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 $\mathtt{Halt}_{\mathtt{TM}}$ 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.	
ii.	
iii.	
iv.	
v.	
vi.	
vii.	
viii.	
ix.	
x.	

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.

Problem 3 [10 points]

Formally prove or disprove the following 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`.

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.

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`.

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!?]

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!?]

Problem 7 [6 points]

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

Problem 8 [20 points]

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

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

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

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

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)
```

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:

$$bc = b(x)c(x) = (b_L x + b_R)(c_L x + c_R) = (b_L c_L)x^2 + ((b_L + b_R)(c_L + c_R) - b_L c_L - b_R c_R)\,x + b_R c_R,$$

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

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
```

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:

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

Pseudocode: LIS - DP

LIS-Iterative$(A[1..n])$:
  $A[n+1] = \infty$
  for $j \leftarrow 0$ to n
    if A[i] $\leq$ A[j] **then** $LIS[0][j] = 1$

  for $i \leftarrow 1$ to $n-1$ **do**
    for $j \leftarrow i$ to $n-1$ **do**
      if $A[i] \geq A[j]$
        $LIS[i,j] = LIS[i-1,j]$
      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 $i \leftarrow 1$ to m **do** $M[i,0] = i\delta$
  for $j \leftarrow 1$ to n **do** $M[0,j] = j\delta$

  for $i = 1$ to m **do**
    for $j = 1$ to n **do**

$$M[i][j] = \min \begin{cases} COST\,[A[i]]\,[B[j]] \\ \qquad +M[i-1][j-1], \\ \delta + M[i-1][j], \\ \delta + M[i][j-1] \end{cases}$$

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 $v_1, v_2, \dots, v_k$ 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 $v_1, v_2, \dots, v_k$ 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.

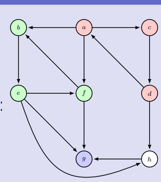
Pre/post numbering

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:

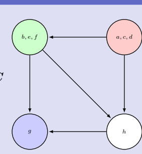
- *Forward edge*: pre$(u) <$ pre$(v) <$ post$(v) <$ post(u)
- *Backward edge*: pre$(v) <$ pre$(u) <$ post$(u) <$ post(v)
- *Cross edge*: pre$(u) <$ post$(u) <$ pre$(v) <$ post(v)

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

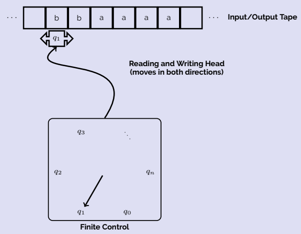
Running time: $\Theta(n^3)$

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

Turing Machines

Turing machine is the simplest model of computation.

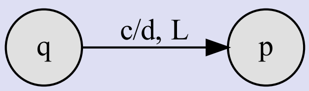
- Input written on (infinite) one sided tape.
- Special blank characters.
- Finite state control (similar to DFA).
- Ever step: Read character under head, write character out, move the head right or left (or stay).
- Every TM **M** can be encoded as a string $\langle M \rangle$



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

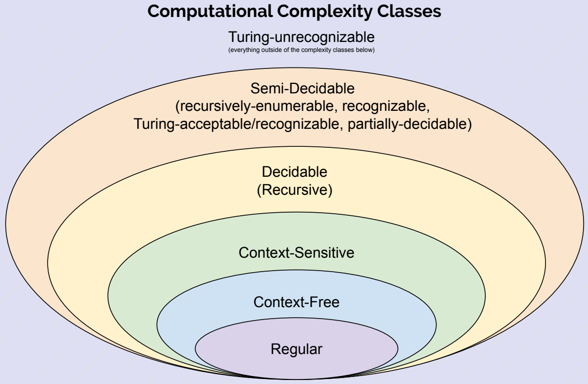
$\delta(q, c) = (p, d, \leftarrow)$

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

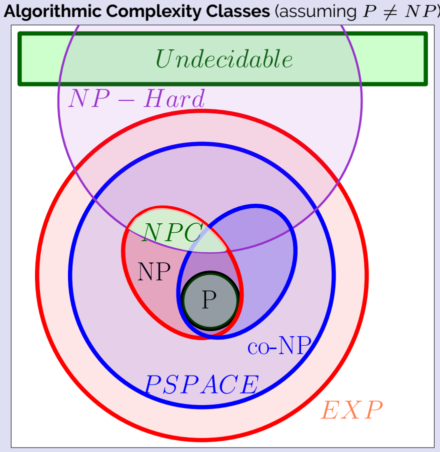


Complexity Classes

Computational Complexity Classes



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



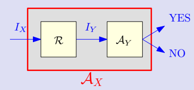
Reductions

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

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

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

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

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