Greedy Algorithms

A greedy algorithm is an algorithm that makes a best choice "right here and right now". They are a very fast class of algorithms, because they usually involve moving through the input only once, making decisions along the way, without knowing the future. In some cases this may lead to a non-optimal solution, which can be an acceptable pay-off for the speed. In other cases, this may still guarantee an optimal global solution.

In a previous topic, we solved SAT by trying all possible assignments of the Boolean variables in the input formula. If we had moved through the variables one by one, assigning `True/False` to each one while trying to keep the formula `True`, there would be no guarantee of finding a solution. In fact, there is no known significantly better approach to SAT. However, under certain restrictions, there is a greedy algorithm that guarantees a correct solution (or a `False` conclusion if there is none). Whether or not you are interested in logic, this is a useful problem that will help you get a feel for fast algorithms versus slow algorithms.

2-SAT

A Boolean formula is in conjunctive normal form (CNF) if it is written as an `And()` function with `Or()` function arguments. Every formula can be written in such a way, and sympy can convert any formula to CNF, or test whether one is in CNF. For example:

```
import sympy as sp
from sympy.logic.boolalg import to_cnf
from sympy.logic.boolalg import is_cnf
x,y,z=sp.symbols('x y z')
is_cnf( ( x | y ) & ( ~ x | ~ y | ~ z ) & ( z ) )
   Out:   True
is_cnf( ( x & y ) | ( ~ x & ~ y & ~ z ) )
   Out:   False
to_cnf( ( x & y ) | ( ~ x & ~ y & ~ z ) )
   Out:   (x|~x)&(x|~y)&(x|~z)&(y|~x)&(y|~y)&(y|~z)
```

A 2-SAT problem is one where we need to find a solution to a formula in which each Or-clause has no more than two variables. So, the following is 2-SAT

```
(x|~x)&(x|~y)&(x|~z)&(y|~x)&(y|~y)&(y|~z)
```

but this one is not:

```
(x|y)&(~x|~y|~z)&(z)
```

Unlike, general satisfiability, 2-SAT can be solved in polynomial time.

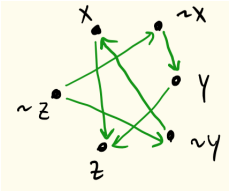
Polynomial time algorithm for 2-SAT

If one of the clauses of a 2-SAT formula is

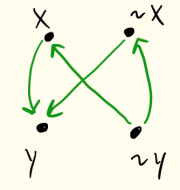
```
(x|y)
```

then another way of saying this is that "If x is False, then y must be True" and "If y is False, then x must be True". Or in short $\sim x \to y$ and $\sim y \to x$. As long as these statements are satisfied in our assignment, the clause is satisfied. For a the complete formula, we can create a little network of implied relationships. For example,

```
s1 = (x | y) & (z | ~x) & (~y |z)
```




```
s2 = (x | y) & (~x | y)
```



For any variable x, if there is a relation $x \to \sim x$ (this may not be a direct relation, but could be caused by multiple steps $x \to \ldots \to \sim x$), then it is impossible for x to be `True`, so we know that x must be `False`. Similarly, if there is a relation $\sim x \to x$, then x must be `True`. If both relations are present, then there is no assignment of x that will satisfy the formula. If neither are present, then x can be either `True` or `False`. This does not mean that we don't know what to assign to x, but rather that it does not matter! If the formula is satisfiable, it will have satisfying assignments where x is `True` and ones where x is `False`.

This is why the greedy algorithm works! We can go through the variables and make assignment choices along the way, based on local information about each variable, without having to worry about making a bad choice. Each time we assign `True/False` to a variable (and its negation), we must follow all the arrows in the diagram that come out of the `True` variable/negation, and assign `True` to all variables reached in this manner (and `False` to their negations). This guarantees that we obtain a good assignment that does not violate any of the implication arrows, and therefore does not violate the formula.

A complete proof can be found here[1], but I will give a short intuition. A problem that you might foresee is that if we set some variable x to `True`, then we might be forced to set both some y and its negation to `True`, which is impossible. This does not happen! If there is a path $x \to u \to \ldots \to y$, then there is also a path $\sim y \to \ldots \to \sim u \to \sim x$. (Why is that?) If there is also a path $x \to \ldots \to \sim y$,

[1] `https://www.iitg.ernet.in/deepkesh/CS301/assignment-2/2sat.pdf`

this leads to a path $x \to \ldots \to\sim y \to \ldots \to\sim u \to\sim x$. But if $x \to\sim x$, we were never supposed to assign x to `True` to begin with.

Let's put together the outline for our greedy, polynomial time algorithm.

```
2-SAT (s)
create the implication network from s
while there are unassigned variables:
   let x be the first unassigned variable
   find all variables that can be reach from x and from ~x
   based on whether x -> ~x and/or ~x -> x are True
      assign x to True or False, or
      return False
   if x/~x is True, set all other variables reached from it to True
   and their negations to False
```

To create the implication network, there is a useful function in sympy called `s.args`. It returns the arguments of the outer-most function of the formula `s`. Since we know the structure of `s`. Let us assume for convenience, that each clause has exactly two variables (or negations) in it. If any clause only has a single variable x, this is easily replaced with (x|x) without changing the meaning. For example, we can find out that s1 has 3 clauses, and that the second clause contains z and x. We can even use `args` to check whether a variable is negated or not, since a `Not()` function has an argument, whereas a non-negated variable does not.

```
s1 = (x | y) & (z | ~x) & (~y |z)
len(s1.args)
   Out:   3
s1.args[1].args[0]
   Out:   z
s1.args[1].args[1]
   Out:   ~x
if s1.args[1].args[0].args:
   print('negation!')
 Out:
if s1.args[1].args[0].args:
   print('negation!')
 Out:   negation!
```

In order to find a list of all the variables (or negations) that can be reached from a starting variable in the implication network, we can perform a Breadth First Search or a Depth First Search. For this purpose, there is very little difference. (They change the order in which nodes are found.) A BFS starts by creating a queue of nodes to be explored, containing just the starting node. It then repeatedly adds "new" neighbors of the first node in the queue to the queue and then moving it to a list of visited nodes. This ensure that every possible path from the starting node is explored. In pseudo-code:

```
BFS (N,x)
Q=[x]
V=[]
while Q:
   append to Q all nbrs of Q[0] that are not in Q or V
   remove Q[0] from Q and add it to V
return V
```

In order to use the network efficiently in a search, it might make sense to store it in a different way from the adjacency matrix that we are used to. The matrix was very useful for linear algebra, but right now we are only interested in obtaining lists of neighbors. We can store the network, for example, as a dictionary that keeps track of the direct implications. For example, for the network for s1:

```
ds1 = {x:[z],~x:[y],y:[z],~y:[x],z:[],~z:[~x,~y]}
```

Of course, you may also decide to index the variables instead. Then you can use lists. Take a moment to see how the following list stores the network:

```
Ns1 = [ [4], [2], [4], [0], [], [1,3] ]
```

Your choice of object to store information will depend on the action that you will be performing on it and sometimes the amount of space taken up by the different options.

Complexity analysis

Take some time yourself to decide carefully what the run-time complexity of your algorithm is, and show that it is indeed polynomial. I will update this space in a few days.