Algorithm to Find a Tree with Maximal Terminal Nodes

Ovidiu Domşa, Emilian Ceuca, Mircea Rasteiu

University „1 Decembrie 1918”, Alba Iulia, 2500
N. Iorga 13, Romania
Email: {ovidiu, eceuca, mrasteiu}@uab.ro

Abstract. Searching algorithms in data structures are the most common problems in the research and development of structures like string, heap or graph. We propose to solve a NP complete problem of searching in a large graph. The algorithm finds a substructure with certain properties in real time. We propose to find a tree with maximal terminal nodes using a heuristic algorithm in 2-approximation method.

1 Introduction

Graphs are the most common data structure and although the most complex structure. The algorithms to solve graph problems are well known but some of them don’t have a linear complexity. Class of the NP-complete algorithms has attracted a lot of scientific researches. We propose a heuristic algorithm in 2-approximation method to solve the next problem.

1.1 Problem

Let G be a connected undirected graph with n nodes and m arcs. Find spanning tree with maximum terminal nodes. Solve the problem for a maximum number of nodes n (n<=4000) and less than 10000 arcs. The algorithm will find the solution using a 2-approximation method in linear time.

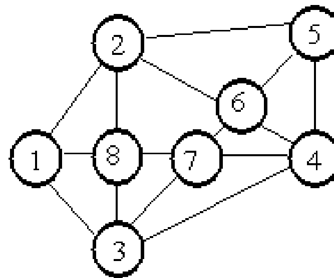

Fig. 1. The graph *G* have *8 nodes* and *15 arcs*

1.2 Example

In the graph G (Fig. 1.), with 8 nodes and 15 arcs, we find a spanning tree with five terminal nodes (Fig. 2.), but they are more.

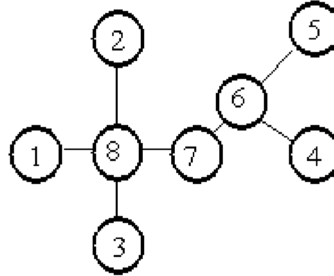

Fig. 2. The spanning tree is obtained by the graph keeping the nodes {1, 2, 3, 4, 5} like terminal nodes

2 Documentation

The problem has applications in the design of communication networks, circuit layouts and in distributed systems. Galbiati G., et. al [1] has proven that the problem is *MAX SNP* complete, and hence that there is no polynomial time approximation scheme for the problem unless *P=NP*. R. Solis-Oba in [3] presents a 2-approximation algorithm for the problem, improving on the previous best performance ratio of 3 achieved by algorithms of Ravi.

There are a lot of results in order to shown that connected graphs with n nodes have:

- for the minimum degree $k=3$ a spanning tree with minimum $n/4+2$ terminal nodes exists;
- for the minimum degree $k=4$, it is proof that the number of terminal nodes is at least equal to $(2n+8)/5$.

3 Algorithm

The main idea in the algorithm is a heuristic strategy based on the expansion rules. Every rule has its own priority and we will use a forest to construct the tree. The algorithm runs in linear time and uses only simple data structures.

Let $G= (N, A)$ be a connected undirected graph. Let m be the number of arcs and n be the number of nodes in G. Let T be a spanning tree of the graph G with maximum terminal nodes. Let F be a forest containing trees obtained by the expansion rules.

Rule 1 (connecting rule)

Let T_i and T_j be two trees in the forest F. Let T be a new tree obtained by connecting T_i with T_j. We say that x is a *killed terminal node*, if x is a terminal node in T_i and an internal node in T_j and becomes an internal node in T.

The expansion rules, which will be used to construct the forest F, are defined in the next section. The purpose of the rules is to construct the final spanning tree T, by connecting trees from the forest F. F is constructed so that each of its trees has a large number of terminal nodes. When T is formed we apply the rules to obtain a minimum number of *killed terminal nodes*, so that the number of *terminal nodes* in T is at the most.

Each tree T_i of the forest F is constructed by choosing as its root of the tree, a node with the degree at least 3. (If n is not so large it's useful to sort the nodes descending by degree) The expansion rules are applied:

Rule 2 (*expansion rule*)

If a terminal node x has at least two neighbors which are not in T_i, then all x neighbors which are not in T_i become its children.

Rule 3 (*expansion rule*)

If x has only one neighbor y, which is not in T_i and at least two neighbors of y are not in T_i, then y becomes the only child of x and all the neighbors of y, that are not in T_i, become y children. We say that x is expanded by the rule when we apply a rule to x.

The priorities of the rules are the following:

a. **Rule 3** has the priority one;

b. **Rule 1** and **2** has the priority two;

c. When a tree is constructed, if two different terminal nodes can be expanded the node that is expanded is the one which can be expanded with the highest priority rule.

d. If two terminal nodes can be expanded with the same priority rules then we pick one arbitrarily.

The tree T_i is expanded until neither expansion rule can be applied.

```
Algorithm tree(G)
F ← ϕ;
While there is a node v with degree at least 3 do
        Construct T_i with root v and terminal nodes all the neighbors of v
        while at least one terminal node in T_i can be expanded do
                find the terminal node in T_i which can be expanded by the rules with
                maximum priority and expand
        end while;
        F ← F ∪ T_i
```

Delete from G all nodes in T_i and all their incident arcs
end while;
Connect the trees from F and the nodes which are not in F to construct a spanning tree T.
End.

Fortuitous, the forest F constructed by our rules is a forest with terminal nodes, so we use its special structure to construct a spanning tree with a number of terminal nodes very close to the one of the forest.

Informal, expansion rules with low priority raise the number of terminal nodes with a low quantity, and the ones with higher priority raise the number of terminal nodes with a high quantity. We can demonstrate that the algorithm is a 2- approximate by showing that every rule with a low priority adds at least one node which has to be an internal node in any T*tree with a maximum number of terminal nodes. More, this set of internal nodes is disjunctive about the set of nodes necessary to interconnect sub trees of T* by the nodes covered by F. That is enough to proof that the maximum approximation factor for the algorithm is two.

4 Algorithm Analysis

Let $F = \{T_0, T_1, \ldots, T_k\}$ be the forest formed by the algorithm, and X the nodes that are not in F. We will name X a set of *exterior nodes*. It's easy to notice that an external node can not be adjacent to an internal node of a tree T_i. The expansion rules let us demonstrate the following properties of external nodes.

Lemma 1
Let $G' = (V', E')$ be the graph formed by shrinking of every tree $T_i \in F$ in one node and by eliminating multiple arcs between the pairs of nodes. In G, every exterior node's degree is maximum 2.

Proof
Let's say there is a node $v \in X$ in G' with the degree at least equal with 3. Let's take 3 from the neighbors of v. We notice that the 3 nodes can't be external node because then the tree algorithm would have chosen v as a root for the new tree. From here, at least one neighbor is in F. Let T_i be the first tree generated by the algorithm, which contains u, one of the neighbors of v. Because v is adjacent with two nodes which aren't in T_i, the tree algorithm would have expanded u.

Lemma 2
Let u be a terminal node of a tree $T_i \in F$. If u is adjacent with two nodes $v, w \notin T_i$, then v and w are terminal nodes of the same tree $T_j \in F$.

Proof
Obviously, *v* and *w* can't be both external nodes. Also, neither *v* and neither *w* can't be internal nodes of one T_j tree, because, for example, if *v* is an internal node of T_j , then:

1. if the algorithm constructs the T_j tree before T_i tree, then *u* would be placed as a child of *v* in T_j
2. if T_i is the first constructed, then *v* would have been placed as child of *u*. To proof this thing we need to notice that every internal node of a tree $T_j \in F$ has at least 3 neighbors in T_j. So, the node *u* would have been expanded when T_j was constructed.

From here, at least one of the nodes *u, w* must be a terminal node in *F*. Let *v* be a terminal node in a tree T_j . Let *p* be the parent of *u* in T_i. If *w* is not a terminal node in T_j , then we can suppose, without diminishing the generality that, when the tree algorithm adds the node *v* to the tree T_j , *v* is not in *F*. We notice that tree T_i can not be constructed before T_j , because the algorithm would have placed *v* and *w* as children of *u*. Let T_j be constructed before T_i . Then the nodes *u*, *p* and *w* are not in *F* by the time T_j is constructed. This means that the tree algorithm would have expanded *v* and would place *u* as a child of it in T_j . From here, *v* and *w* must be terminal nodes in T_j .

A Second Method is also Heuristic
Initially the nodes are marked with 0, for example.
The node with the highest degree is picked and all its neighbors are marked with 1, with the hope that some of them will remain terminal nodes.
One after another these nodes are covered, in reverse order by the degree, and it's applied the same process; the nodes that become internal nodes are marked with 2.
The algorithm ends when there are no more nodes marked with 0.

5 C++ Program for the Algorithm.

```
#include <stdlib.h>
#include <stdio.h>

#define NMAX 4000
#define MMAX 10000

int *l[NMAX+1], nl[NMAX+1], deg[NMAX+1], m[NMAX+1];
int lsa[NMAX], lsb[NMAX], nls;
int i,j,k,t,n,max,nf,pr, nm;

void citeste()
// read the graph G, with n nodes
// each line contains the list *l[i] of nodes neighbors
// preceding by their number t.
{
FILE* f= fopen("arbore.in","rt");
```

```
fscanf(f,"%d",&n);
for (i=1;i<=n;i++)
          {
          fscanf(f,"%d",&t);
          nl[i]= t;
          if ((l[i]= new int[t+1])==NULL)
          {puts("alloc err"); exit(1);} ;
          for (k=1; k<=t; k++) fscanf(f,"%d",&l[i][k]);
          }
          fclose(f);
}

void mark(int a, int b)
// mark in deg all the arcs incidents with (a, b)
{
          nls++; lsa[nls]= a; lsb[nls]= b;
          static int i;
          for (i=1; i<=nl[b]; i++)
           deg[l[b][i]]--;
          nf++; nm++;
          m[b]=1;
}
void rezolva()
// the algorithm "tree"
{
          for (i=1; i<=n;i++) deg[i]= nl[i];
          pr= 1;
          while (1)
          {
           max= 0;
           for (i=1; i<=n; i++)
            if ( (m[i]==1 || pr) && deg[i]>max)
            {
             max= deg[i];
             k= i;
            }
           if (pr) { mark(0,k); nls--; pr= 0; }
           if (!max) break;
           nf--;
           m[k]= 2;
           for (i=1; i<=nl[k]; i++)
            if (!m[l[k][i]]) mark(k,l[k][i]);
          }
          // condition for connected graph G
          if (nm<n) {puts("Neconex!");};
}

void scrie()
// print out (if the graph is connected) the tree by
```

```
// the numbers of arcs and remaining arcs
(lsa[i],lsb[i])
{
          FILE* f= fopen("arbore.out","wt");
          if (nm!=n) fputs("Neconex!\n",f);
          fprintf(f,"%d\n",nf);
          for (i=1; i<=nls; i++)
            fprintf(f,"%d %d\n",lsa[i],lsb[i]);
          fclose(f);
}

void main()
{
          citeste();
          rezolva();
          scrie();
}
```

6 Experiments

In the **Table 1**, we compare some tests. All the test runs in less than 1 second. We developed more than one solution. That we have other heuristic source, in Pascal and the results are comparable in terms of 2- approximations method.

Pascal Program

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N+,O-,P-,Q-,R-,S-,T-,V+,X+}
{$M 65000,0,655360}
type vec=array[1..10000] of integer; pvec=^vec;
var
   fil                              :text;
   N,m,i,j,k,bs,f,q,s               :integer;
   G,ba,bb,l,x,u,a,b                :array[1..2000] of in-
teger;
   V                                :array[1..2000] of
pvec;
Procedure AddEdge(i,j:integer);
begin
  inc(k);
  ba[k]:=i; bb[k]:=j;
end;
Procedure AddNode(q,t:integer);
var i,j:integer;
begin
  AddEdge(q,t);
  L[k+1]:=q;
  U[q]:=k+1;
```

```
    x[q]:=0;
    for i:=1 to G[q] do
    if U[v[q]^[i]]<>0 then dec(x[v[q]^[i]])
                      else inc(x[q]);
  end;
  Procedure G1;
  begin
     k:=0; j:=1;
     for i:=2 to N do if G[i]>G[j] then j:=i;
     l[1]:=j;
     u[j]:=1;
     x[j]:=g[j];
     for i:=1 to G[j] do addnode(v[j]^[i],j);
     bs:=g[j];
     while k<n-1 do
     begin
       j:=l[1];
       for i:=1 to k+1 do
        if x[l[i]]>x[j] then j:=l[i];
       bs:=bs+x[j]-1;
       for i:=1 to G[j] do
        if u[v[j]^[i]]=0 then Addnode(v[j]^[i],j);
     end;
  end;
  Procedure G2;
  begin
     k:=0; j:=1;
     for i:=2 to N do if G[i]>G[j] then j:=i;
     l[1]:=j;
     u[j]:=1;
     x[j]:=g[j];
     for i:=1 to G[j] do addnode(v[j]^[i],j);
     bs:=g[j];
     while k<n-1 do
     begin
       j:=l[1];
       for i:=1 to k+1 do
        if x[l[i]]>=x[j] then j:=l[i];
       bs:=bs+x[j]-1;
       for i:=1 to G[j] do
        if u[v[j]^[i]]=0 then Addnode(v[j]^[i],j);
     end;
  end;
  Procedure Checksol;
  begin
    if bs>s then
    begin
      move(ba,a,2*k);
      move(bb,b,2*k);
      s:=bs;
```

```
    end;
  end;

  begin
     assign(fil,'arbore.in');
     reset(fil);
     readln(Fil,N);
     for i:=1 to N do
     begin
       read(fil,G[i]);
       getmem(V[i],2*G[i]);
       for j:=1 to g[i] do read(fil,V[i]^[j]);
       readln(Fil);
     end;
     close(fil);
     G1;
     move(ba,a,k*2); move(bb,b,k*2);
     s:=bs;
     fillchar(u,2*n,0);
     G2;
     CheckSol;
     assign(fil,'ARBORE.OUT');
     rewrite(fil);
     writeln(fil,s);
     for i:=1 to n-1 do writeln(fil,a[i],' ',b[i]);
     close(Fil);
  end.
```

Nodes	Arcs	Terminal nodes C++	Terminal nodes Pascal
10	40	7	7
20	100	15	15
100	1000	85	85
1000	10000	846	850
2000	20000	1686	1697

Table 1. Comparison between number of nodes, number of arcs in the graph *G* and the number of terminal nodes found in the tree

References

1. Galbiati G., Morzenti A., Maffioli F.: On the Approximability of Some Maximum Spanning Tree Problems. *Theoretical Computer Science* 181(1):107-118 (1997)
2. Galbiati G., Morzenti A., Maffioli F.: A Short Note on the Approximability of the Maximum Leaves Spanning Tree Problem. *Information Processing Letters* 52(1):45-49 (1994)
3. Solis-Oba R.: 2-Approximation Algorithm for Finding a Spanning Tree with Maximum Number of Leaves. *Proceedings ESA Symposium*, Venice, Springer LNCS Vol.1461 (1998) .441-452