COMP 314 Class 20: PolyCheck and NPoly are identical

Recall the definitions of complexity classes PolyCheck and NPoly:

- PolyCheck: problems whose positive instances can be verified in polynomial time (using a polynomial-sized hint, if needed).
- NPoly: problems that can be solved in polynomial time by a nondeterministic program.

At first sight, these two definitions appear to rely on completely different properties. Remarkably, we will now prove that the two definitions define exactly the same complexity class:

Claim: PolyCheck and NPoly are identical. That is, any problem in PolyCheck is also in NPoly, and vice versa.

We will split the proof into the obvious two parts, and devote a separate section to each. The next section shows that PolyCheck is a subset of NPoly; the section after that shows that NPoly is a subset of PolyCheck.

1 Every PolyCheck problem is in NPoly

In this section, we will be proving

Claim: Every PolyCheck problem is in NPoly.

Proof of the claim: Let F be a problem in PolyCheck. Given an input I of length n, let $S(I)$ to denote a proposed solution to F. The solution is "proposed" because it still needs to be verified. We know the length of the solution $S(I)$ is bounded by some polynomial—say, $a(n)$. (Why? We know it's possible to verify the correctness of the output in polynomial time, so the output itself had better have polynomial length—otherwise we couldn't even read it in polynomial time.)

By the definition of PolyCheck, we might need a hint $H(I)$ before we can verify the output $S(I)$. We know the length of this hint is bounded by some polynomial—say, $b(n)$. (Why? The polynomial length of the hint is guaranteed directly in the definition of PolyCheck. Note that some PolyCheck problems don't need any hint, but that just means we can take $b(n) \equiv 0$, which is a perfectly good polynomial.)

Also by the definition of PolyCheck, we know there exists some Python program `verify.py`, which can take as input the strings $S(I)$ and $H(I)$ concatenated together, and will output "`yes`" if and only if the alleged value of $S(I)$ is correct. And we know that `verify.py` will run in polynomial time, unless $S(I)$ = "`no`". Let's say the running time of `verify.py` is bounded by a polynomial $c(n)$, for positive instances.

Now we are going to use these building blocks to write a nondeterministic Python program `guessAndVerify.py` that computes a solution to F in polynomial time. The

program itself is shown below, beginning on page 3. The program might look intimidating, but the underlying idea is reasonably simple. We just use the power of nondeterminism to *guess* the solution (and the hint). The number of possibilities is ludicrously large: if there are A symbols in the alphabet, then there are $A^{a(n)}$ possible solutions, and $A^{b(n)}$ possible hints. This makes a total of $A^{a(n)+b(n)}$ combined possibilities. However, through the power of nondeterminism, we can try all of these possibilities is simultaneously. Any single thread in this computation just has to make $a(n) + b(n)$ choices (each choice selects one of the A possible symbols) for its guess of the solution and hint. Then, the guess needs to be verified, which takes time $c(n)$. So the total time needed by any one thread is at most the polynomial $a(n) + b(n) + c(n)$. ■

To understand the details, read through the listing of `guessAndVerify.py` on page 3. You can also look at the listing of a particular example of the `verify.py` program, on page 6. This verification program is used to find a single factor of the input. The input, solution, and hint are all expressed as binary strings. For example, one possible solution to the input "`110`" (decimal 6) is "`10`" (decimal 2). Of course, this simple example doesn't require a hint, since we can easily verify the solution by checking that 2 divides 6. To make the example more interesting—and in particular, to demonstrate how the hint can be used—this particular example assumes we are in a strange universe where division operations are extremely expensive, and we therefore only want to use multiplication operations in the verification program. So, if the input is 6 and the proposed solution is 2, the hint will be 3, and the verification program can check that $2 \times 3 = 6$. In general, for input M and proposed solution K, the hint will be $L = M/K$, and the program checks that $K \times L = M$.

```
# Python program guessAndVerify.py

# we need facilities in the threading and time modules
import threading
import time

# We assume a function to verify a solution, possibly given a hint, is
# provided, and import it here. The function
# verify(input,solnAndHint) takes two parameters and returns True or
# False. The second parameter is a string containing both the solution
# and, optionally, the hint. The hint, if present, is separated from
# the solution by a semicolon. Example: verify("abc", "xy;123") should
# return True if "xy" is the solution to input "abc", verified using
# the hint "123".
from verify import verify

# a(n) is the known polynomial bound on the length of the solution.
# As a concrete example, we assume here the solution is shorter than
# the input.
def a(n):
    return n-1

# b(n) is the known polynomial bound on the length of the hint. As a
# concrete example, we assume here the hint is shorter than the
# input.
def b(n):
    return n-1

# Except for very tiny problems, this program will create a vast
# number of threads -- too many for Python's threading module to cope
# with. Therefore, it's preferable to simulate the nondeterminism
# using recursive function calls instead. To use actual threads, set
# the following variable to False, but you will need to use very short
# inputs, and very small polynomials a and b above.
useRecursionToSimulateNondeterminism = True

# In principle, this program works with any alphabet. In practice,
# since we will be guessing all possible solutions and hints up to a
# given length, we need to keep the alphabet small so that the number
# of possibilities is manageable. Hence, we restrict to binary strings
```

```
    # involving '0' and '1', but we also permit the semicolon since it
42  # will be used as a separator between solution and hint when they are
    # concatenated.
44  alphabet = ['0', '1', ';']


46

    # nondetGuessAndVerify nondeterministically guesses and verifies all
48  # possible solutions and hints that start with the given string
    # solnAndHintGuess. If a correct solution is found, the global
50  # variable answer is updated with the solution.
    def nondetGuessAndVerify(input, solnAndHintGuess):
52      # By assumption, the solution has length at most a(n), and the
        # hint has length at most b(n), so when we concatenate them with a
54      # semicolon separator, the total length of the guess is at most
        # a(n) + b(n) + 1. Hence, we can immediately give up and return if
56      # our current guess is longer than that.
        n = len(input)
58      if len(solnAndHintGuess) > a(n) + b(n) + 1:
            return
60

        # Try all possible extensions of the current guess. That is,
62      # append each possible symbol to the current guess, and launch new
        # threads to analyze each of these possibilities.
64      for symbol in alphabet:
            # append the chosen symbol to previous guess
66          newGuess = solnAndHintGuess + symbol
            # As noted above, the number of threads becomes
68          # overwhelming except in trivial examples, so we can
            # simulate nondeterminism with a recursive function call
70          # if desired.
            if useRecursionToSimulateNondeterminism:
72              nondetGuessAndVerify(input, newGuess)
            else:
74              t = threading.Thread(target=nondetGuessAndVerify, \
                                     args = (input, newGuess))
76              t.start()


78

        # Now we use the verify function to find out if our current guess
80      # is correct. This takes at most time c(n).
        foundSolution = verify(input, solnAndHintGuess)
```

```
82
        # If we have guessed a correct solution, update the global
84      # variable answer.
        if foundSolution:
86          global answer
            # The solution consists of the part of solnAndHintGuess before
88          # the semicolon, if one is present.
            answer = solnAndHintGuess.split(';')[0]
90

92  # guessAndVerify is the driver function that will launch our
    # nondeterministic version of the same function, nondetGuessAndVerify.
94  def guessAndVerify(input):
        # Initialize the global variable answer to 'no'. If any of the
96      # threads finds a solution, it will store the solution in this
        # variable.
98      global answer
        answer = 'no'
100
        # Our initial guess for the solution, and the hint, consists of
102     # the empty string.
        solnAndHintGuess = ''
104
        # Launch the nondeterministic computation
106     nondetGuessAndVerify(input, solnAndHintGuess)

108     # If we are using real threads (i.e. not simulating them via
        # recursion), we had better wait until they have all finished.
110     if not useRecursionToSimulateNondeterminism:
            while threading.activeCount() > 1:
112             time.sleep(1)

114     # Return the answer
        return answer
116
    # Here's an example of the entire program in action
118 solution = guessAndVerify('1110')
    print solution
```

```
# Python program verify.py

# This function returns true if the proposed solution is correct for
# the given input. The solution can be computed using a hint. The
# solution and hint are concatenated into a single string parameter,
# solnAndHint, separated by a semicolon. In this particular function,
# we are verifying that the solution is a factor of the input (when
# both are interpreted as binary strings). To make the hint
# meaningful, we are pretending that division is too expensive, so the
# verification can use only multiplication. Thus, if the input is '1111'
# (decimal 15), one possible solution is '101' (decimal 5), in which
# case the hint should be '11' (decimal 3) -- so that we can verify
# the solution by computing 5*3==15.
def verify(input, solnAndHint):
    # Our verification strategy requires a hint, so return False if
    # the hint separator (';') is not present.
    if solnAndHint.count(';') != 1:
        return False

    # Split the solution and hint into separate strings, and return
    # False if either is empty.
    strings = solnAndHint.split(';')
    solution = strings[0]
    hint = strings[1]
    if len(solution)==0 or len(hint)==0:
        return False

    # Convert the binary strings into integer values.
    inputVal = int(input, 2)
    solutionVal = int(solution, 2)
    hintVal = int(hint, 2)

    # Verify that solutionVal * hintVal == inputVal. We also need to
    # rule out the trivial solutions (1, and the input value).
    if solutionVal>1 and solutionVal<inputVal and solutionVal * hintVal == inputVal:
        return True
    else:
        return False
```

2 Every NPoly problem is in PolyCheck

In this section, we will be proving

Claim: Every NPoly problem is in PolyCheck.

Proof of the claim: Let F be a problem in NPoly. That means there is a nondeterministic program P computing solutions to F in polynomial time. When P runs, we can imagine a tree of the threads it creates, as in Figure 1. Each node in this tree can number its children 0, 1, 2, 3, and so on. If we are given a positive instance of the problem, then at least one of the leaves of the tree represents a valid solution. (Leaves are shown as open circles in Figure 1.) There is some path from the root of the tree to this solution leaf, and we can identify that path by the numerical IDs of each node to pass through. For example, the path could be described by the sequence 2, 2, 4. This sequence of choices is precisely the hint that will be fed into our verifying program. Note that the length of the hint is a polynomial function of the input length, since any path from the root to a leaf takes polynomial time. So, the verification program is just a deterministic version of P: instead of launching new threads, it chooses exactly what to do next according to the hint it has been given. When it reaches the leaf, it can compare the solution there to the alleged solution it is meant to be verifying.

The total running time is polynomial since every path in the original, nondeterministic computation had polynomial running time. ■

We can add a little more detail to the above proof by thinking about what changes would need to be made to the code. Specifically, we can assume that any part of the nondeterministic program P that launches new threads looks something like this:

```
treeDepth = treeDepth + 1
for nondetChoice in range(numChoices):
    t = threading.Thread(target=someFunction, \
            args = (input, treeDepth, nondetChoice, someOtherParameters))
    t.start()
```

In the verification program, we change this code to:

```
treeDepth = treeDepth + 1
nondetChoice = hint[treeDepth]
someFunction(input, treeDepth, nondetChoice, someOtherParameters)
```

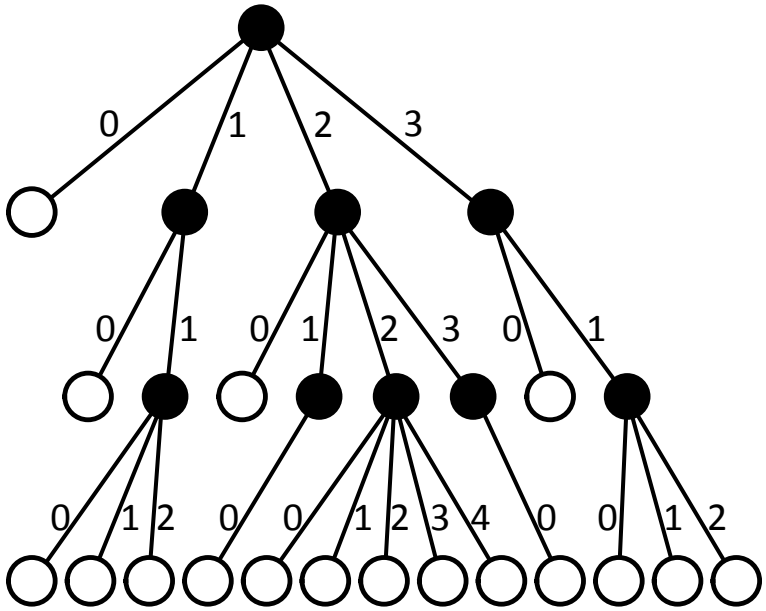

Figure 1: A tree representing the threads launched by a nondeterministic program. Leaves are represented by unfilled circles.