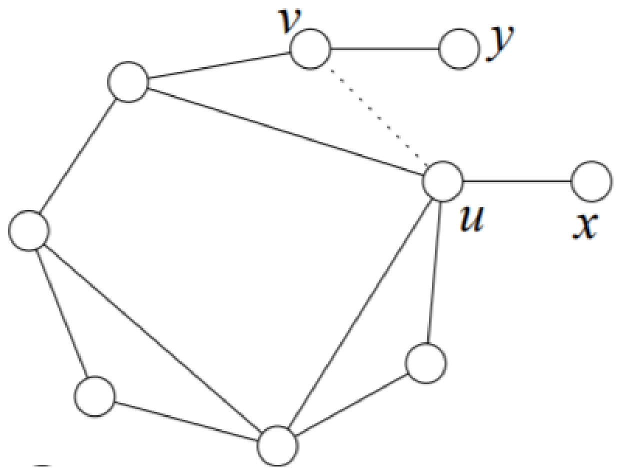


Figure 5: Hamilton Circuit $\leq_P$ Hamilton Path

Polynomial Time Proof Based on the original graph and the converter only needs to add 2 vertices, so it takes $O(1)$ which is in polynomial time.

Correctness Proof If G has a Hamilton circuit, then there must be a Hamilton path (simply delete an edge on the cycle).

Let the resulting graph be called G'. Then we invoke our Hamiltonian Path subroutine to see whether G' has a Hamiltonian path. If it does, then it must start at x to u, and end with v to y (or vice versa). Then we know that the G had a Hamilton circuit (starting at u and ending at y). If this fails for all edges, then we report that G has no Hamiltonian cycle. Thus, if there is a Hamilton path, there must be a Hamilton circuit.

Hamilton Path $\leq_P$ Longest Path

Given an instance G of Hamiltonian Path, we create an instance (G', K) of Longest Path as follows: Take $G' = G$ and set $K = |V| - 1$ where $|V|$ is the number of vertices. Then there exists a simple path of length K in G' if and only if G contains a Hamiltonian path.

Polynomial Time Proof Since we can directly use the original graph, so there is no extra work to do and the converter problem runs in polynomial time.

Correctness Proof If the graph G contains a Hamilton path, it has a path which visits each vertex exactly once. This path is also a longest path with length $K = |V| - 1$, since it does not visit a vertex twice. Therefore, the graph G' also contains a longest path with length K.

If the graph $G' = G$ contains a longest path with length $K = |V| - 1$, similarly, there is no duplicate visited vertices and all vertices are included, so G also contains a Hamilton path. Therefore, Hamilton Circuit $\leq_P$ Hamilton Path $\leq_P$ Longest Path so **Hamilton Circuit $\leq_P$ Longest Path**.

(d)

Exact Cover by 3-sets $\leq_P$ Exact Cover by 4-sets

The number of sets in the solution is $m = |X|/k$. To do reduction, construct a new set $X' = X$ and add m new elements into X'.

Polynomial Time Proof The converter only needs to add m new elements which is bounded by the size of X, so it is in polynomial time.

Correctness Proof If there is an exact cover by 3-sets in X, then assume the solution set $S = \{\{x_1, x_2, x_3\}, \{x_4, ..., x_6\}, ..., \{x_{3(m-1)+1}, ..., x_{3m}\}\}$. If m new elements are added, then the solution of 4-sets for X' is S' where for each 4-set in S', allocates one element of m new elements (so each set in S' has 4 elements now). In this way, original $3m$ elements are covered and m new elements are also covered by them after the allocating process. Therefore, there is also an exact cover by 4-sets in X'.

If there is an exact cover by 4-sets in X', then assume the solution set $S' = \{\{x_1, x_2, x_3, x_4\}, ...\}$. Let $X = X'$. Each set in S' removes 1 element and those removed $1 \times m = m$ elements are also removed from X. In this way, S' has m 3-sets which also cover X after removing process. Therefore, there is an exact cover by 3-sets in X and the solution set $S \subseteq S'$.