DilworthDecomposition User Manual

1

Version 1.0, by Sid-Ahmed-Ali Touati
In Memory of Vincent Bouchitté

Tue Aug 14 17:25:15 2007

Chapter 1

Dilworth Decomposition in C++ using the LEDA Graph library

Author:

: Sid-Ahmed-Ali Touati (in memory of Vincent Bouchitté)

1.1 Who was Vincent Bouchitté ?

Vincent Bouchitté was one of my former professors at Ecole Normale Supérieure de Lyon (France). He taught me deep and excellent fundamental results of graph theory, lattices and orders. Thanks to his courses, we were able to produce fundamental results on code optimisation published in the following article:

Sid-Ahmed-Ali Touati. Register Saturation in Instruction Level Parallelism. International Journal of Parallel Programming, Springer-Verlag, Volume 33, Issue 4, August 2005. 57 pages.

Vincent Bouchitté was member of the LIP laboratory at Ecole Normale Supérieure de Lyon. He died after a long disease. He was 47 years old. Very discrete, he was very appreciated by all his colleagues and students. He leaves major results in graph theory and advised beautiful Ph.D. theses. He was buried at Salindre in France, his birthplace, on March 15th, 2005.

In memory of him, I implemented Dilworth decomposition. Vincent Bouchitté taught us beautiful algorithms and formal proofs on the subject.

1.2 Quick Recall of Basic Notions and Notations on Graphs and Orders

An order is simply a directed acyclic graph (DAG). Given a DAG $G = (V, E)$, the following notations were used by Vincent Bouchitté and are usually used in lattices and orders algebra:

- $\Gamma^+_G(u) = \{v \in V | (u, v) \in E\}$ is the set of successors of u in the graph G ;
- $\Gamma^-_G(u) = \{v \in V | (v, u) \in E\}$ is the set of predecessors of u in the graph G ;
- $\forall e = (u, v) \in E, source(e) = u \wedge target(e) = v$. u, v are called *endpoints* ;

- $\forall u, v \in V : \ u < v \iff \exists$ a path $(u, \ldots, v)$ in G ;
- $\forall u, v \in V : \ u||v \iff \neg(u < v) \wedge \neg(v < u)$. u and v are said to be *parallel* ;
- $\forall u \in V \ \uparrow u = \{v \in V | v = u \vee v < u\}$ is the set of u 's ascendants including u . In other terms, a node u is an ascendant of a node v iff $u = v$ or if there exists a path from u to v ;
- $\forall u \in V \ \downarrow u = \{v \in V | v = u \vee u < v\}$ is the set of u 's descendants including u . In other terms, a node u is a descendant of a node v iff $u = v$ or if there exists a path from v to u ;
- Two edges e, e' are *adjacent* iff they share an endpoint;
- $A \subseteq V$ is an antichain iff all nodes belonging to A are parallel. Formally, $A \subseteq V$ is an antichain in G iff $\forall u, v \in A, \ u||v$;
- AM is a *maximal* antichain iff its size in terms of number of nodes is maximal. Formally, AM is a *maximal* antichain $\forall A$ antichain in $G, \quad |A| \leq |AM|$;
- The size of a maximal antichain is called the *width* of the DAG and is noted $w(G)$.
- $C \subseteq V$ is a chain iff all nodes belonging to C are not parallel. Simply, all nodes of a chain belongs to the same path in the DAG. Formally, $C \subseteq V$ is a chain in G iff $\forall u, v \in C, \ u < v \vee v < u$;
- $CD = \{C_1, \cdots, C_p\}$ is a chain partition of G if any $C_i \in CD$ is a chain and: $\forall u \in V, \exists! i \in [1, p] : u \in C_i$.
- A chain decomposition CD is minimal if its indice p is minimal. Such minimal indice is noted $p(G)$.
- In 1950, Dilworth proved that $p(G) = w(G)$, and each maximal antichain is equivalent to a minimal chain decomposition (and vice-versa).

1.3 Why do we choose LEDA Graph Library ?

LEDA is a famous C++ graphs and general data structures library. We have used it since many years, and we can safely say that it is *better* than other existing C++ graph and data structures libraries that we experimented (BOOST, STL, etc.). Initially, LEDA was an academic project from Germany. LEDA sources was distributed for free under a specific academic license untill version 4.2 (with g++2.95). Then, in 2001 (when g++ changed to version 3), LEDA team changed the free academic license into a low-priced academic license. LEDA is a high level library greatly helping to implement complex algorithms in a quick, robust and modular way. According to our deep experience, a C++ code using LEDA looks like a high level algorithm, allowing to easily debug it without suffering from programming details. Furthermore, LEDA offers the largest set of implementation of well known algorithms in graph theory and data structures.

1.4 Code Example for Usage

```
int main(int argc, char *argv[])
{
        graph G;
        LEDA::string filename;
        set<node> MA; //maximal antichain
        node_array<int> C; //indices of chains
        node_list nl;
        h_array<int,node_list*> chain(nil);
        int i,status;
        int size_ma, // size of a maximal antichain
                size_mc; // size of a minimal chain decomposition
        node u;
        if(argc!=2){
                cerr << argv[0] << ": Dilworth decomposition." << endl;
                cerr << "Usage:"<<argv[0] << " graph_filename" << endl;
                cerr << "The filename extension should be .gw for a graph in leda/gw format, or in .gml for a graph in G
                return EXIT_FAILURE;
        }

        filename=LEDA::string(argv[1]);
        if (filename.contains(LEDA::string(".gw"))) {
                cout<<"Reading GW" <<filename<<endl;
                status=G.read(filename);
        }
        else if (filename.contains(LEDA::string(".gml"))) {
                cout<<"Reading GML "<< filename<<endl;
                status=G.read_gml(filename);
        }
        else {
                cerr << "Usage:"<<argv[0] << " graph_filename" << endl;
                cerr << "The filename extension should be .gw for a graph in leda/gw format, or in .gml for a graph in G
                return EXIT_FAILURE;
        }

        switch(status){
        case 0:
        case 2: break;
        case 1: cerr<< filename << " does not exist."<<endl; break;
        case 3: cerr<<filename <<" does not contain a graph"<<endl; break;
        default: return EXIT_FAILURE;
        }

        size_mc=MINIMAL_CHAIN(G, C);
        cout<<"Minimal Chain Decomposition"<<endl;
        cout<<"---------------------------"<<endl;
        cout<<"There are "<<size_mc<<" chains"<<endl;
        forall_nodes(u,G){
                if ((chain[C[u]])==nil){
                        chain[C[u]]=new node_list;
                }
                (chain[C[u]])->append(u);
        }
        for(i=0;i<size_mc;i++){
                cout<<"chain "<<i<<": ";
                forall(u, *chain[i]){
                        G.print_node(u);
                }
                cout<<endl;
        }


        size_ma=MAXIMAL_ANTI_CHAIN(G, MA);
        cout<<"Maximal Antichain"<<endl;
        cout<<"---------------------------"<<endl;
        cout<<"Size of this maximal anctichain : "<<size_ma<<" nodes"<<endl;
```

```
        cout<<"Here are all these nodes:"<<endl;
        i=0;
        forall(u, MA){
                cout<<"node "<<i<<": ";
                G.print_node(u);
                cout<<endl;
                i++;
        }
        return EXIT_SUCCESS;
}
```

1.5 Download the Sources

This implementation has been done by Sid-Ahmed-Ali Touati and distributed under GPL. The first version is available `here` .

Chapter 2

DilworthDecomposition File Documentation

2.1 dilworthdecomposition.cpp File Reference

2.1.1 Detailed Description

This file contains the C++ implémentation of Dilworth decomposition using the LEDA graph library.

Functions

- template<class T> set< T > **list_to_set** (const list< T > l)
  This function returns the set of members in a list. I.e, it converts an ordered list to a set.
- int **MAXIMAL_ANTI_CHAIN** (const graph &G, set< node > &MA)
  This function computes a maximal antichain chain of a DAG (order).
- int **MINIMAL_CHAIN** (const graph &G, node_array< int > &chain)
  This function computes a minimal chain decomposition of a DAG (an order).
- template<class T> list< T > **set_to_list** (const set< T > l)
  This function returns the an ordered from a set.

2.1.2 Function Documentation

2.1.2.1 template<class T> template< class T > set< T > list_to_set (const list< T > *l*)

This function returns the set of members in a list. I.e, it converts an ordered list to a set.

Parameters:

← ***l*** The ordered list to convert.

Returns:

the set of the elements of the ordered list.

2.1.2.2 template<class T> template< class T > list< T > set_to_list (const set< T > *l*)

This function returns the an ordered from a set.

Parameters:

← ***l*** The set to convert.

Returns:

the ordered list of the input set members.

2.1.2.3 int MAXIMAL_ANTI_CHAIN (const graph & *G*, set< node > & *MA*)

This function computes a maximal antichain chain of a DAG (order).

Parameters:

← ***G*** The DAG.
→ ***MA*** A Maximal antichain.

Returns:

$w(G)$ the width of the DAG. It is equal to the size of a maximal antichain.

Remarks:

A maximal antichain may not be unique. This function returns an arbitrary one.

Precondition:

G is acyclic.

2.1.2.4 int MINIMAL_CHAIN (const graph & *G*, node_array< int > & *chain*)

This function computes a minimal chain decomposition of a DAG (an order).

Parameters:

← ***G*** The DAG.
→ ***chain*** $\forall u \in V, chain[u] \in [0, p(G)-1]$ contains the number of the chain to which u belongs.

Returns:

$p(G)$ the minimal number of chains of the DAG. Dilworth proved that $p(G) = w(G)$, that is, it is equal to the size of a maximal antichain.

Remarks:

A minimal chain decomposition may not be unique. This function returns an arbitrary one.

Precondition:

G is acyclic.

2.2 dilworthdecomposition.h File Reference

2.2.1 Detailed Description

This is the C++ header file of Dilworth decomposition implementation. To be included for use.

Namespaces

- namespace **LEDA**
- namespace **std**

Functions

- template<class T> set< T > **list_to_set** (const list< T > l)
  This function returns the set of members in a list. I.e, it converts an ordered list to a set.
- int **MAXIMAL_ANTI_CHAIN** (const graph &G, set< node > &MA)
  This function computes a maximal antichain chain of a DAG (order).
- int **MINIMAL_CHAIN** (const graph &G, node_array< int > &chain)
  This function computes a minimal chain decomposition of a DAG (an order).
- template<class T> list< T > **set_to_list** (const set< T > l)
  This function returns the an ordered from a set.

2.2.2 Function Documentation

2.2.2.1 template<class T> list<T> set_to_list (const set< T > *l*)

This function returns the an ordered from a set.

Parameters:

← ***l*** The set to convert.

Returns:

the ordered list of the input set members.

2.2.2.2 template<class T> set<T> list_to_set (const list< T > *l*)

This function returns the set of members in a list. I.e, it converts an ordered list to a set.

Parameters:

← ***l*** The ordered list to convert.

Returns:

the set of the elements of the ordered list.

2.2.2.3 int MAXIMAL_ANTI_CHAIN (const graph & *G*, set< node > & *MA*)

This function computes a maximal antichain chain of a DAG (order).

Parameters:

← ***G*** The DAG.

→ ***MA*** A Maximal antichain.

Returns:

$w(G)$ the width of the DAG. It is equal to the size of a maximal antichain.

Remarks:

A maximal antichain may not be unique. This function returns an arbitrary one.

Precondition:

G is acyclic.

2.2.2.4 int MINIMAL_CHAIN (const graph & *G*, node_array< int > & *chain*)

This function computes a minimal chain decomposition of a DAG (an order).

Parameters:

← ***G*** The DAG.

→ ***chain*** $\forall u \in V, chain[u] \in [0, p(G)-1]$ contains the number of the chain to which u belongs.

Returns:

$p(G)$ the minimal number of chains of the DAG. Dilworth proved that $p(G) = w(G)$, that is, it is equal to the size of a maximal antichain.

Remarks:

A minimal chain decomposition may not be unique. This function returns an arbitrary one.

Precondition:

G is acyclic.