LINEAR TIME MAXIMUM INDUCED MATCHING ALGORITHM FOR TREES

MICHELE ZITO
Department of Computer Science
University of Liverpool
Liverpool L69 7ZF, UK
M.Zito@csc.liv.ac.uk

Abstract. A matching in a graph $G$ is a collection of non-intersecting edges. The matching is induced if no two edges in the matching are joined by an edge in $G$. This paper studies the complexity of finding a largest induced matching when the input graph is a tree. We describe the first linear time algorithm for this problem.

CR Classification: G.2.2, F.2.2

Key words: graphs, matching, induced, trees, algorithm

1. Introduction

If $G = (V, E)$ is a graph, a set $M \subseteq E$ is a matching if $e_1 \cap e_2 = \emptyset$ for all $e_1, e_2 \in M$. Let $V(M)$ be the set of vertices belonging to edges in the matching. A matching is induced if no two edges in the matching are joined by an edge in $G$. Let $\nu_I(G)$ denote the maximum cardinality of an induced matching in $G$. The maximum induced matching problem (MIM) is that of finding an induced matching in $G$ with $\nu_I(G)$ edges.

The problem was introduced by Stockmeyer and Vazirani [1982] as a variation of the maximum matching problem and motivated as the "risk-free" marriage problem: find the maximum number of pairs such that each married person is compatible with no married person other than the one he (or she) is married to. Induced matchings have stimulated a lot of interest in discrete mathematics because finding large induced matchings is a subtask of finding a strong edge-colouring in a graph [see Erdős 1988, Steger and Yu 1993, Liu and Zhou 1997, for more recent results], a proper colouring of the edges such that no edge is adjacent to two edges of the same colour. MIM is NP-hard even for bipartite graphs of maximum degree four [Stockmeyer and Vazirani 1982]. Zito [1999] presented some positive and negative results concerning the approximability of $\nu_I(G)$ for various classes of bounded degree graphs.

A graph $G = (V, E)$ is a tree if it is connected and it has no cycle, it is chordal if any cycle of at least four vertices contains an edge connecting two non-consecutive vertices. Cameron [1989] showed that MIM in chordal

*Supported by EPSRC grant GR/L/77089.

Received September 16, 1999; accepted March 31, 2000.
graphs can be reduced to finding the largest independent set and an algorithm by Gavril [1972] can be used to solve optimally the latter in polynomial time. Since trees are chordal graphs, this argument (and an efficient implementation of Gavril’s algorithm) implies the existence of a $O(|V|^2)$ algorithm for finding a maximum induced matching in a tree. In Section 2 we present an alternative algorithm which again solves MIM optimally if the input graph is a tree but it runs in $O(|V|)$ time. To stress the simplicity of our approach, in Section 3 we also give a C++ implementation of the same algorithm using the library of efficient data types and algorithms LEDA [see Mehlhorn et al. 1999], to represent the trees and to implement some basic data structures.

2. Linear time algorithm

Although NP-hard for several classes of graphs including planar or bipartite graphs of maximum degree four and regular graphs of degree four, the problem of finding a largest induced matching admits a polynomial time solution on trees [Cameron 1989]. The algorithmic approach suggested by Cameron reduces the problem to that of finding an independent set of maximum cardinality in a graph that can be defined starting from the given tree. If $G = (V, E)$ is a tree, the graph $H(G) = (W, F)$ has $|V| - 1$ vertices, one for each edge in $G$ and there is an edge between two members of $W$ if and only if the two original edges in $G$ are either incident or connected by a single edge. Notice that $|F| = O(|V|^2)$. Moreover each induced matching in $G$ is an independent set of the same size in $H(G)$. Efficient implementations of Gavril’s algorithm find a largest independent set in a chordal graph with $n$ vertices and $m$ edges in $O(n + m)$ time. Since the graph $H(G)$ is chordal, the largest induced matching in the tree can be found in $O(|V|^2)$ time. In this section we describe a simpler and more efficient way of finding a maximum induced matching in a tree. If $G = (V, E)$ is a tree we choose a particular vertex $r \in V$ to be the root of the tree (and we say that $G$ is rooted at $r$). If $v \in V \setminus \{r\}$ then parent$(v)$ is the unique neighbour of $v$ in the path from $v$ to $r$; if parent$(v) \neq r$ then grandparent$(v) =$ parent$(\text{parent}(v))$. In all other cases parent and grandparent are not defined. If $u = \text{parent}(v)$ then $v$ is $u$’s child. All children of the same node are siblings of each other. Let $c(v)$ be the number of children of node $v$. The upper neighbourhood of $v$ (in symbols UN$(v)$) is empty if $v = r$, it includes $r$ and all $v$’s siblings if $v$ is a child of $r$ and it includes $v$’s siblings, $v$’s parent and $v$’s grandparent otherwise. $E(\text{UN}(v))$ is the set of edges in $G$ connecting the vertices in $\text{UN}(v)$.

Claim 1. If $G = (V, E)$ is a tree and $M$ is an induced matching in $G$ then $|M \cap E(\text{UN}(v))| \leq 1$, for every $v \in V$.

Note that if $M$ is an induced matching in $G$, any node $v$ in the tree belongs to one of the following types with respect to the set of edges $E(\text{UN}(v))$: 

Type 1. the node $\{v, \text{parent}(v)\}$ is part of the matching,

Type 2. either $\{\text{parent}(v), \text{grandparent}(v)\}$ or $\{\text{parent}(v), w\}$ (where $w$ is
some siblings of \( v \) belongs to the matching.

Type 3. Neither Type 1, nor Type 2, applies.

The algorithm for finding the largest induced matching in a tree \( G \) with \( n \) vertices handles an \( n \times 3 \) matrix \( \text{Value} \) such that \( \text{Value}[i,t] \) is the size of the matching in the subtree rooted at \( i \) if vertex \( i \) is of type \( t \).

**Lemma 1.** If \( G \) is a tree with \( n \) vertices, \( \text{Value}[i,t] \) can be computed in \( O(n) \) time for every \( i \in \{1, \ldots, n\} \) and \( t = 1, 2, 3 \).

**Proof.** Let \( G \) be a tree with \( n \) vertices. We assume \( G \) is in adjacency list representation. If \( h \) is the height of the tree, some linear preprocessing is needed to define an array \( \text{level}[i] \) (for \( i = 0, \ldots, h \)) such that \( \text{level}[i] \) contains all vertices at distance \( i \) from the root.

The matrix \( \text{Value} \) can be filled in a bottom-up fashion starting from the deepest vertices of \( G \). If \( i \) is a leaf of \( G \) then \( \text{Value}[i,t] = 0 \) for \( t = 1, 2, 3 \). In filling the entry corresponding to node \( i \in V \) of type \( t \) we only need to consider the entries for all children of \( i \).

1. \( \text{Value}[i,1] = \sum_{k=1}^{c(i)} \text{Value}[j_k,2] \). Since \( \{i, \text{parent}(i)\} \) will be part of the matching, we cannot pick any edge from \( i \) to one of its children. The matching for the tree rooted at \( i \) is just the union of the matchings of the subtrees rooted at each of \( i \)'s children.

2. \( \text{Value}[i,2] = \sum_{k=1}^{c(i)} \text{Value}[j_k,3] \). We cannot pick any edge from \( i \) to one of its children here either.

3. If node \( i \) has \( c(i) \) children then \( \text{Value}[i,3] \) is the maximum between \( \sum_{k=1}^{c(i)} \text{Value}[j_k,3] \) and a number of terms

\[
 s_{j_k} = 1 + \text{Value}[j_k,1] + \sum_{l \neq k} \text{Value}[j_l,2]
\]

If the upper neighbourhood of \( i \) is unmatched we can either combine the matchings in the subtrees rooted at each of \( i \)'s children (assuming these children are of type 3) or add to the matching an edge from \( i \) to one of its children \( j_k \) (the one that maximises \( s_{j_k} \)) and complete the matching for the subtree rooted at \( i \) with the matching for the subtree rooted at \( j_k \) (assuming \( j_k \) is of type 1) and that of the subtrees rooted at each of \( i \)'s other children (assuming these children are of type 2).

Option (3) above is the most expensive involving the maximum over a number of sums equal to the degree of the vertex under consideration. Since the sum of the degrees in a tree is linear in the number of vertices the whole table can be computed in linear time. \( \square \)

**Theorem 1.** MIM can be solved optimally in polynomial time if \( G \) is a tree.

**Proof.** The largest between \( \text{Value}[r,1] \), \( \text{Value}[r,2] \) and \( \text{Value}[r,3] \) is the size of the largest induced matchings in \( G \). By using an appropriate data structure it is also possible to store the actual matching. Details are given at the end of Section 3. The complexity of the whole algorithm is linear in the number of vertices of \( G \). \( \square \)
3. Implementation

In this section we present an implementation of the algorithm described in
Section 2. The code below computes the value of $\nu_1(G)$ for a given tree $G$. At
the end of this section we suggest how to modify the code so that it return
a maximum cardinality induced matching in $G$. The implementation was
designed in C++ and uses the library of efficient data types and algorithms
LEDA [see Mehlhorn et al. 1999], to represent the trees and to implement
some basic data structures.

```c
// The input is a tree, stored as a directed graph with all edges oriented
// away from the root.
#include <LEDA/tuple.h>
#include <LEDA/graph.h>
#include <math.h>
#include <LEDA/graph_alg.h>
#define MAX_DEPTH 100

main() {
    graph G;
    node u, v, w;
    int i, j, top;
    int depth;

    // The following function allows the user to input the tree as
    // a directed graph.
    test_graph(G);

    // Building the array of node layers.
    node_list layer[MAX_DEPTH];

    layer[0].append(G.first_node());
    for (i = 1; i < MAX_DEPTH; i++)
        if (layer[i-1].empty()) {
            depth = i-2;
            i = MAX_DEPTH;
        } else
            forall(v, layer[i-1])
                forall_adj_nodes(w, v)
                    layer[i].append(w);

    // Dealing with the array of values; 3 values for each node in the graph
    node_array<int> Value1(G,-1);
    node_array<int> Value2(G,-1);
    node_array<int> Value3(G,-1);

    int sum[MAX_DEPTH];
    int twos;
```
forall( v, layer[depth] )
Value1[v] = Value2[v] = Value3[v] = 0;

if (depth > 0)
  for( i = depth-1 ; i >= 0 ; i-- )
    forall( v, layer[i] ) {
    Value1[v] = 0;
    forall_adj_nodes(w,v)
    Value1[v] = Value1[v] + Value2[w];
    Value2[v] = 0;
    forall_adj_nodes(w,v)
    Value2[w] = Value2[w] + Value3[v];
    Value3[v] = 0;
    twos = 0;
    forall_adj_nodes(w,v) {
    Value3[v] = Value3[v] + Value3[w];
    twos = twos + Value2[w];
    }
    if (G.outdeg(v) > 0) {
    j = 0;
    forall_adj_nodes(w,v) {
    sum[j] = 1 + twos - Value2[w] + Value1[w];
    j++;
    }
    top = Value3[v];
    for (j = 0 ; j < G.outdeg(v) ; j++ )
    if (sum[j] > top)
    top = sum[j];
    Value3[v] = top;
    }
}

After the input, the tree nodes are sorted by their distance from the root: the data structure layer[MAX_DEPTH] is an array of lists, with layer[i] containing the nodes in the tree at depth i (the root being at depth zero). Then the entries of Value are computed following the algorithm in Section 2. At the end of each execution, if v is the unique node at level zero, then either Value1[v] or Value2[v] or Value3[v] contains v(G).

The code presented can be easily modified to return a maximum cardinality induced matching. Three linear lists of edges Match1[v], Match2[v], Match3[v] are associated with each v ∈ V(G). Moreover two pointers top1[v], bot1[v] (resp. top2[v], bot2[v] and top3[v], bot3[v]) to the first and last element in Match1[v], (resp. Match2[v] and Match3[v]) are defined. Insertion of a single edge in these lists is always performed at the top of the list and involves updating two pointers. The only other operation that is needed on these lists is the union, which, again, can be carried out in constant time. Initially all these lists are empty and all the pointers are set to NULL. At the end of each execution, for each v ∈ V(G), Match1[v] contains the edges of a maximum cardinality induced matching for the sub-
tree rooted at $v$, computed under the assumption that $v$ is a vertex of Type 1 (the definitions of $\text{Match}_2[v]$ and $\text{Match}_3[v]$ are similar). The content of $\text{Match}_1[v]$, $\text{Match}_2[v]$, and $\text{Match}_3[v]$ is filled when vertex $v$ is processed. If $v$ is of Type 1 (resp. Type 2) with respect to $\text{UN}(v)$ then $\text{Match}_1[v]$ (resp. $\text{Match}_2[v]$) is defined as the union of $\text{Match}_1[u]$ (resp. $\text{Match}_2[u]$) over all children $u$ of $v$. If $v$ is of Type 3, then $\text{Value}_3[v]$ is computed first and then $\text{Match}_3[v]$ is filled appropriately, possibly using one insertion and at most $c(v) - 1$ union operations.

4. Conclusions

In this paper we presented an algorithm which finds a largest induced matching in a tree. Our algorithm is simple in that it does not reduce the original problem to another one, its complexity improves the one of the best algorithm known and it is optimal in the sense that to define an induced matching in a tree $\Omega(n)$ operations are needed. We also presented a simple implementation of the algorithm above in C++ using the classes and data structures defined in LEDA.

References


