**CSC373 Assignment 3**

**Terminology and Notation Table:**

|  |  |
| --- | --- |
| **Notation** | |
|  | such that holds |
|  | Shorthand for the sequence  is used for setting a “pattern precedent” (e.g. it’s unclear that represents the series , but it is completely clear when we write something like – we’re setting a “precedent” for the pattern)  If is not included, we assume this to be the sequence |
|  | Encoding of the variable sequence (assumed to be binary encoding) |
|  | The set of all encodings of where predicate holds |
|  | Problem can be reduced in polynomial time to problem  is poly-time reducible to |
|  | Some path between nodes and |
|  | Directed edge leading from node to node |
|  | Undirected edge connecting nodes |
|  | Could be or |
| **Symbols / Functions / Algorithms / Predicates** | |
|  | Language of some problem |
|  | Some certificate for problem |
|  | Verifier for some problem that checks if certificate is an appropriate solution to the input variables |
|  | Algorithm / function that solves problem . In other words,  Alternative phrasing: algorithm that decides |
| **Problem Names** | |
|  | Binary Feasibility Problem (feasibility problem) |
|  | Node Disjoint Path decision problem |
|  | Edge Disjoint Path decision problem |
|  | Exam Scheduling decision problem |
|  | Mine Sweeper Satisfiability problem |

1. Integer Programming
   1. Dual for Integer Programming
      * Definition:
        + Minimize:
        + Subject to:
          - (horizontal -dimensional integer-vector)
      * Integer Programming Weak Duality:
        + Definition:
        + Proof:
          - Let:

be a feasible solution to the **primal** integer program

be a feasible solution to the **dual**integer program

* + - * + Since is feasible, according to the primal we have
        + Now we multiply both sides by to get:
        + From the definition of the dual,
        + By transitivity:
        + Thus we can conclude .
  1. Integer Programming Strong Duality Counter Example
     + Definition of Strong Duality:
       - (Primal Feasible)(Primal Bounded)(Optimal Primal Value = Optimal Dual Value)
       - To disprove an implication we need to find a case where , so for this definition, we need to find:

(Primal Feasible)(Primal Bounded)(Optimal Primal Value Optimal Dual Value)

* + - Consider a primal-dual pair of integer programs with the following input values:
    - Primal:
      * Integer Program
        + Maximize
        + Subject to:
      * Clearly, we can maximize with , as it is the highest integer that does not violate the constraint that .
      * Thus, the maximized value of the objective function here is 4
    - Dual of Primal:
      * Integer Program:
        + Minimize
        + Subject to:
      * Clearly, the lowest we can go with is , which is the lowest integer that does not violate the constraint that .
      * Thus, the minimized value of the objective function here is .
    - We can now say that strong duality does not hold under integer programming, as we have shown a case where:
      * The primal integer program was feasible and bounded
      * The primal’s optimal value was **not** equal to the dual’s optimal value.
  1. -Feasibility Problem

|  |  |
| --- | --- |
| Inputs |  |
| Outputs | Truth value of statement: |

* + - Language
    - Proof of NP-Completeness
      * Proof that
        + Proof that is polynomial sized

is the in

We know that is a binary vector of length .

Thus, is of length where is a fixed constant

* + - * + Proof that runs in polynomial time

# Some notation rules

A.row(i) = i-th row of matrix A (as a vector)

A x B = matrix multiplication of matrices A, B

A.num\_rows = number of rows of matrix A

v.dot(u) = dot product of vectors v and u

v[i] = i-th element of vector v

def V\_BF(A, b, gamma\_BF):

# Multiply A and certificate to get result

result = A x gamma\_BF

# Check whether the result satisfies the constraint

for i in range(len(result)):

if gamma\_BF.dot(A.row(i)) > b[i]:

return False

# All constraints must be true at this point

return True

All the code executed by runs in polynomial time

: Matrix multiplication runs in polynomial time (DPV)

is of length , the loop that iterates over it must take time, which is clearly polynomial

Thus the whole of must run in polynomial time as well

* + - * Lemma: the following Boolean/Integer conversion system preserves truth values
        + Let:

respectively

be the integer value of Boolean

be the Boolean value of Integer

* + - * + Conversion Table

|  |  |
| --- | --- |
| **Conversion** | **Truth/Integer Table** |
| **Negation/**  **Subtraction** | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  |  |  |  |  | |  |  |  |  |  | |  | 0 |  |  |  | |
| **Or/**  **Addition** | |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |  |  |  |  |  |  |  | |

* + - * + We note that the above conversions maintain truth values: i.e. the Boolean conversion of the integer conversion equals the Boolean value, **provided**:

The input to negations are within the set

The inputs of or statements are within the range , where 0 indicates and indicates

* + - * Proof that – pick . Proof that :
        + Let:

be the variables in the 3SAT circuit

be the clauses in the 3SAT circuit

* + - * + An algorithm to solve that uses some :

Convert inputs from in polynomial time:

Convert clauses to integer equations:

Represent each clause with an integer equation

using the conversion system described above

Create the following constraint for each :

The above is equivalent to each being true, which we need to have if is to be true)

Once we have converted our Boolean variables and clauses to integer variables and equations, it is trivial to reformat them into standard form (needed for the integer program’s inputs):

is the coefficient matrix of size

-th row represents the -th equation’s coefficients

-th column represents the coefficient for

is the constraint vector of size

-th element represents the constraint of the -th equation

Since there are and clauses of size 3 (each clause has 3 variables) to convert into linear equations , conversion into standard form should take time , which is clearly polynomial with respect to the input size

We can be sure that the Boolean/integer conversions are valid because all integer versions of variables will be between 0 and 1, which means:

All negation conversions are valid (since we only negate single variables, the lowest value of a negation is 0)

Since the lowest value of a negation is 0, we don’t need to worry about OR statements outputting the wrong result (there will be no negative inputs)

Solve the converted inputs of to with some

If returns true it means:

There exists some such that .

By the constraints we set for each , the above means that:

We note that a way to get all the to be was to give them the binary vector as input

We can use our Boolean/integer conversion system to translate back into (since each , each )

To maintain the truth values of with respect to the integer values of , the integer inputs to each must also be passed through the Boolean/integer conversion system as well

Thus for every to return true, we can set the input variables to the (element-wise) Boolean equivalent of :

Thus, there exists a satisfying assignment of variables for the 3SAT circuit

If it returns false:

There is no that satisfies the inequality in the converted instance

Thus there is no assignment of variables that satisfies the 3CNF circuit

Therefore, we can directly use the output of some as the output of our algorithm (this is constant time and is therefore a polynomial conversion)

* + - * + Therefore we can use an algorithm that solves to solve , by converting between their respective inputs and outputs in polynomial time.
        + Thus, we can say that . In other words, we have found a language in that reduces in polynomial time to .
      * We have verified that and , which means that we can conclude that

1. Disjoint Paths
   1. Language Formulation
   2. Proof that : we use some to solve by converting the I/O in polynomial time
      * Input conversion from :
        + Let:
          - be the variables in the circuit, and be an arbitrary variable
          - be the clauses in the circuit, and be an arbitrary clause
        + Create the following graph :
          - Nodes :

For each variable , add the following nodes to :

(source node for )

) (terminal node for )

( “intermediate True” node for )

( “intermediate False” node for )

For each clause , add the following nodes to :

(source node for )

) (terminal node for )

* + - * + Edges:

For each variable , make two by adding the following edges

Add ’s “true path”, composed of the following edges:

Add ’s “false path”, composed of the following edges:

For each clause , where (WLOG for literals ) , add the following edges:

Let be one of the literals .

If is negated, add edges:

Otherwise, add edges:

* + - * + Time to create this graph is polynomial:

Each clause and each variable in the original 3SAT problem requires the introduction of:

node

Some number of edges that is polynomial with respect to

Since the number of nodes and edges is polynomial with respect to the input sizes of 3SAT, we can say that the total number of nodes and edges in is also polynomial with respect to the input sizes of 3SAT

* + - * Let the source and terminal nodes be as follows:
        + Making these lists would require at most time (we only need to loop over clauses and variables)
        + Thus, this process is also polynomial with respect to 3SAT’s input sizes
    - We execute to solve for the converted inputs, thereby solving . Proof that this call to returns true is satisfiable:
      * Assume that the input 3CNF circuit was satisfiable, i.e. such that the circuit returns True
        + In this case, we can construct the disjoint paths as follows:

Between variable nodes:

If was True in : ’s “true” path:

If was False in : ’s “false” path:

By the structure of , we know that all of the are node disjoint with respect to each other (they run “parallel” to each other)

Between clause nodes (for some , WLOG for ):

By the structure of , we know that there are three possibilities for :

Since is an or-statement, at least one must be true in order for to be true

If literal is not negated in :

is equal to the truth value of its underlying variable , thus for to be true we need to be true as well

By the structure of the only path available to us is (non-negated literals take the “false” path)

Clearly, this path does not intersect with the occupied “true” path of , and these paths are node disjoint

If literal is negated in :

is equal to the opposite truth value of its underlying variable , thus for to be true we need to be false

By the structure of the only path available to us is (negated literals take the “true” path)

Clearly, this path does not intersect with the occupied “false” path of , and these paths are node disjoint

* + - * + Thus we have constructed node-disjoint paths that connect each source-terminal pair, which means that our call to will return True when the input circuit is satisfiable
      * Assume that there are node-disjoint paths connecting and in .
        + We can now construct a satisfying variable assignment that makes the 3SAT circuit return true:

If uses the “true” path of , we set to be true.

In this case, all the clauses that contain a non-negated will be true

If uses the “false” path of , we set to be false.

In this case, all the clauses that contain a negated will be true

Since all variables are either be negated or non-negated in every clause, this assignment essentially sets all clauses to true, which makes the whole circuit true

* + - * + Thus, if we have node disjoint paths, we can be certain that the input 3CNF circuit is satisfiable
    - We have shown that the result of calling is true iff the input circuit to 3SAT was satisfiable.
    - Thus, the result of our call to on a graph built in time polynomial with respect to is logically equivalent as to whether the 3CNF circuit is satisfiable
    - Therefore, we have shown that (the inputs from 3SAT can be converted to the inputs of in poly-time, and the output of called on these inputs is equal to whether the input circuit belongs in ).
  1. Proof of Corollary that :
* Proof that
  + Proof that is polynomial sized
    - We let be the list of (supposed) node disjoint paths that link the - many source and terminal nodes together
      * Each path is represented according to nodes:
        + Example: represent , using list
        + Representation has size per-path (we can’t have more than nodes in a path)
      * There are -many paths in the path-list (we need one path per pair, and there are such pairings)
    - Thus, the total size of is , which is polynomial with respect to the size of inputs and
  + Proof that runs in polynomial time
    - Algorithm:

# Keep a lookup array of intermediate nodes to keep track of what

# we’ve already seen for O(1) lookup time

# Assume that the graph’s nodes have id numbers 1…|V|

# Assume that s\_nodes and t\_nodes are aligned and have indices 1…k

# Assume that the certificate’s paths are aligned with

# s\_nodes/t\_nodes (i.e. the i-th path connects s\_i to t\_i)

def V\_NDP(graph, s\_nodes, t\_nodes, gamma\_NDP):

# Array of size |V| to keep track of seen nodes

seen\_nodes = [False] \* len(graph.nodes)

# This is k

k = len(s\_nodes)

for i in range(k):

s\_t\_pair = (s\_nodes[i], t\_nodes[i])

cur\_path = gamma\_NDP[i]

# Check if the path connects s\_i with t\_i

if (cur\_path[0], cur\_path[-1]) != s\_t\_pair:

return False # abort: path fails to connect s\_i -> t\_i

# Check all the path’s nodes

else:

# Check if any node has been seen in another path

for node in cur\_path:

# Abort if we’ve seen this node in another path

if seen\_nodes[node.id\_number]: return False

# Indicate we’ve seen the node

else: seen\_nodes[node.id\_number] = True

# At this point we know that all the paths connect the s,t

# nodes they say they’re supposed to, and that none of them

# have the same nodes

return True

* + - * + This algorithm has runtime , which is polynomial with respect to the size of inputs :

Outer loop runs times (one for each path in the certificate)

Inner loop runs times (one for each node in a path, paths have length , and only executes constant-time instructions

Thus the outer loop runs in time. This is the dominant runtime because the rest of the algorithm executes instructions with lower time-bounds.

* + Proof that : verified in the previous sub-question, where we proved that (we know ).
* Since we have shown that and that some language () reduced to , we can conclude that
  1. -Space Classification of
     + Checking the validity of proposed solutions to can be done in poly-time using a similar algorithm to (instead of keeping track of seen nodes, keep track of seen edges)
     + Can do a poly-time reduction of to with a similar graph conversion, but with the following changes:
       - Add 2 more nodes to each true/false path of each
       - Instead of “piping directly through some “true”/”false” path node, we pipe it through two neighbouring nodes in the “true”/”false” path node
       - The above ensures that our source-terminal paths share edges in the event of satisfiability
     + Since we can verify in polynomial time and , we can say that

1. K-Coloring
   1. Poly-time Algorithm to Decide 2-Color-ability
      * English Description
        + We do a breadth first search from some arbitrary node with the following modifications:
          - Color with . Let be the opposite color of , and vice versa.
          - For every uncolored node we encounter in the algorithm, color it using the color opposite to its parent

Example: we color the neighbours/“children” of with T (since has color ), and we color the “grandchildren” of with (as their parents had color ), and so on

* + - * + If we encounter a node that has been colored already, and this node has the same color as the color we are about to use, we abort the algorithm – the graph cannot be 2-colored
        + If we don’t run into the above case at all during our run of BFS, our graph can be 2-colored
    - Correctness
      * Assume we encounter a node that has the same color as a node we just painted
      * Then in order to properly color it with the current color, we’d have to alternate the colors of its neighbours
      * We can’t do this as we would have to keep alternating colors of the neighbours’ neighbours and so on to keep the coloration property
      * If we did the above, we’d effectively reverse the coloration of the entire graph
      * Thus there’s no way paint this node
    - Running Time:
      * BFS has a runtime of (CLRS 22.2)
      * We can store colors as attributes or in an array indexed by node ID number for color inspection and modification time
      * The only modifications that we did to BFS are the addition of color assignments and neighbour/parent color inspections, which each take time
      * These color operations do not need to be put inside their own loops
        + Assignment can be done after popping
        + Neighbour color inspections can be done within in the inner-for loop of BFS (exploring the current node)
        + Parent color is trivial to inspect (we can also keep track of parents/discoverers when we push nodes into the queue)
      * Thus, the runtime is still , as only function calls were added
  1. Exam Scheduling NP-Completeness (let be the curly (number of students), I can’t find it in Word)
     + Language Formulation
       - Let:
         * be a subset of exams scheduled for timeslot
         * be a subset of exams that student needs to take
     + Proof that
       - Proof that :
         * Proof that is polynomial sized

We let our certificate for this problem be an exam schedule

We can represent an exam schedule as an 2D array , where:

* + - * + Proof that runs in polynomial time

Algorithm:

# Let each student’s final exams be represented as a bit-vector F

# – if they have a final in some course i,

# F[i] = 1, otherwise 0

# Let each timeslot be a bit-vector S, where S[i] = 1

# if F\_i is running during this timeslot

def V\_EXS(students, finals, gamma\_EXS):

# Go through every student

for student in students:

# Check the exam schedule

for timeslot in gamma\_EXS:

# If the student has more than 1 final during the

# selected timeslot, dot product of vectors

# will exceed 1

if dot(student.courses, timeslot) > 1:

return False

# At this point we’ve verified that no student has a conflict

return True

Runtime of the above verifier is polynomial:

Student for-loop runs times

Timeslot for-loop runs times

Dot-product of two -length vectors takes time

Thus time-slot for-loop runs in time

Thus student for-loop and therefore the whole algorithm runs in time, which is clearly in polynomial time with respect to input sizes

* + - * + Since we have proven that is polynomial sized, and the runtime of is polynomial, we can say that
      * Proof that – pick . Proof that :
        + An algorithm to solve that uses some :

Let the inputs to be:

Graph , where are arbitrary nodes/edges

Number of colors

Convert inputs from in polynomial time:

Represent each node as a final exam (conversion takes time, looping over nodes)

Represent each edge as a student taking final exams and (conversion takes time, looping over edges)

Let (number of colors) be (number of timeslots) (simple assignment takes time)

Solve the converted with some

If it returns true it means:

Given timeslots, it’s possible to schedule exams such that every student has at most 1 exam per time slot

Rephrasing: given timeslots, every student is “connected” to two exams, which each occupy separate timeslots

Reversing input translation: given colors, every edge is connected to two nodes, which each have a separate color

Rephrasing the translation: given colors, no two nodes of the same color are connected

The above sentence is logically equivalent to the input graph being -colorable

If it returns false, it means:

It’s not possible to construct such an exam schedule

Thus there exists some student that has more than 1 exam in the same time slot

Rephrasing: there exists some student that “connects” two exams in the same time slot together

Reversing the input translation: there exists some edge that connects 2 nodes of the same color together

This violates the -coloring property, thus it’s not possible to -color the input graph

* + - * + Therefore, we can use an algorithm that solves to solve , by converting between their respective inputs and outputs in polynomial time.
        + Thus, we can say that . In other words, we have found a language in that reduces in polynomial time to .
      * We have verified that and , which means that we can conclude that

1. Deciding Daggers

Note: (inclusive slicing), and

* 1. Semantic Array
     + = whether or not the word belongs to
     + Answer: contained inside (whether the whole word belongs to
  2. Computational Array
  3. Proof of Array Equivalence
     + Base Case (): this subword is the empty string , which we know is inside by convention, thus we return True
     + Recursive Case (:
       - We note that . Proof:
         * (definition from assignment)
         * (expanding the union)
         * (recursive definition of language powers)
         * (“factoring” out )
         * (re-expressing the infinite union of powers)
         * (the infinite union is the dagger)
         * (since is already included in the definition of )
       - This means that any word in is the concatenation of a word in with a word in
         * Formally:
       - Thus to check if a word is in , we need to have the following two conditions be true (for any way of splitting the word in two)
         * The first part of the word is in
         * The remainder of the word is in
       - This is essentially what we are doing in the computational array: we check whether such that the following two conditions are met:
         * is true (i.e. the first part of the word up to index is part of )
         * is true (i.e. the remaining part of the word (from indices to ) is part of )
  4. Runtime of Algorithm
     + Let:
       - ’s runtime be , where:
         * is the size of the input word to
         * is some positive, non-decreasing function bounded by a polynomial (has to be bounded because question specifies that is a poly-time algorithm)
       - ’s size be (input word)
     + Each element in the computational array essentially has to run a for-loop for iterations.
       - During this for-loop (call some arbitrary iteration ), we do the following:
         * Access . This takes time
         * Call . This takes time.

We call on an instance of size

This means that we experience a runtime of

We note that , and since is non-decreasing function,

Thus this step takes time

* + - * + Clearly, the total of the steps inside this for-loop is in time
      * Since the for-loop runs iterations, the total runtime of the loop is

* + - We note that the computational array has elements, and since each element has to run the aforementioned for-loop, we end up with a final running time of:
    - This run-time is clearly polynomial with respect to the input size ()
      * is polynomial with respect to
      * is polynomial with respect to (if is the bounding polynomial of , is the bounding polynomial of )
    - Therefore, if we can decide whether a word belongs in in polynomial time, we can also decide whether it belongs in in polynomial time

1. Minesweeper Satisfiability
   1. Language Formulation
      * Let:
        + be the labelling function (nodes with no label are labelled )
        + be the mine-placement function (may only place on nodes labelled )
        + be a function that returns the following:
          - number of mines placed on the neighbours of
   2. Reduction
      * Input conversion from :
        + Let:
          - be the variables in the circuit, and be an arbitrary variable
          - be the clauses in the circuit, and be an arbitrary clause
        + Create the following mine-graph :
          - Nodes :

For each variable , add the following nodes to :

(“true” node for )

) (“middle” node for )

(“false” node for )

For each clause , add the following nodes to :

(“true” node for )

) (“middle” node for )

(“false” node for )

* + - * + Edges:

For each variable add the following edges:

For each clause , where (WLOG for literals ) , add the following edges:

Between the clause’s nodes:

Between and the three literals’ nodes (where , and is the underlying variable of ):

If is NOT negated:

If is negated:

* + - * + Labelling:

For each variable , set

For each clause , set

* + - * + Time to create this graph is polynomial:

Each variable requires the introduction of:

3 nodes:

2 edges:

1 labelling:

Each clause (with being the underlying variables of literals ) requires the introduction of:

3 nodes:

5 edges:

1 labelling:

Since each clause and variable introduces a constant number of nodes/edges/labels, this conversion from the inputs of to the inputs of clearly takes polynomial time

* + - We execute to solve for the converted outputs, thus solving . Proof that this call to returns true the input circuit is satisfiable:
      * Assume that the input circuit was satisfiable. We can place mines on such that returns true. Proof:
        + We can use the following partial mine placement plan to satisfy the restrictions (for every variable in a satisfying assignment ):
        + Under this partial mine placement plan, if the circuit is satisfiable, mines can be placed around every such that . Proof:

If the circuit was satisfiable, all ’s contained a literal that was true

Let be a true literal in with underlying variable .

If was negated

had to be false for to be true

In this case had node

Under the partial placement plan, if ,

If was not negated

had to be true for to be true

In this case had node

Under the partial placement plan, if ,

Thus, under the partial placement plan

Since is connected to the not yet mine-occupied nodes , we can always make to satisfy by placing the (at most 2) “missing” mines on

* + - * Assume that the call to returned true. Proof that this implies the input circuit is satisfiable:
        + Let:

= underlying variables of literals

be some arbitrary literal, with underlying variable

* + - * + Each has a mine on some “variable node” . Proof:

By the structure of , is connected to the following nodes:

“Clause nodes”:

“Variable nodes”:

We have to place three mines on these nodes for to be satisfied

We note that two of the nodes are “clause nodes”

Thus, we always have at least one leftover mine that needs to be put on a “variable node”

* + - * + Recall the labelling for that we constructed in the poly-time conversion from :

Thus, (otherwise the labelling above would be violated)

* + - * + We can use the values of each to create the circuit-satisfying assignment :

Since only one of the “true”/”false” nodes of can have a mine, these assignment, these assignment rules always result in a certain value for

* + - * + Proof that the above assignment rule satisfies the circuit:

At least one of the “variable nodes” connected to has a mine on it

This node is either has to be or .

Assume that the node is .

By our input conversion, is only connected to when is NOT negated in

If is NOT negated, needs to be true for to be true

We have assumed that

Thus where needs to be true

Assume that the node is .

By our input conversion, is only connected to when is negated in

If is negated, needs to be false for to be true

We have assumed that

Thus where needs to be false

* + - We have shown that the result of calling is true iff the input circuit to 3SAT was satisfiable.
    - Thus, the result of our call to on a graph built in time polynomial with respect to is logically equivalent as to whether the 3CNF circuit is satisfiable
    - Therefore, we have shown that (the inputs from 3SAT can be converted to the inputs of in poly-time, and the output of called on these inputs is equal to whether the input circuit belongs in ).
  1. Proof of Corollary that :
* Proof that
  + Proof that is polynomial sized
    - We let be the mine-placement function that makes this graph (supposedly) mine-consistent
      * It can be represented as a hash-map/array that maps nodes values (whether there is a mine on the given node)
      * This map is of size , as at most nodes are unlabelled
      * Each element has size (1 or 0 is 1 bit)
    - Thus, the total size of is , which is polynomial with respect to the size of input
  + Proof that runs in polynomial time
    - Algorithm:

# Function to count the number of mine nodes around a given node

def sigma(graph, node, mu):

num\_mines = 0

for neighbour in graph.neighbours\_of(node):

num\_mines += mu[neighbour.id]

return num\_mines

# Verification function

def V\_MSW(graph, lambda, gamma\_MSW):

# Check each labelled node to see if sigma aligns with lambda

for node in graph.nodes:

if lambda[node.id] != -1:  
 if sigma(graph, node, gamma\_MSW) != lambda[node.id]:

# Abort: labels don’t line up!

return False

# At this point all labelled nodes are mine-consistent

return True

* + - * + This algorithm has runtime , which is polynomial with respect to the size of inputs :

Main for-loop does iterations (in the worst case it has to iterate over each node)

Sigma function runs in time

Each node in the graph has at most neighbours

Looking up the value of the neighbour takes time (since we can represent as a hashmap/array that maps nodes to values

Thus, the main-for-loop has a total runtime of

* + Proof that : verified in the previous sub-question, where we proved that (we know ).
* Since we have shown that and that some language () reduced to , we can conclude that