CS140: Algorithms

Fall 2000

Solutions to Assignment 6

1. An *independent set* in a graph $G = (V, E)$ is a subset of vertices, $W \subseteq V$, such that no pair of vertices in W have an edge in G. The problem of finding a largest independent set of a graph is NP-hard. The problem is tractable, however, when the input is restricted to forests.

   (a) Let F be a forest with n nodes. Suppose v is a leaf in some tree of F. Prove that some largest independent set of F contains v.

   Let W be a largest independent set of F. If v is not in W then W must contain a vertex u that is adjacent to v. Since v is a leaf and adjacent to u, it is not adjacent to any vertex in W except u. Thus $W - \{u\} + \{v\}$ is a largest independent set that contains v.

   (b) Suppose you are given a black box that will compute a largest independent set for any forest containing $n - 1$ or fewer nodes. How could you use this black box to help construct a largest independent set for a forest containing n nodes.

   Find a leaf v in F. If v has degree zero, let $F' = F - \{v\}$. Otherwise let u be the node adjacent to v and let $F' = F - \{u, v\}$. Use the black box to compute a largest independent set W' of F'. Then $W = W' + \{v\}$ is a largest independent set of F.

   (c) Give a recursive algorithm for finding a largest independent set in a forest. (Be sure to specify your base cases.)

   `algorithm` LargestIndependentSetInForest(F)

   > `If` F `is empty return` ϕ
   >
   > `Let` v `be a leaf in` F
   >
   > `If` v `has degree 0 let` $F' = F - \{v\}$
   >
   > `Else let` u `be the neighbor of` v `and let` $F' = F - \{u, v\}$
   >
   > `Return` $\{v\}$+`LargestIndependentSetInForest(`F'`)`

   `end algorithm`

   (d) What is the running time of your algorithm. Give a brief sketch of the data structures you need to achieve this running time.

   We can achieve an $O(n)$ running time, where n is the number of vertices, by using the following data structures:

   i. An adjacency list representation of F with head pointers stored in an array indexed by vertex number; i.e. Vertex[i] points to a linked list of vertices that are adjacent to vertex v_i in F. The records in the list are doubly linked. Further, if v_i and v_j are connected by an edge, then the records for v_i in Vertex[j] and v_j in Vertex[i] are doubly linked.

ii. An array, Degree, that stores the degree of each vertex in F.

iii. A double linked list, Leaves, containing the leaves of F; we'll assume standard stack operations of push and pop are supported for this list. Also, an array of pointers into Leaves; i.e IsLeaf[i] is NULL if v_i is not a leaf and otherwise points to the record for v_i in Leaves.

Now we'll fill in implementation details and analysis of running time.

`If` F `is empty return` ϕ

Note that F is empty if and only if Leaves is empty. We can check if Leaves is empty in $O(1)$ time.

`Let` v_i `be a leaf in` F

Pop Leaves in $O(1)$ time.

`If` v_i `has degree 0 let` $F' = F - \{v_i\}$

If Degree[i] $= 0$ set Degree[i] $= -1$. (We'll use this to signify the vertex has been removed.)

`Else let` v_j `be the neighbor of` v_i `and let` $F' = F - \{v_i, v_J\}$

We can find v_i's neighbor in $O(1)$ time from Vertex[i].

Remove v_i from F: Set Degree[i] $= -1$; Remove the record of v_i from the list at Vertex[j]; Set Vertex[j] to NULL. This takes $O(1)$.

Remove v_j from F

Remove each edge (v_j, v_k) incident to v_j:

Remove the record of v_j in Vertex[k] and decrement Degree[k].

If Degree[k] $= 1$ create a new leaf record for v_k, link it from IsLeaf[k], and push the record onto Leaves.

Remove the record of v_k in Vertex[j].

It takes $O(1)$ to remove each edge incident to v_i.

Now remove the vertex v_j:

Set Degree[j] $= -1$.

If IsLeaf[j] is not NULL, remove the leaf record for j from Leaves.

This takes $O(1)$ time.

`Return` $\{v\}$+`LargestIndependentSetInForest(`F'`)`

The running time is $O(n+m)$ where n is the number of vertices and m is the number of edges of F. Since F is a forest $m \leq n$ so the algorithm runs in time $O(n)$.

2. A boolean formula is *satisfiable* if there is some truth assignment for the underlying variables that causes the formula to evaluate to True. For

example, the formula $(x_1 + x_2)(\bar{x_1} + \bar{x_2})$ is satisfiable; $x_1 = T, x_2 = F$ is a satisfying assignment. On the other hand $(x_1 + x_2)(\bar{x_1} + x_2)(x_1 + \bar{x_2})(\bar{x_1} + \bar{x_2})$ is not satisfiable; none of the four possible truth assignments cause the formula to evaluate to True.

The problem of determining whether or not a boolean formula is satisfiable is NP-complete. There are some restricted versions of the problem, however, that are tractable. One is called 2-SAT. The input to 2-SAT is a boolean formula such that the formula is in *conjunctive normal form* and each clause contains at most two *literals*. *Conjunctive normal form* means the formula consists of some "or" clauses that are "anded" together. A *literal* is one of the underlying variables or its complement. The two examples given earlier are both instances of 2-SAT.

(a) First consider the problem 1-SAT; here the formula is in conjunctive normal form and each clause has exactly one literal. For example $(x_1)(x_2)(\bar{x_2})$ is a 1-SAT formula. Give necessary and sufficient conditions for a 1-SAT formula to be satisfiable.

The formula does not contains clauses (v_i) and $(\bar{v_i})$ for any variable v_i.

(b) Suppose you have a black box that determines whether or not a 2-SAT formula with $n - 1$ or fewer variables is satisfiable. How could you use this box to determine whether or not a 2-SAT formula with n variables is satisfiable.

Choose a variable v_i.

If the formula contains clauses (v_i) and $(\bar{v_i})$ then the formula is not satisfiable. Else continue.

If F does not contain any clauses $(\bar{v_i})$ try setting v_i to True: Remove all clauses containing v_i. Remove $\bar{v_i}$ from all remaining clauses. Test if the resulting formula is satisfiable. If so, then F is satisfiable. Else continue.

If F does not contain any clauses (v_i) try setting v_i to False: Remove all clauses containing $\bar{v_i}$. Remove v_i from any remaining clauses. Test if the resulting formula is satisfiable. If so, then F is satisfiable. Else continue.

F is not satisfiable.

(c) Give an algorithm for solving 2-SAT. (Your algorithm only has to answer Yes or No; it doesn't have to construct a satisfying assignment.)

Replace the "test if the resulting formula is satisfiable" in the previous description by a recursive call.

(d) What is the running time of your algorithm?

The running time is $O(2^n)$. We can, however, achieve a running time of $O(nm)$, where n is the number of variables and m is the number of clauses by always choosing a variable v_i that is forced when one exists; by forced we mean the formula contains a clause (v_i) or $(\bar{v}_i)$. This solution was discussed in class.