COMP 6651: Assignment 2 - partial solution

PROBLEMS.

2 You are given a set of m constraints over n Boolean variables $x_1, \ldots, x_n$. The constraints are of two types:

- **equality constraints**: $x_i = x_j$ for some $i \neq j$;
- **inequality constraints**: $x_i \neq x_j$ for some $i \neq j$.

Design an efficient **greedy** algorithm that given the set of equality and inequality constraints determines if it is possible or not to satisfy all the constraints simultaneously. If it is possible to satisfy all the constraints, your algorithm should output an assignment to the variables that satisfies all the constraints.

(1) Choose a representation for the input to this problem and state the problem formally using the notation **Input: ...**, **Output: ...**.

(2) Describe your greedy algorithm in plain English. In what sense is your algorithm "greedy"?

(3) Describe your greedy algorithm in pseudocode.

(4) Briefly justify correctness of your algorithm.

(5) State and justify the running time of your algorithm. The more efficient algorithm the better. Inefficient solutions may lose points.

Solution.

(1) Represent equality constraints by an array of lists $EQ[1..n]$, where $EQ[i]$ is a list of indices $(j_1, \ldots, j_k)$ indicating that $x_i = x_{j_1}, x_i = x_{j_2}, \ldots, x_i = x_{j_k}$ and these are all the equality constraints involving variable x_i.

Similarly, represent inequality constraints by an array of lists $IN[1..n]$, where $IN[i]$ is a list of indices $(\ell_1, \ldots, \ell_p)$ indicating that $x_i \neq x_{\ell_1}, x_i \neq x_{\ell_2}, \ldots, x_i \neq x_{\ell_p}$ and these are all the inequality constraints involving variable x_i.

The problem can be stated formally as follows:

Input: $EQ[1..n], IN[1..n]$ – two arrays of lists representing equality and inequality constraints on n Boolean variables, as described above.

Output: $A[1..n]$ – array of 0s and 1s representing an assignment to the n variables satisfying all the constraints, if such an assignment exists. Otherwise, output should be "no satisfying assignment."

(2) Set the first variable to 0 and propagate this assignment to all variables in $EQ[1]$ and the opposite values in $IN[1]$. If there is a conflict, return "no satisfying assignment," otherwise pick one of the variables to which the assignment has been propagated, and repeat. This can be implemented with the help of a queue. If all variables which could be reached from the first variable has received an assignment and there was no conflict, pick another unassigned variable and repeat the process.

This algorithm is greedy in the sense of trying to satisfy as many constraints as the propagation process gets exposed to and it does so via *irrevocable* decisions (we never go back and adjust our assignments to previously considered variables).

(3) The pseudocode implementing the above idea is below.

---

procedure $FindAssignment(EQ[1..n], IN[1..n])$
  instantiate array $A[1..n]$
  for $j = 1$ to n **do**
    $A[j] \leftarrow -1$
  $Q \leftarrow \emptyset$ – queue
  for $j = 1$ to n **do**
    if $A[j] = -1$ **then** ▷ Found a new variable, initialize propagation
      $A[j] \leftarrow 0$
      $Q.push(j)$
      while $Q.size() \neq 0$ **do**
        $c \leftarrow Q.pop()$
        for $x \in EQ[c]$ **do**
          if $A[x] \neq -1$ and $A[x] \neq A[c]$ **then**
            ▷ x was assigned before and violates equality constraint with c
            return "no satisfying assignment"
          else if $A[x] = -1$ **then**
            ▷ x wasn't assigned before and should be equal to c
            $A[x] \leftarrow A[c]$
            $Q.push(x)$ ▷ propagate assignment to x further
        for $x \in IN[c]$ **do**
          if $A[x] \neq -1$ and $A[x] = A[c]$ **then**
            ▷ x was assigned before and violates inequality constraint with c
            return "no satisfying assignment"
          else if $A[x] = -1$ **then**
            ▷ x wasn't assigned before and should receive opposite value of c
            $A[x] \leftarrow 1 - A[c]$
            $Q.push(x)$ ▷ propagate assignment to x further
  return A

---

(4) First of all observe that if $A[1..n]$ is a satisfying assignment then $B[1..n]$ defined by $B[j] = 1 - A[j]$ is also a satisfying assignment. That is, flipping the value of each variable preserves all equality and inequality constraints. Therefore, when we encounter an unassigned variable outside of the propagation process we have the freedom of whether to assign it 0 or 1 without loss of generality. We chose to assign 0. After that all variables reachable from the current variable via a sequence of equality or inequality constraints are completely determined. If x is reachable from c via a single equality constraint and c got value 0 then in order for the assignment to be satisfying x has to be assigned 0 as well. Other cases (when c is 1 or when we deal with inequality constraint) are similar. Thus, we continue propagating

the values until we either encounter a contradiction or we finish the propagation routine without encountering a contradiction. A contradiction simply means that there is a chain of constraints such that when put together imply that a variable must be equal to its negation, which is clearly impossible, so the set of constraints is unsatisfiable. If we haven't encountered a contradiction, we have to check if there are some other unassigned variables remaining and repeat the procedure.

(5) Let m be the total number of constraints (both equality and inequality). Each variable is placed into queue Q at most once since we check $A[x] = -1$ first and then assign value to $A[x]$ (either 0 or 1) prior to placing it into Q. Each constraint involving two variables x_i and x_j can therefore be examined at most twice – once when x_i is popped from the queue and its list of constraints is considered, and once more when x_j is popped from the queue and its list of constraints is considered. Therefore, the overall running time is $O(n + m)$.

3 Consider the Longest Increasing Subsequence (LIS) problem (see slides for problem definition).

(1) Give a greedy algorithm for LIS. You should: firstly describe your algorithm in English, then write down the pseudocode, and finally point out its time complexity.

(2) Your algorithm most likely is not optimal. Give one example on which your greedy algorithm outputs an optimal solution. Given another example on which your greedy algorithm outputs a suboptimal solution.

Solution Sketch.

(1) DESCRIPTION OF A GREEDY ALGORITHM (note: you may give a different greedy algorithm): Let the sequence be $A[1..n]$. Start by picking $A[1]$ into our desired subsequence, then linearly scan the array from left to right, and pick $A[i]$ as long as it is the first element that is $> A[1]$, then continue this scan and pick $A[j]$ as long as it is the first element that is $> A[i]$, etc, continue in this way until the end of the array. This algorithm is greedy because: (a) every local step that we take is trying to grow the current increasing subsequence, i.e., it is localling maximizing the length of the current increasing subsequence; (b) the algorithm makes irrevoable decisions on whether or not to put an element into the desired increasing subsequence.

PSEUDOCODE: omit.

TIME COMPLEXITY: $O(n)$, since we perform a linear scan.

(2) EXAMPLE ON WHICH MY ALGORITHM OUTPUTS AN OPTIMAL SOLUTION: $[1, 2, 3, 4, 5, 6, 7, 8]$. My algorithm will output the optimal solution which is the whole sequence.

EXAMPLE ON WHICH MY ALGORITHM OUTPUTS A SUBOPTIMAL SOLUTION: $[8, 7, 6, 5, 1, 2, 3, 4]$. My algorithm will output an increasing subsequence consisting of a single number 8, but the optimal solution is $[1, 2, 3, 4]$ of length 4.