CS/ECE-374-B: Algorithms and Models of Computation, Spring 2024

Midterm exam 3 – April 25, 2024

- You can do hard things! Grades do matter, but not as much as you may think, but then life is uncertain anyway, so what.
- **Don't cheat.** The consequence for cheating is far greater than the reward. Just try your best and you'll be fine.
- **Please read the entire exam before writing anything.** There are 6 problems and most have multiple parts.
- This is a closed-book exam. At the end of the exam, you'll find a multi-page cheat sheet. *Do not tear out the cheat sheet!* No outside material is allowed on this exam.
- You should write your answers legibly and in the space given for the question. Overly verbose answers will be penalized.
- Scratch paper is available on the back of the exam. *Do not tear out the scratch paper!* It messes with the auto-scanner.
- **You have 75 minutes (1.25 hours) for the exam.** Manage your time well. *Do not spend too much time on questions you do not understand and focus on answering as much as you can!*
- Make sure you *use the time well to think, be precise, and show as much work as possible*.

Name: ________________________________

NetID: ________________________________

Date: ________________________________

Problem 1 [10 points]

For each of the following statements, answer if it is `True` or `False`. Use the table at the bottom to mark you choices.

i. Dijkstra's algorithm works well on graphs with negative edge weights provided there is no negative length cycle.

ii. A problem can either be `NP-Complete` or `NP-Hard` but not both.

iii. If `P = NP` then every `NP-Complete` problem can be solved in polynomial time.

iv. `Graph 2-Coloring` can be decided in linear time.

v. The set of all programs is larger than the set of all languages.

vi. Every undecidable language is also unrecognizable.

vii. If language L is undecidable then either L or $\bar{L}$ is unrecognizable.

viii. If using an Oracle for problem X, one can obtain a decider for the Halt_{TM} then X is decidable.

ix. If a barber shaves everyone who doesn't shave themselves then the barber shaves themselves.

x. If a graph is 3-colorable then it has 3 independent sets.

Table 1.

Statement	Your choice
i.	False
ii.	False
iii.	True
iv.	True
v.	False
vi.	False
vii.	True
viii.	False
ix.	False
x.	True

Problem 2 [10 points]

Given a directed graph $G = (V, E)$ with non-negative edge lengths $l(e), e \in E$ and a node $s \in V$, describe an algorithm to find the length of a shortest cycle containing the node s.

Solution: Refer to Lab 15, Problem 2. ■

Problem 3 [10 points]

Formally prove or disprove the following statement. *There is no program that always stops and solves the halting problem.*

Solution: Answer: The statement is True. Refer to Lecture 22 for the proof of halting is undecidable. Alternatively, we can prove halt is undecidable via the fact that self-halt is undecidable. We prove by 2 steps:

- First, we show that self-halt is undecidable: Assuming self-halt is decidable, there exists a TM, named “SH", s.t. Accept(SH) = self-halt. and Diverge(SH) = ∅. Then if we create another TM, named “$\tilde{SH}$" and every transition to accept state in SH to be routed to a new hang state in $\tilde{SH}$. And similarly for every transition to reject state in SH to be accept in $\tilde{SH}$. Therefore Accept($\tilde{SH}$) = $\Sigma^*\backslash$self-halt and Reject($\tilde{SH}$) = ∅. This means $\tilde{SH}$ accepts $<\tilde{SH}>$ if and only if $\tilde{SH}$ does not halt on $<\tilde{SH}>$, which is a contradiction. So self-halt must be undecidable.

- Given self-halt is undecidable, we prove halt is undecidable. Now suppose halt is decidable, then there is a TM, H, that decides halt. Then we can create another machine, named SH, that decides H. Clearly, SH decides self-halt. Then given strings, we build the following transition/reduction function (assuming $<M> \in$ self-halt):

```
def SH(<M>):
    # reduction
    def M_(x):
        if x == <M>: return True
        return False
    # proxy
    if H(<M_, <M_>>):
        return True
    else:
        return False
```

As a result, we prove halt is undecidable (since self-halt is undecidable). Hence the statement is True (by the definition of undecidability).

Rubrics: 3 points for the proof of halt is undecidable, 3 points for the correct answer (True/False), 4 points for the reasoning that why undecidability leads to the statement: *There is no program that always stops and solves the halting problem.* ■

Problem 4 [20 points]

The `4-Set-Packing` problem is defined as follows.

- Inputs: A collection of m sets $S = \{S_1, S_2, \ldots, S_m\}$ such that $|S_i| = 4 \ \forall i \in \{1, \ldots, m\}$ and an integer k.
- Output: `True` if there exists a disjoint subcollection $L \subseteq S$ of size k. `False` otherwise.

  Note: Disjoint subcollection means no individual element belongs to two different sets in it.

The `3-Dimensional-Matching` problem is defined as follows.

- Inputs: Three disjoint sets X, Y and Z of n elements each, and a set of triplets $T \subseteq X \times Y \times Z$.
- Output: `True` if there exist disjoint triplets from T whose union is $X \cup Y \cup Z$. `False` otherwise.

Given `3-Dimensional Matching` is `NP-Complete`, show that `4-Set-Packing` is `NP-Complete`.

Solution: `4-Set-Packing` is in NP, as given any instance L, we can check in polynomial time: if the size of L is k; and that no element appears in more than one set in L.

To prove `4-Set-Packing` is NP-Hard, we reduce from `3-Dimensional-Matching`, which is known to be NP-Complete, in two steps:

Step 1: Construct the polynomial time reduction

Given an instance of `3-Dimensional-Matching`, with disjoint sets X, Y, and Z, and a set of triplets $T \subseteq X \times Y \times Z$, construct a `4-Set-Packing` instance by creating a set $S_i = \{x, y, z, w_i\}$ for each triplet $(x, y, z) \in T$, where w_i is a new element introduced to ensure each set has exactly four elements. The new elements w_i are unique to each set S_i.

Step 2: Prove the correctness of the reduction

Forward Direction: If there is a solution to the `3-Dimensional-Matching` instance, then we have a subset of triplets $M \subseteq T$ where $|M| = n$ and no two triplets in M share any element. We can map M to a collection L of $k = n$ sets in the `4-Set-Packing` instance, where each set in L corresponds to a triplet in M plus the unique element w_i. This collection L will be a valid solution to the `4-Set-Packing` problem.

Reverse Direction: Conversely, if there is a solution to the `4-Set-Packing` instance under our reduction, then we have a collection of sets L where no element is repeated across the sets. By removing the unique elements w_i from each set in L, we obtain a collection of triplets that form a valid solution to the original `3-Dimensional-Matching` instance.

Therefore, `4-Set-Packing` is NP-Hard, and since it is also in NP, it is NP-Complete. ■

Note: A reduction that doesn't allow different values of n will be completely incorrect, and just stating for NP it is verifiable in polynomial time is worth no point.

Problem 5 [14 points]

a. A quasi-satisfying assignment (`quasiSAT`) for a `3CNF` boolean formula ϕ is an assignment of truth values to the variables such that at most one clause in ϕ does not contain a `True` literal. Prove that it is `NP-Complete` to determine whether a given `3CNF` boolean formula has a quasi-satisfying assignment or not.

Solution: `quasiSAT` is trivially `NP`, since given a quasi-satisfying assignment for an instance, the quasi-satisfiability of the instance can be checked in polynomial time by applying the assignment and evaluating the formula.

To prove `NP-hardness` of `quasiSAT`, let ϕ be an arbitrary `3CNF` formula, and let ϕ' be a `3CNF` formula constructed by introducing 3 new variables x, y, z and combining ϕ with every 8 clauses that contains the literals of x, y, z as the following:

$$\phi' = \phi \wedge (x \vee y \vee z) \wedge (x \vee y \vee \overline{z}) \wedge (x \vee \overline{y} \vee z) \wedge \ldots \wedge (\overline{x} \vee \overline{y} \vee z) \wedge (\overline{x} \vee \overline{y} \vee \overline{z})$$

Then, ϕ is satisfiable if and only if ϕ' is quasi-satisfiable.

$\rightarrow$ Suppose ϕ is satisfiable. Then, there exists a satisfying assignment A for ϕ. Let A' be an assignment constructed by combining A with the assignment $x = \texttt{T}$, $y = \texttt{T}$, $z = \texttt{T}$. If we apply the assignment A' on ϕ', all clauses except for $(\overline{x} \vee \overline{y} \vee \overline{z})$ would contain a `True` literal. Therefore, ϕ' is quasi-satisfiable if ϕ is satisfiable.

$\leftarrow$ Suppose ϕ' is quasi-satisfiable. Note that regardless of the assignment on the variables x, y, z, exactly one of the last 8 clauses would not contain a `True` literal. This implies that there exists an assignment A^* for ϕ' such that all clauses before the last 8 clauses contain a `True` literal. We can construct a satisfying assignment A for ϕ by getting rid of the assignments for x, y, z from A^*. Therefore, ϕ is satisfiable if ϕ' is quasi-satisfiable.

Since `quasiSAT` is `NP`, and also there exists a polynomial time reduction from `3SAT` to `quasiSAT`, we conclude that `quasiSAT` is `NP-Complete`. ■

b. Show that the `Hamiltonian Cycle` problem for **undirected** graphs is `NP-Complete`. Note: You may use that `Hamiltonian Cycle` problem for directed graphs is `NP-Complete`.

Solution: `Hamiltonian Cycle` for undirected graph is trivially `NP`, since given a Hamiltonian cycle in an instance, we can simply iterate over the cycle and check if it indeed is a Hamiltonian cycle.

To prove `NP-hardness`, let $G = (V, E)$ be an arbitrary directed graph, and let $G' = (V', E')$ be an undirected graph defined as the following:

$$V' = \{v_{in}, v_{mid}, v_{out} \mid v \in V\}$$
$$E' = \{(u_{out}, v_{in}) \mid (u, v) \in E\} \cup \{(v_{in}, v_{mid}), (v_{mid}, v_{out}) \mid v \in V\}$$

Then, V has a directed Hamiltonian cycle if and only if V' has an undirected Hamiltonian cycle.

$\rightarrow$ Suppose there exists a directed Hamiltonian cycle $C = (c_1, c_2, ..., c_n)$ in G. By construction, $C' = (c_{1in}, c_{1mid}, c_{1out}, c_{2in}, ..., c_{nmid}, c_{nout})$ is an undirected Hamiltonian cycle in G'.

$\leftarrow$ Suppose there exists an undirected Hamiltonian cycle C' in G'. By construction, C' can be written in the following form:

$$C' = (c_{1in}, c_{1mid}, c_{1out}, c_{2in}, ..., c_{nmid}, c_{nout})$$

Since u_{out} is connected to v_{in} if and only if $(u, v) \in E$, $C = (c_1, c_2, ..., c_n)$ forms a directed Hamiltonian cycle of G.

We showed that `Hamiltonian Cycle` for undirected graphs is both `NP` and `NP-hard`. Therefore we conclude that `Hamiltonian Cycle` for undirected graphs is `NP-complete`. ■

Problem 6 [10 points]

Identify the errors in the following proofs.

a. Define the following problems.

- `DFA-Accepts`
  Inputs: A DFA D and a string w. Output: `True` if $w \in L(D)$. `False` otherwise.
- `NFA-Accepts`
  Inputs: A NFA N and a string w. Output: `True` if $w \in L(N)$. `False` otherwise.

Note the following.

- `DFA-Accepts` is in `P` as there is a single execution path for w on D.
- Its highly unlikely that `NFA-Accepts` is in `P`. Intuitively, there are exponentially many ways to simulate w on N that makes `NFA-Accepts` `NP-Hard`.

Construct a solver for `NFA-Accepts` as follows.

Step 1. Convert the given NFA into an equivalent DFA.

Step 2. Now use the poly-time solver for `DFA-Accepts` to solve `NFA-Accepts`.

This implies `NFA-Accepts` which is `NP-Hard` has a poly-time solver implying `P = NP`. [Did we just solve the millennium problem!?]

Solution: While a polynomial time reduction from a known NP-Hard problem to a problem in P would imply that P=NP , the incremental subset construction algorithm for converting NFAs to DFAs is exponential in the number of states of the NFA. ■

b. Refer to the cheat sheet for the definition of the `Independent Set` decision problem. Consider the following decider for this problem.

`DecideIndependantSet`$(G = (V, E),\ k)$:

For each $S \subseteq V$ such that $|S| = k$:

  bool ← `True`

  For every pair of two vertices (u, v) from the set S:

    If there is an edge between u and v:

      bool ← `False`

If bool == True:

  return True

Else:

  return False

The runtime of the above algorithm is $T(n) = O\left((n^k)k^2\right)$. This implies `Independent Set` which is `NP-Hard` has a poly-time solver implying `P = NP`. [Did we just solve the millennium problem again!?]

Solution: While a polynomial time solution to any NP-Hard problem would imply that P=NP, the provided algorithm is not polynomial time in terms of all of the inputs. A polynomial of n and k takes the form $(n + k)^{\alpha}$, which would never contain an n^k term.

■

Problem 7 [6 points]

Prove or disprove that the `Halting` problem is `NP-Hard`.

Solution: Refer to Lecture 23 – Pre-Lecture Brain Teaser. The solution is in the scribbles.

Reduce SAT to Halting:
For an arbitrary SAT solver, modify it as follows:
If the input instance is satisfiable, return accept, otherwise let turning machine M not halt on input w.
Then the turning machine halts if and only if the SAT instance is satisfiable and the reduction takes polynomial time.
So Halting is NP-Hard.

Note that Halting is not NP-complete as verifying the input in polynomial time is impossible.

You cannot prove NP-hard from undecidable. Some problems are undecidable but not NP-hard (out of the scope of this course). ■

Problem 8 [20 points]

For definitions of $\texttt{A}_{\texttt{TM}}$, $\texttt{Halt}_{\texttt{TM}}$, $\texttt{HaltB}_{\texttt{TM}}$ refer to the cheat sheet.

a. Using undecidability of $\texttt{A}_{\texttt{TM}}$, show that $\texttt{HaltB}_{\texttt{TM}}$ is undecidable.

Solution: (Refer to Lecture 24. The solution is in the scribbles.) Remember that

$$\texttt{HaltB}_{\texttt{TM}} = \{\langle M \rangle | M \text{ is a TM and } M \text{ halts on blank input.}\}$$

Suppose there is an algorithm DecideHaltOnBlank that correctly decides the language HaltOnBlank. Then, we can solve the AcceptOnInput problem as follows:

```
DecideAcceptOnInput(⟨M, w⟩):
  Encode the following Turing machine M′:
      M′(x):
        if x == ε:
            if M(w):
                return True
            else:
                Loop forever
        else:
            Loop forever
if DecideHaltOnBlank(⟨M′⟩)
  return True
else
  return False
```

Alternatively, M' can also be constructed as

```
M′(x):
  if M(w)
      return True
  Loop forever
```

We prove this reduction correct as follows:

$\Longrightarrow$ Suppose M accepts input w.
Then M' accepts the input string ϵ.
So DecideHaltOnBlank accepts the encoding $\langle M' \rangle$.
So DecideAcceptOnInput correctly accepts the encoding $\langle M, w \rangle$.

$\Longleftarrow$ Suppose M does not halt on input w.
Then M' does not halt on *any* input string x.
So DecideHaltOnBlank rejects the encoding $\langle M' \rangle$.
So DecideAcceptOnInput correctly rejects the encoding $\langle M, w \rangle$.

In both cases, DecideAcceptOnInput is correct. But that's impossible because Halt is undecidable. We conclude that the algorithm DecideHaltOnBlank does not exist. ■

b. Using undecidability of $\texttt{Halt}_{\texttt{TM}}$, show that the following language is undecidable.

$$\texttt{Reg}_{\texttt{TM}} = \{\langle M\rangle | M \text{ is a TM and } L(M) \text{ is regular.}\}$$

Solution: (Refer to Lab 22.)

Suppose there is an algorithm DecideAcceptRegular that correctly decides the language AcceptRegular. Then, we can solve the DecideHaltOnInput problem as follows:

```
DecideHaltOnInput(⟨M, w⟩):
  Encode the following Turing machine M':
      M'(x):
        if x is of the form 0^n 1^n
            return True
        else
            run M on input w
            return True
  if DecideAcceptRegular(⟨M'⟩)
      return True
  else
      return False
```

We prove this reduction correct as follows:

$\Longrightarrow$ Suppose M halts on input w.
Then M' accepts *every* the input string, i.e., Σ^*, which is regular.
So DecideAcceptRegular accepts the encoding $\langle M'\rangle$.
So DecideHaltOnInput correctly accepts the encoding $\langle M, w\rangle$.

$\Longleftarrow$ Suppose M does not halt on input w.
Then M' does not halt on *any* input string except 0^n1^n, which is not regular.
So DecideAcceptRegular rejects the encoding $\langle M'\rangle$.
So DecideHaltOnInput correctly rejects the encoding $\langle M, w\rangle$.

In both cases, DecideHaltOnInput is correct. But that's impossible because HaltOnInput is undecidable. We conclude that the algorithm DecideAcceptRegular does not exist. Therefore REG_{TM} must be undecidable, ■

This page is for additional scratch work!

ECE 374 B Algorithms: Cheatsheet

1 Recursion

Simple recursion

Definitions

- **Reduction:** solve one problem using the solution to another.
- **Recursion:** a special case of reduction - reduce problem to a *smaller* instance of *itself* (self-reduction).
  - Problem instance of size n is reduced to *one or more* instances of size $n-1$ or less.
  - For termination, problem instances of small size are solved by some other method as *base cases*

Tower of Hanoi

Arguably the most famous example of recursion. The goal is to move n disks one at a time from the first peg to the last peg.

Pseudocode: Tower of Hanoi

```
Hanoi (n, src, dest, tmp):
    if (n > 0) then
        Hanoi (n − 1, src, tmp, dest)
        Move disk n from src to dest
        Hanoi (n − 1, tmp, dest, src)
```

Recurrences

Suppose you have a recurrence of the form $T(n) = rT(n/c) + f(n)$.

The *master theorem* gives a good asymptotic estimate of the recurrence. If the work at each level is:

Decreasing:	$rf(n/c) = \kappa f(n)$ where $\kappa < 1$	$T(n) = O(f(n))$
Equal:	$rf(n/c) = f(n)$	$T(n) = O(f(n) \cdot \log_c n)$
Increasing:	$rf(n/c) = Kf(n)$ where $K > 1$	$T(n) = O(n^{\log_c r})$

Some useful identities:

- Sum of integers: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
- Geometric series closed-form formula: $\sum_{k=0}^{n} ar^k = \frac{1-r^{n+1}}{1-r}$
- Logarithmic identities: $\log(ab) = \log a + \log b$, $\log(a/b) = \log a - \log b$, $a^{\log_c b} = b^{\log_c a}$ $(a, b, c > 1)$.

Backtracking

Backtracking is the algorithm paradigm involving guessing the solution to a single step in some multi-step process and recursing backwards if it doesn't lead to a solution. For instance, consider the longest increasing subsequence (LIS) problem. You can either check all possible subsequences:

Pseudocode: LIS - Naive enumeration

```
algLISNaive(A[1..n]):
    maxmax = 0
    for each subsequence B of A do
        if B is increasing and |B| > max then
            max = |B|
    return max
```

On the other hand, we don't need to generate every subsequence; we only need to generate the subsequences that are increasing:

Pseudocode: LIS - Backtracking

```
LIS_smaller(A[1..n], x):
    if n = 0 then return 0
    max = LIS_smaller(A[1..n − 1], x)
    if A[n] < x then
        max = max {max, 1 + LIS_smaller(A[1..(n − 1)], A[n])}
    return max
```

Divide and conquer

Divide and conquer is an algorithm paradigm involving the decomposition of a problem into the same subproblem, solving them separately and combining their results to get a solution for the original problem.

Sorting algorithms

Algorithm	Runtime	Space
Mergesort	$O(n \log n)$	$O(n \log n)$ $O(n)$ (if optimized)
Quicksort	$O(n^2)$ $O(n \log n)$ if using MoM	$O(n)$

Karatsuba's algorithm

We can divide and conquer multiplication like so:

$$bc = 10^n b_L c_L + 10^{n/2}(b_L c_R + b_R c_L) + b_R c_R.$$

We can rewrite the equation as:

$$\begin{aligned} bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R) &= (b_L c_L)x^2 \\ &+ ((b_L + b_R)(c_L + c_R) - b_L c_L - b_R c_R)\, x \\ &+ b_R c_R, \end{aligned}$$

Its running time is $O(n^{\log_2 3}) = O(n^{1.585})$.

Linear time selection

The *median of medians* (MoM) algorithms give a element that is larger than $\frac{3}{10}$'s and smaller than $\frac{7}{10}$'s of the array elements. This is used in the linear time selection algorithm to find element of rank k.

Pseudocode: Quickselect with median of medians

```
Median-of-medians (A, i):
    sublists = [A[j:j+5] for j ← 0, 5, . . . , len(A)]
    medians = [sorted (sublist)[len (sublist)/2]
              for sublist ∈ sublists]

    // Base case
    if len (A) ≤ 5 return sorted (a)[i]

    // Find median of medians
    if len (medians) ≤ 5
        pivot = sorted (medians)[len (medians)/2]
    else
        pivot = Median-of-medians (medians, len/2)

    // Partitioning step
    low = [j for j ∈ A if j < pivot]
    high = [j for j ∈ A if j > pivot]

    k = len (low)
    if i < k
        return Median-of-medians (low, i)
    else if i > k
        return Median-of-medians (low, i-k-1)
    else
    return pivot
```

Dynamic programming

Dynamic programming (DP) is the algorithm paradigm involving the computation of a recursive backtracking algorithm iteratively to avoid the recomputation of any particular subproblem.

Longest increasing subsequence

The longest increasing subsequence problem asks for the length of a longest increasing subsequence in a unordered sequence, where the sequence is assumed to be given as an array. The recurrence can be written as:

$$LIS(i,j) = \begin{cases} 0 & \text{if } i = 0 \\ LIS(i-1,j) & \text{if } A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1,j) \\ 1 + LIS(i-1,i) \end{cases} & \text{else} \end{cases}$$

Pseudocode: LIS - DP

```
LIS-Iterative(A[1..n]):
    A[n+1] = ∞
    for j ← 0 to n
        if A[i] ≤ A[j] then LIS[0][j] = 1

    for i ← 1 to n − 1 do
        for j ← i to n − 1 do
            if A[i] ≥ A[j]
                LIS[i, j] = LIS[i − 1, j]
            else
                LIS[i, j] = max {LIS[i − 1, j],
                                 1 + LIS[i − 1, i]}
    return LIS[n, n + 1]
```

Edit distance

The edit distance problem asks how many edits we need to make to a sequence for it to become another one. The recurrence is given as:

$$\mathrm{Opt}(i,j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{Opt}(i-1,j-1), \\ \delta + \mathrm{Opt}(i-1,j), \\ \delta + \mathrm{Opt}(i,j-1) \end{cases}$$

Base cases: $\mathrm{Opt}(i,0) = \delta \cdot i$ and $\mathrm{Opt}(0,j) = \delta \cdot j$

Pseudocode: Edit distance - DP

```
EDIST(A[1..m], B[1..n])
    for i ← 1 to m do M[i, 0] = iδ
    for j ← 1 to n do M[0, j] = jδ

    for i = 1 to m do
        for j = 1 to n do
                          ⎧ COST[A[i]][B[j]]
                          ⎪         +M[i − 1][j − 1],
            M[i][j] = min ⎨ δ + M[i − 1][j],
                          ⎩ δ + M[i][j − 1]
```

2 Graph algorithms

Graph basics

A graph is defined by a tuple $G = (V, E)$ and we typically define $n = |V|$ and $m = |E|$. We define (u, v) as the edge from u to v. Graphs can be represented as **adjacency lists**, or **adjacency matrices** though the former is more commonly used.

- *path*: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $v_i v_{i+1} \in E$ for $1 \leq i \leq k-1$. The length of the path is $k-1$ (the number of edges in the path). *Note:* a single vertex u is a path of length 0.
- *cycle*: sequence of *distinct* vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k-1$ and $(v_k, v_1) \in E$. A single vertex is not a cycle according to this definition.
  Caveat: Sometimes people use the term cycle to also allow vertices to be repeated; we will use the term *tour*.
- A vertex u is *connected* to v if there is a path from u to v.
- The *connected component* of u, $\mathrm{con}(u)$, is the set of all vertices connected to u.
- A vertex u can *reach* v if there is a path from u to v. Alternatively v can be reached from u. Let **rch**(u) be the set of all vertices reachable from u.

Directed acyclic graphs

Directed acyclic graphs (dags) have an intrinsic ordering of the vertices that enables dynamic programming algorithms to be used on them.
A *topological ordering* of a dag $G = (V, E)$ is an ordering $\prec$ on V such that if $(u, v) \in E$ then $u \prec v$.

Pseudocode: Kahn's algorithm

Kahn($G(V,E)$,u):
toposort←empty list
for $v \in V$:
  in$(v) \leftarrow |\{u \mid u \to v \in E\}|$
while $v \in V$ that has in$(v) = 0$:
  Add v to end of toposort
  Remove v from V
  for v in $u \to v \in E$:
    in$(v) \leftarrow$ in$(v) - 1$
return toposort

Running time: $O(n + m)$

- A dag may have multiple topological sorts.
- A topological sort can be computed by DFS, in particular by listing the vertices in decreasing post-visit order.

DFS and BFS

Pseudocode: Explore (DFS/BFS)

Explore(G,u):
  for $i \leftarrow 1$ to n:
    Visited$[i] \leftarrow$ False
  Add u to ToExplore and to S
  Visited$[u] \leftarrow$ True
  Make tree T with root as u
  while B is non-empty **do**
    Remove node x from B
    for each edge (x, y) in $Adj(x)$ **do**
      if Visited$[y]$ = False
        Visited$[y] \leftarrow$ True
        Add y to B, S, T (with x as parent)

Note:

- If B is a queue, *Explore* becomes BFS.
- If B is a stack, *Explore* becomes DFS.

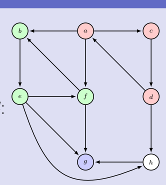
Pre/post numbering

Pre and post numbering aids in analyzing the graph structure. By looking at the numbering we can tell if a edge (u, v) is a:

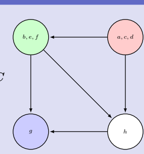
- *Forward edge*: pre(u) < pre(v) < post(v) < post(u)
- *Backward edge*: pre(v) < pre(u) < post(u) < post(v)
- *Cross edge*: pre(u) < post(u) < pre(v) < post(v)

Strongly connected components

- Given G, u is *strongly connected* to v if $v \in$ rch(u) *and* $u \in$ rch(v).
- A *maximal* group of G: vertices that are all strongly connected to one nother is called a strong component.



G^{SCC}



Pseudocode: Metagraph - linear time

Metagraph($G(V,E)$):
  Compute rev(G) by brute force
  ordering ← reverse postordering of V in rev(G)
    by **DFS**(rev(G), s) for any vertex s
  Mark all nodes as unvisited
  for each u in ordering **do**
    if u is not visited and $u \in V$ **then**
      $S_u \leftarrow$ nodes reachable by u by **DFS**(G, u)
      Output S_u as a strong connected component
      $G(V, E) \leftarrow G - S_u$

Shortest paths

Dijkstra's algorithm:
Find minimum distance from vertex s to **all** other vertices in graphs *without* negative weight edges.

Pseudocode: Dijkstra

for $v \in V$ **do**
  $d(v) \leftarrow \infty$
$X \leftarrow \varnothing$
$d(s, s) \leftarrow 0$
for $i \leftarrow 1$ to n **do**
  $v \leftarrow \arg\min_{u \in V - X} d(u)$
  $X = X \cup \{v\}$
  for u in Adj(v) **do**
    $d(u) \leftarrow \min\{(d(u),\ d(v) + \ell(v, u))\}$
return d

Running time: $O(m+n\log n)$ (if using a Fibonacci heap as the priority queue)

Bellman-Ford algorithm:
Find minimum distance from vertex s to **all** other vertices in graphs *without* negative cycles. It is a DP algorithm with the following recurrence:

$$d(v,k) = \begin{cases} 0 & \text{if } v = s \text{ and } k = 0 \\ \infty & \text{if } v \neq s \text{ and } k = 0 \\ \min \begin{cases} \min_{uv \in E}\{d(u, k-1) + \ell(u, v)\} \\ d(v, k-1) \end{cases} & \text{else} \end{cases}$$

Base cases: $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$.

Pseudocode: Bellman-Ford

for each $v \in V$ **do**
  $d(v) \leftarrow \infty$
$d(s) \leftarrow 0$

for $k \leftarrow 1$ to $n - 1$ **do**
  for each $v \in V$ **do**
    for each edge $(u, v) \in$ in(v) **do**
      $d(v) \leftarrow \min\{d(v), d(u) + \ell(u, v)\}$

return d

Running time: $O(nm)$

Floyd-Warshall algorithm:
Find minimum distance from *every* vertex to *every* vertex in a graph *without* negative cycles. It is a DP algorithm with the following recurrence:

$$d(i,j,k) = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } (i,j) \notin E \text{ and } k = 0 \\ \min \begin{cases} d(i,j,k-1) \\ d(i,k,k-1) + d(k,j,k-1) \end{cases} & \text{else} \end{cases}$$

Then $d(i, j, n-1)$ will give the shortest-path distance *from* i *to* j.

Pseudocode: Floyd-Warshall

Metagraph($G(V,E)$):
  for $i \in V$ **do**
    for $j \in V$ **do**
      $d(i, j, 0) \leftarrow \ell(i, j)$
      (* $\ell(i,j) \leftarrow \infty$ if $(i,j) \notin E$, 0 if $i = j$ *)

  for $k \leftarrow 0$ to $n - 1$ **do**
    for $i \in V$ **do**
      for $j \in V$ **do**
        $d(i, j, k) \leftarrow \min \begin{cases} d(i,j,k-1), \\ d(i,k,k-1) + d(k,j,k-1) \end{cases}$
  for $v \in V$ **do**
    if $d(i, i, n-1) < 0$ **then**
      return "∃ negative cycle in G"

  return $d(\cdot, \cdot, n-1)$

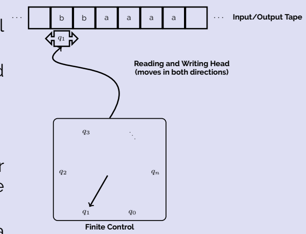
Running time: $\Theta(n^3)$

ECE 374 B Reductions, P/NP, and Decidability: Cheatsheet

Turing Machines

Turing machine is the simplest model of computation.

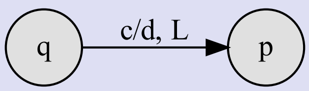
- Input written on (infinite) one sided tape.
- Special blank characters.
- Finite state control (similar to DFA).
- Ever step: Read character under head, write character out, move the head right or left (or stay).
- Every TM **M** can be encoded as a string $\langle M \rangle$



Transition Function: $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow, \square\}$

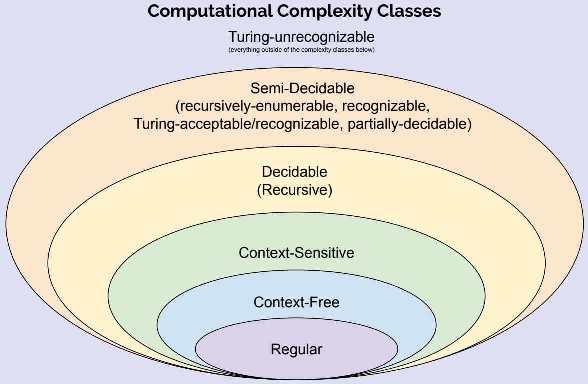
$\delta(q, c) = (p, d, \leftarrow)$

- q: current state.
- c: character under tape head.
- p: new state.
- d: character to write under tape head
- $\leftarrow$: Move tape head left.

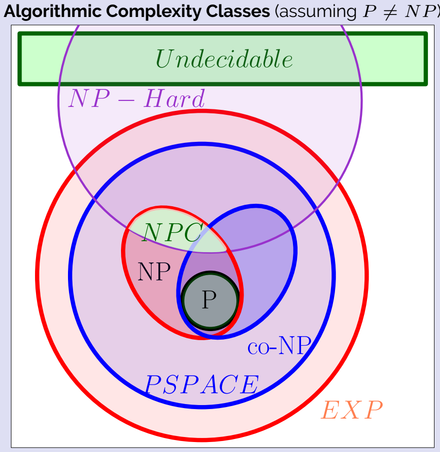


Complexity Classes

Computational Complexity Classes



Algorithmic Complexity Classes (assuming $P \neq NP$)



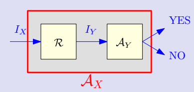
Reductions

A general methodology to prove impossibility results.

- Start with some *known* hard problem X
- *Reduce* X to your favorite problem Y

If Y can be solved then so can $X \implies Y$. But we know X is hard so Y has to be hard too. On the other hand if we know Y is easy, then X has to be easy too.

The Karp reduction, $X \leq_P Y$ suggests that there is a polynomial time reduction from X to Y.



Assuming

- $R(n)$: running time of $\mathcal{R}$
- $Q(n)$: running time of $\mathcal{A}_Y$

Running time of $\mathcal{A}_X$ is $O(Q(R(n)))$

Sample NP-complete problems

CircuitSat: Given a boolean circuit, are there any input values that make the circuit output True?

3Sat: Given a boolean formula in conjunctive normal form, with exactly three distinct literals per clause, does the formula have a satisfying assignment?

IndependentSet: Given an undirected graph G and integer k, what is there a subset of vertices $\geq k$ in G that have no edges among them?

Clique: Given an undirected graph G and integer k, is there a complete complete subgraph of G with more than k vertices?

KPartition: Given a set X of kn positive integers and an integer k, can X be partitioned into n, k-element subsets, all with the same sum?

3Color: Given an undirected graph G, can its vertices be colored with three colors, so that every edge touches vertices with two different colors?

HamiltonianPath: Given graph G (either directed or undirected), is there a path in G that visits every vertex exactly once?

HamiltonianCycle: Given a graph G (either directed or undirected), is there a cycle in G that visits every vertex exactly once?

LongestPath: Given a graph G (either directed or undirected, possibly with weighted edges) and an integer k, does G have a path $\geq k$ length?

- Remember a **path** is a sequence of distinct vertices $[v_1, v_2, \ldots v_k]$ such that an edge exists between any two vertices in the sequence. A **cycle** is the same with the addition of a edge $(v_k, v_1) \in E$. A **walk** is a path except the vertices can be repeated.
- A formula is in conjunction normal form if variables are or'ed together inside a clause and then clauses are and'ed together: $((x_1 \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee x_4 \vee x_5))$. Disjunctive normal form is the opposite $((x_1 \wedge x_2 \wedge x_3) \vee (\overline{x_2} \wedge x_4 \wedge x_5))$.

Sample undecidable problems

AcceptOnInput: $A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts on } w\}$

HaltsOnInput: $Halt_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and halts on input } w\}$

HaltOnBlank: $HaltB_{TM} = \{\langle M\rangle \mid M \text{ is a TM \& } M \text{ halts on blank input}\}$

Emptiness: $E_{TM} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \varnothing\}$

Equality: $EQ_{TM} = \left\{\langle M_A, M_B\rangle \;\middle|\; \begin{array}{l} M_A \text{ and } M_B \text{ are TM's} \\ \text{and } L(M_A) = L(M_B) \end{array}\right\}$