A Synthesis on Partition Refinement: a useful Routine for Strings, Graphs, Boolean Matrices and Automata

Michel Habib* Christophe Paul* Laurent Viennot†

September 12, 1997

Abstract

Partition refinement techniques are used in many algorithms. This tool allows efficient computation of equivalence relations and is somehow dual to union-find algorithms. The goal of this paper is to propose a single routine to quickly implement all these already known algorithms and to solve a large class of potentially new problems. Our framework yields to a unique scheme for correctness proofs and complexity analysis. Various examples are presented to show the different ways of using this routine.

1 Introduction

A *partition* of a finite set E is a collection of disjoint subsets of E called *classes* whose union is E. *Refining* a partition consists in splitting its classes into smaller classes. Partition refinement techniques have been studied in four main papers [7, 15, 13, 6]. Hopcroft [7] may be the very first designer of such a technique. He used it in order to minimize the number of states of a deterministic finite automaton. Spinrad [15] investigated the graph partitioning field with application in substitution decomposition and transitive orientation. Paige and Tarjan [13] used partition refinement techniques in three different applications: strings lexicographic sort, doubly lexicographic ordering of a boolean matrix and relational coarsest partition (the authors of [13] point out that this last problem is very close to deterministic finite automaton minimization). Habib, McConnell, Paul and Viennot [6] proposed new efficient algorithms based on partition refinement for the recognition of various classes of graphs and boolean matrices that have the consecutive one's property.

It turns out that partition refinement is used in many different area of computer science dealing with graphs, strings, boolean matrices or automata. Indeed, many articles rely on partition refinement subroutines even if they do not develop them. The goal of this paper is to propose a single routine to quickly implement all these algorithms and to solve a large class of problems. A sample of examples is presented to show different ways of using this routine: computing twins of a graph, lexicographic string sorting, consecutive ones test of boolean

*LIRMM, Montpellier, France; email : (habib,paul)@lirmm.fr
†LIAFA, Paris, France;email: lavie@litp.ibp.fr

matrix and minimization of a deterministic automata. Just small basic procedures have to be adapted for each applications. This compilation allows similar correctness proofs and a general scheme for complexity issues.

Section 2 presents the partition refinement paradigm and the main routine in details. A classification of the different applications is proposed in Section 3. The last sections develop involved examples that make a powerful use of the partition refinement technique: automata minimization and consecutive ones test. Detailed proofs are only given for Hopcroft's Algorithm for automaton minimization to give an example and to show this proof that has the reputation of being difficult becomes quite simple. The reader interested in detail proofs of the other algorithms is invited to consult the references given in this paper.

2 Partition Refinement

All the algorithms we are going to propose are based on the following routine that iteratively refines a partition of a set E according to a subset S of E called *pivot set*: each class $\mathcal{X}$ is replaced by $\mathcal{X} \cap S, \mathcal{X} - S$. Partitions will be implemented by sorted lists. Therefore our partitions are implicitly ordered. A partition Q is *compatible* with a partition P if every class of Q is included in a class of P and if the ordering in P respects the ordering in Q (i.e. if in P the class $\mathcal{X}$ is before the class $\mathcal{Y}$ then any class $\mathcal{X}' \subseteq \mathcal{X}$ of Q is before any class $\mathcal{Y}' \subseteq \mathcal{Y}$). Refining a partition produces a compatible partition. We say by extension that an ordering $x_1, \ldots, x_n$ of the elements of E is *compatible* with a partition L if the partition $(\{x_1\}, \ldots, \{x_n\})$ is compatible with L. Given a subset $S \subseteq E$, a partition L of E is said to be S*-stable* when no class properly overlap S. After the refinement step of L by S, L is S-stable: each class $\mathcal{X} \in L$ verifies either $\mathcal{X} \subseteq S$ or $\mathcal{X} \cap S = \emptyset$.

Algorithm 1 shows an implementation of the partition refinement routine that allows to simulate many algorithms. Depending on the use of the routine, four basic procedures, *PivotSet*, *LaunchPartition*, *InsertRight* and *AddPivot* have to be implemented. *PivotSet* should compute a pivot set (i.e. a subset of E) from some small information called *pivot*. In simple applications, the pivot is a pointer on some subset of E given in the input data and *PivotSet* returns this set. In others, the pivot set must be computed when needed. A stack *pivots* stores the pivots that may be used to refine the partition. There are two ways of adding pivots to the stack: with the procedure *AddPivot*, whenever a class is splited, and with the procedure *LaunchPartition*, whenever the stack is empty. The management of the pivots is critical for the complexity issues. The last procedure concerns the order of the classes in the partition. Whenever a class $\mathcal{X}$ is splited by a pivot set S, $\mathcal{X}$ is replaced either by $\mathcal{X} \setminus S, \mathcal{X} \cap S$ or $\mathcal{X} \cap S, \mathcal{X} \setminus S$ in L according to the result of the procedure *InsertRight*.

Partition Refinement Correctness Proof

All the algorithms of this paper can be proved with this invariant of the inner while loop of Algorithm 1. Properties A and B are defined for each application:

- ***A:*** *some property implying the existence of a solution compatible with the partition*
- ***B:*** *some property obtained when the partition is enough refined*

Invariant: *The partition L always verifies A and if L does not verify B then some pivot in* pivots *will strictly refine L.*

Algorithm 1: Procedure *PartitionRefinement*(L)

Input: a partition L of a set E in classes; L is ordered from left to right
Output: a refined partition $L = (\mathcal{X}_1, \ldots, \mathcal{X}_h)$ of E

```
begin
  pivots = ∅ is an empty stack of pivots (each pivot is associated to a
  subset of E via the procedure PivotSet)
  while the LaunchPartition procedure does not break the loop do
    LaunchPartition(L)
    while pivots ≠ ∅ do
      pick a pivot p in pivots
      S = PivotSet(P)
      refinement step by S
        let M be the set of classes properly overlapping S (i.e. inter-
        secting S and not included in S)
        let N be the set of classes included in S
        for each class X ∈ M do
          let Y be the members of X that are in S
          remove Y from X
          if InsertRight(X, Y, p, M, N) then
              insert Y immediately on the right of X
          else insert Y immediately on the left of X
          AddPivot(X, Y)
end
```

To prove the correctness of all presented algorithms, we just have to prove that any refinement step maintains Property A and that enough pivots are added so that B is verified when there are no more pivots. After the inner while loop A and B are verified. The correctness of the procedure *LaunchPartition* is a key point of the proofs.

Complexity Issues

The refinement step can be performed in $O(|S|)$ using the following data structure. All the elements in E are stored in a doubly linked list. Each class consists in an interval of this list and is made of two pointers to its first element and to its last element. All the classes are stored in a doubly linked list L. The integers bounds of the intervals can also be maintained to allow constant computation of the cardinality of the classes and their relative positions in L. Each element keeps a pointer to its class. This data structure is illustrated by Figure 1.

During the refinement step, each element in the pivot set S is simply removed from the list and inserted at the end of its new class. This allows to preserve the initial ordering of the vertices inside the classes when S is sorted according to this ordering. Notice that the classes are in fact totally ordered subsets of E. The set M of classes intersecting S is computed while scanning S. Afterwards M is traversed and the empty classes are un-splited, removed from M, and added to N (to unsplit a class, simply give it the bounds values of the associated new class which is deleted from L).

A bound on the overall execution time of the *PartitionRefinement* proce-

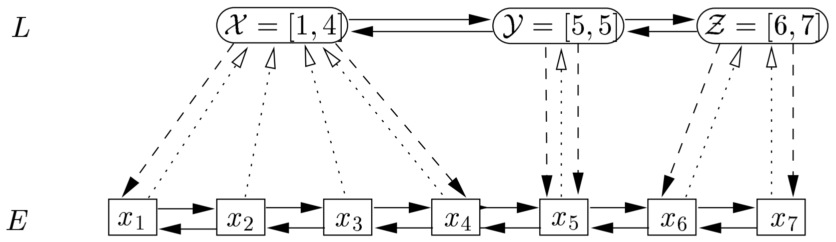

Figure 1: Implementation of an ordered partition

dure is thus easily obtained from a bound on the sum of the sizes of the pivot sets and on the overall time spent in the procedures *PivotSet*, *LaunchPartition*, *InsertRight* and *AddPivot*. *InsertRight* will always consist in a test computable in constant time.

Bounding the complexity in time and space of the partition refinement routine resides in bounding the number of pivots added in the stack. All the pivot sets are either distinct parts of the input data or are computed from the elements of one of the two subclasses newly created by splitting a class. In the first case, pivots are known in advance and each one is used once yielding a linear time and space algorithm. In the second case, the Hopcroft "process the smaller half" strategy allows to get an $O(m \log n)$ time (where m is the size of the input) algorithm. Breaking the pivot set computation in two steps with *AddPivot* and *PivotSet* allows to bound the stack size to get a linear space algorithm. This discussion inspires a classification of the partition refinement applications presented in the paragraph "Pivot Rule" of the next section.

3 Classifications of Partition Refinement Applications

There are mainly three criteria to classify the partition refinement applications. The first one concerns the type of problem that the application solves and will be used as a main classification along the paper. The second classification concerns the way the pivots are chosen and the third one concerns the use of the procedure *LaunchPartition*.

Ordered and Unordered Partition Refinement

There are mainly two classes of problems that can be solved with partition refinement depending on the nature of the solution: an unordered partition (the classes of some equivalence relation have to be computed) or an ordering of the elements (sorting for example). In the first case, the order of the partition is not important with regard to the problem. In the second case the ordering is obtained by computing an ordered partition where all classes are singletons.

Unordered Partition Refinement: Unordered refinement partition algorithms allows to compute congruence relations when the information "x, y does not belong to the same class" is easy to compute. In our framework, a pivot separates x and y. Dually, when the information "x, y belongs to the same class" is easier to compute, a natural paradigm is to use union-find techniques [16].

Algorithm 2: Computing twins

Input: the adjacency lists of a graph with vertex set V

Output: a partition $L = (\mathcal{X}_1, \ldots, \mathcal{X}_h)$ of the vertices in classes of twins

begin

let L be the one element list (V)

run *PartitionRefinement*(L) with the following definitions (the pivots are vertices)

end

Procedure *PivotSet*(p)

return the adjacency list of the vertex p

Procedure *LaunchPartition*(L)

add all the vertices in V to *pivots* at the first call to *LaunchPartition* and exit and return L at the second call

Procedure *InsertRight*$(\mathcal{X}, \mathcal{Y}, p, M, N)$

return true

Procedure *AddPivot*$(\mathcal{X}, \mathcal{Y})$

do nothing

Unordered partition refinement is used in twins computation, automaton minimization [7, 2], modular decomposition [15, 11, 5] and coarsest partition computation [13]. All these problems can be described as the computation of congruence classes of some congruence (or equivalence relation). This relation is the Nerode equivalence in automaton minimization; the coarsest partition problem is very similar and a similar relation can be defined. Two vertices of a graph are twins if they have same neighborhood, "being twins" is clearly an equivalence relation and its computation is given as an example of unordered partition refinement in Algorithm 2.

- ***A:*** *if the vertices u and v are twins, then they are in the same class*
- ***B:*** *L is stable for every neighborhood $N(v)$ of a vertex v*

Invariant: *The partition L always verifies A and if L does not verify B then some pivot in* pivots *will strictly refine L.*

The correctness of Algorithm 2 is due to the above invariant. If two vertices are twins (they have the same neighbors), then no vertex can separate then. That proves property A. If property B is not true then there exists a vertex w that splits some class $\mathcal{X}$ of L that must still be in the stack *pivots*.

Algorithm 2 is linear since each vertex is used once as pivot and the refinement step by a pivot v takes $O(|N(v)|)$ time where $N(v)$ is the adjacency list of v.

Ordered Partition Refinement: In ordered partition refinement algorithms, the partition is generally refined until all classes are singletons yielding a total ordering of the elements. Partition refinement can be seen here as a sorting algorithm where at any moment, the elements of a class are considered equal with respect to computation done up to that point.

Ordered partition refinement may also compute a congruence relation in addition. This is the case when the final classes are not necessarily singletons. The final classes are then the classes of elements that can be ordered independently and the final ordering associated to the partition is a solution.

Algorithm 3: Lexicographic string sorting

Input: n strings $x_1, \ldots, x_n$

Output: a partition $L = (\mathcal{X}_1, \ldots, \mathcal{X}_h)$ of the strings

begin

  let L be the one element list $(\{x_1, \ldots, x_n\})$
  run *PartitionRefinement*(L) with the following definitions (the pivots are sets of strings)

end

Procedure *PivotSet*(p)

  return p

Procedure *LaunchPartition*(L)

  if this is the second call to *LaunchPartition* then exit and return L
  precompute the non empty sets $S_{i,a}$ of string having a fixed letter a at a fixed position i (create a couple $(-i, a)$ for each letter of any string and radix sorting them)
  add all these sets to *pivots* so that sets corresponding to largest ordered pairs $(-i, a)$ will be picked first

Procedure *InsertRight*$(\mathcal{X}, \mathcal{Y}, p, M, N)$

  return true

Procedure *AddPivot*$(\mathcal{X}, \mathcal{Y})$

  do nothing

Ordered partition refinement is used in transitive orientation [11], Lex-BFS (lexicographic breadth first search) [14], consecutive ones property testing [8, 6], string sorting [13] and sorting in general. We give a simple algorithm for lexicographic string sorting (see Algorithm 3) as example. Here is its invariant:

- ***A:*** *a lexicographic sort of the string is compatible with L*
- ***B:*** *two different strings cannot appear in the same class*

Invariant: *The partition L always verifies A and if L does not verify B then some pivot in* pivots *will strictly refine L.*

This algorithm does not terminate with singletons but with classes of equal strings. The radix sort procedures takes $O(n + k + m)$ time where n is the number of strings, k is the size of the alphabet and m is the sum of the string sizes. The sum of the pivot set cardinalities is m and the partition refinement procedure thus takes $O(n+m)$ time. It is shown in [13] how to get a $O(n+k+m')$ algorithm by carefully computing the pivot sets during the partition refinement process where m' is the sum of the distinguishing prefix sizes ($m = m'$ in the worst case). Similarly a simulation of quick-sort with partition refinement is left to the reader.

Pivot Rule

As we have seen before, the partition refinement applications can use different strategy to choose the pivots (i.e. to add them in time in the stack *pivots*).

There are two basic cases: the pivots are known in advance or they are computed during the algorithm depending on the current partition. The first case is more simple and leads generally to a linear time algorithm as we mentioned in the previous section. Twins computation, lexicographic string sorting, Lex-BFS are of this type.

The second case corresponds to more involved problems like automaton minimization, coarsest partition computation, consecutive ones test, modular decomposition and transitive orientation. For these problems, the Hopcroft's "process the smaller half" strategy allows to get very simple $O(m \log n)$ algorithms (m is the size of the data and n the number of elements). It is surprising to see that all these problems except automaton minimization and coarsest partition computation can be solved in linear time by finding a clever pivot choice [8, 6, 12] (Finding the pivots may be extremely complex, such as in linear transitive orientation for example [12]). This suggests that it should be possible to minimize a deterministic finite automaton in linear time.

The order in which the pivots are picked from the stack can be important or not. Sometimes only one pivot must be chosen among a set of possible ones. A rule has to be included in the procedure *LaunchPartition* to break this tie. This is the subject of the next classification.

Tie-break Rule

Some further differences come from the existence of a tie-break rule. In some algorithms the process is launched just once. It means that when the stack of pivot is empty and the algorithm ends (the inner while loop of Algorithm 1 is executed only once). This is the case in twins computation, lexicographic string sorting, automaton minimization and modular decomposition. The other algorithms have to be launched again until the resulting partition is a set of singletons. In that case, a tie-break rule has to be designed. This is the case for Lex-BFS, transitive orientation and the consecutive ones test. The way of breaking the tie is often a key part of the algorithm. Based on Lex-BFS, some interval graph recognition algorithms have been proposed by computing several successive Lex-BFS with more elaborate tie-break rules [4].

Considerations about the Representation of the Input

An interesting property of partition refinement is that refining a partition by a subset S or its complement $E - S$ is equivalent (the procedure *InsertRight* may have to be tuned slightly differently from one case to the other). This allows some degrees of freedom with the input data structure.

In graph algorithms, E is often the vertex set and S is the neighborhood of a vertex p. In that case, it is possible to work on a graph using its complementary as a data structure that represents it. In other words, we can run the partition refining routine on the complement of a graph without computing it, using only the edges of the graph itself. This nice property was used in [11] to recognize permutation graphs which are the comparability graphs such that the complementary graph is also a comparability graph and in [6] to recognize co-chordal and co-interval graphs. This idea is further developed in [5] where it is proposed to represent a graph by giving for each vertex either the list of its neighbors or the list of its non neighbors. Partition refinement allows to compute with such a representation as easily as with a classical one.

Algorithm 4 is an adaptation of an existing partition refinement algorithm (namely Lex-BFS) when its input is given in an other form.

Lex-BFS [14] is a partition refinement algorithm that computes a special ordering of the vertices of a graph called a Lex-BFS ordering. When the input

Algorithm 4: Clique Lex-BFS

Input: a graph $G = (V, \mathcal{E})$ given by its maximal cliques $C_1, \ldots, C_k$

Output: a lex-BFS ordering $L = C'_1, \ldots, C'_k$ of the cliques

begin

let L be the one element list $(\{C_1, \ldots, C_k\})$

precompute the set of cliques containing each vertex; and set $i = |V|$

run *PartitionRefinement*(L) with the following definitions (the pivots are vertices and are all set as "unused" at the beginning)

end

Procedure *PivotSet*(p)

number p by i and set i to $i - 1$

return the list of cliques containing the vertex p

Procedure *LaunchPartition*(L)

if *there is only singleton classes in* L **then** exit and return L

let C be a clique in the rightmost class $\mathcal{X}$ with cliques containing unnumbered vertices

if $\mathcal{X}$ is not a singleton then replace $\mathcal{X}$ by $\mathcal{X} \setminus \{C\}, \{C\}$ in L

add all the unused vertices in C to *pivots* and set them as "used"

Procedure *InsertRight*$(\mathcal{X}, \mathcal{Y}, p, M, N)$

return true

Procedure *AddPivot*$(\mathcal{X}, \mathcal{Y})$

do nothing

is chordal, this ordering allows to compute in linear time the maximal cliques of the input (a clique is a set of vertices inducing a complete graph). Moreover, this ordering induces an ordering of the maximal cliques called Clique Lex-BFS ordering. Algorithm 4 allows to compute a Clique Lex-BFS when the maximal cliques are directly given as input. This algorithm is useful l for the consecutive ones test algorithm presented in [6]. Note that we cannot compute a classical representation of the graph in order to run the classical Lex-BFS on it for complexity reasons since its size may be significantly greater.

4 An Unordered Partition Refinement Problem

Hopcroft's Algorithm for Deterministic Finite Automata Minimization Revisited

We now introduce a partition refinement version of Hopcroft's Algorithm [7]. The hard part of automata minimization is to compute the classes of the Nerode equivalence. Two states q and q' of an automata are Nerode equivalent if and only if $q.w = q'w$ for every word w ($q.w$ denotes the state reached by reading w when the automata is in state q).

Algorithm 5 shows how to simulate the Hopcroft algorithm for computing these classes with a call to our partition refinement procedure. A partition of the states set of the automata is refined according to pivot sets of the form $a^{-1}\mathcal{X}$ where a is a letter, $\mathcal{X}$ is a class of the current partition and $a^{-1}S$ denotes the set of the states q such that $q.a \in S$ (for any letter a and any state subset S).

Algorithm 5: Deterministic Finite Automata Minimization

Input: a complete accessible deterministic automata (Q, i, T)

Output: a partition $L = (\mathcal{X}_1, \ldots, \mathcal{X}_h)$ of the vertices in Nerode classes

begin

let L be the one element list (Q)

precompute for each state q and each letter a the set $a^{-1}q$ of the states p such that there exists a transition $q = p.a$

run *PartitionRefinement*(L) with the following definitions (the pivots are couples (class, letter))

end

Procedure *PivotSet*(p)

p is the couple $(\mathcal{X}, a)$

compute $a^{-1}\mathcal{X}$ by merging the sets $a^{-1}q$ for $q \in \mathcal{X}$ and return this set

Procedure *LaunchPartition*(L)

if this is the second call to *LaunchPartition*, exit and return L

remove T from Q and insert it as a new class in L

AddPivot$(Q - T, T)$

Procedure *InsertRight*$(\mathcal{X}, \mathcal{Y}, p, M, N)$

return true

Procedure *AddPivot*$(\mathcal{X}, \mathcal{Y})$

let $\mathcal{Z}$ be the smallest class between $\mathcal{X}$ and $\mathcal{Y}$

for each $a \in A$ **do**

if $\mathcal{X}, a$ *is already in* pivots*(i.e* $\mathcal{X}, a$ *has been added to* pivots *before* $\mathcal{Y}$ *was removed from it)* **then**

add $(\mathcal{Y}, a)$ to *pivots*

else add $(\mathcal{Z}, a)$ to *pivots*

Algorithm 5 runs in time $O(nk \log n)$ where n is the number of states and k the number of letters since each set $a^{-1}q$ is traversed at most $\log n$ times.

The correctness of Algorithm 5 comes from the fact that the partition L always verifies the following invariant:

- ***A:*** *if two states p and q are Nerode equivalent then they are in the same class*
- ***B:*** *L is stable for $a^{-1}\mathcal{X}$ for every class $\mathcal{X} \in L$ and every letter $a \in A$*

Invariant: *The partition L always verifies A and if L does not verify B then some pivot in* pivots *will strictly refine L.*

The proof of Hopcroft's Algorithm has the reputation of being quite intricate. We now give the complete proof of Algorithm 5 to show how our formalism makes it simple.

Proof: We say that a word w *separates* two states q and q' when $q.w \in T$ and $q'.w \notin T$ or when $q.w \notin T$ and $q'w \in T$. Two states are Nerode equivalent when no word separates them. The empty word is denoted ε.

Let us first show the conservation of Property A. Non terminal and terminal states are not Nerode Equivalent since ε separates them. A is thus verified after the first call to *LaunchPartition*. Suppose that A is true before an iteration of the inner while loop, we show that it is still after the refinement step by a pivot set $a^{-1}\mathcal{X}$ where a is some letter and $\mathcal{X}$ some class of the partition. Suppose by contradiction that two Nerode equivalent states q and q' are splited apart in two different classes. This means that $q.a$ and $q'.a$ appear in different classes. This contradicts the induction hypothesis.

We now show that if the property B is false then some pivot in *pivots* will strictly refine L. Suppose that L is not stable for $a^{-1}\mathcal{Z}$ for some letter a and some class $\mathcal{Z}$. There must exist two states q and q' in the same class such that $q.a \in \mathcal{Z}$ and $q'.a \notin \mathcal{Z}$. Consider the first time $q.a$ and $q'.a$ have been splited apart in two different classes. The smallest one contained either $q.a$ or $q'.a$ and has been added to *pivots* with the letter a. This implies that either the class of $q.a$ or the class of $q'.a$ appears in *pivots* with the letter a. This ordered pair will produce a pivot set containing either q or q' and pivoting on it will strictly refine the partition. Notice that whenever a class $\mathcal{Z}$ is splited in two classes $\mathcal{X}$ and $\mathcal{Y}$, if $(\mathcal{Z}, a)$ was in *pivots* then $(\mathcal{X}, a)$ and $(\mathcal{Y}, a)$ are in *pivots* after the call to *AddPivot*.

Let us finally prove that the conservation of the invariant implies that the final partition is made of the Nerode classes. We just have to prove that non Nerode equivalent states cannot be in the same class of the final partition. Suppose by contradiction that two states q and q' are in the same class but there exists a word w separating them. Let u be the longest prefix of w such that $q.u$ and $q'.u$ are in the same class of the final partition. We have $u \neq w$ since the final partition is a refinement of $(T, Q - T)$ implying that $q.w$ and $q'.w$ cannot be in the same class. Let a be the letter following u in w. Let $\mathcal{X}$ be the class of $q.ua$. The property B is then false since the partition is not stable for $a^{-1}\mathcal{X}$. This is a contradiction since *pivots* is empty at the end of the algorithm. □

5 An Ordered Partition Refinement Problem

Consecutive Ones Property Testing

A boolean matrix has the consecutive ones property if its columns can be permuted such that in each row the one entries occur consecutively. Such a permutation will be called a *consecutive permutation*. The problem consists in testing wether a given boolean matrix has the consecutive ones property and to compute a consecutive permutation if there exists one.

The first linear time algorithm for this problem [3] used the PQ-trees, a complicated data structure. A simpler algorithm was also presented in [8]. In [6] a new linear algorithm avoiding PQ-trees is proposed. A consecutive ones test can be used for interval graph recognition but it is a more general problem. A graph G is an interval graph iff its maximal cliques can be ordered such that the maximal cliques containing a given vertex appears consecutively. Thus if we represent an interval graph by its incidence vertex-clique matrix M, such an ordering of the maximal clique is exactly a consecutive permutation of the columns of M. Since Lex-BFS can be adapted for a clique representation of the graph (see algorithm 4), this correspondence allows the same adaptation for a vertex-clique matrix representation.

Algorithm 6 shows how to solve the consecutive ones problem thanks to a call to the partition refinement routine. Here, a partition of the columns is refined by pivoting on the rows where the pivot set associated to a row p is the set of columns containing a one entry at row p. The structure of the algorithm is similar to the algorithm proposed in [6] but here, the strategy for choosing the pivots is inspired from the Hopcroft Algorithm rule and is much more simple. The result is an extremely simple $O(n + n' + m \log n)$ algorithm for testing the consecutive ones property (m is the number of one entries, n is the number of columns, and n' the number of rows).

The input is given by the coordinates of the m one entries of the matrix (the other entries are zeros). An efficient algorithm for consecutive ones test must have a tight bound on m rather than nn'. All zeros columns and rows can easily be treated separately. We say that a column C contains a row r if the corresponding entry of the matrix M is one ($M(r, C) = 1$). This allows to consider the columns as sets of rows. We can also associate to each row r the set $C(r)$ of columns containing them. These sets are easily computed in linear time by radix sorting the coordinates of the one entries. Those which are not empty are given as input to Algorithm 6. The input matrix verifies the consecutive ones property if the computed permutation of the column is consecutive (this can be tested in linear time by scanning each set $C(r)$).

Algorithm 6: Consecutive ones testing

Input: a boolean matrix M with no all zeros column and no all zero row

Output: a consecutive ones permutation of the columns of M if there exists one

begin

  compute a Lex-BFS ordering $C_1 < \cdots < C_k$ of the columns of M
  let L be the one element list $(\{C_1, \ldots, C_k\})$
  bucket sort all the sets $C(r)$ of columns containing a given row r according to the Lex-BFS ordering of the columns
  run *PartitionRefinement*(L) with the following definitions (the pivots are rows and are all set as "unused" at the beginning)

end

Procedure *PivotSet*(p)

  if *all the columns in* $C(p)$ *are in the same class of* L **then**
    set p as "unused" and return $\emptyset$
  else return $C(p)$

Procedure *LaunchPartition*(L)

  if *there is only singleton classes in* L **then** exit and return L
  let $\mathcal{X}$ be a non singleton class in L and let C be the smallest column in $\mathcal{X}$ according to the Lex-BFS ordering of the columns
  replace $\mathcal{X}$ by $\mathcal{X} \setminus \{C\}, \{C\}$ in L
  add the "unused" rows in C to *pivots* and set them as "used"

Procedure *InsertRight*$(\mathcal{X}, \mathcal{Y}, p, M, N)$

  let $\mathcal{Z}$ be a class distinct from $\mathcal{X}$ in $M \cup N$
  return ($\mathcal{Y}$ is somewhere on the left of $\mathcal{Z}$)

Procedure *AddPivot*$(\mathcal{X}, \mathcal{Y})$

  let $\mathcal{Z}$ be the smallest class between $\mathcal{X}$ and $\mathcal{Y}$
  add the "unused" rows in the union of the columns in $\mathcal{Z}$ to *pivots* and set them as "used"

The correctness of this algorithm follows from this invariant:

- ***A:*** *if M has the consecutive ones property, then there exist a consecutive permutation compatible with the ordered partition L*
- ***B:*** *the set of "used" rows is the same for every column of a given class $\mathcal{X}$ of L*

Invariant: *The partition L always verifies A and if L does not verify B then some pivot in* pivots *will strictly refine L.*

We now assume that M has the consecutive ones property. Let us remark that Property A implies that for any row p, the set S of columns containing p as one entry appears in consecutive classes of L.

To give an idea of the proof, we just mention of properties proved in [6] that shows the correctness of the procedure *LaunchPartition*. First, there exist a consecutive permutation that ends with the last column numbered by a Lex-BFS ordering σ of the columns of M [9]. Then the first call to *LaunchPartition* preserve the invariant. When B is true, the authors of [6] proves that for any class $\mathcal{X}$ of L, σ induces a Lex-BFS ordering of the sub-matrices induced by the columns of $\mathcal{X}$. In that case, the refinement process can be launch again by the procedure *LaunchPartition*.

The complexity of Algorithm 6 is $O(m \log n)$. The reader should note that this complexity can be improved to linear time. In the presented algorithm the bottleneck is the choice of the pivot. It is shown in [6], that a clever choice of the pivot can be done using a special tree structure. By applying the Hopcroft's "process the smaller half" strategy we have obtained an extremely simple algorithm for testing the consecutive ones property.

6 Conclusions

Paige and Tarjan [13] conclude their introduction by "Although these applications are very different, the similarities among them are compelling and suggest further investigation of the underlying principles." We hope that this paper provides a step in this direction.

Acknowledgment

The authors wish to thank J.E. Pin for drawing their attention to the connexion between automata minimization and partition refinement.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Design and analysis of computer algortihms*. Addison-Wesley, 1974.

[2] J. Amilhastre. Représentation par automates de l'ensemble des solutions d'un problème de satisfaction de contraintes. Technical Report 94056, LIRMM, 1994.

[3] K.S. Booth and G.S. Lueker. Testing for the consecutive ones property, interval graphs and graph planarity using pq-tree algorithms. *J. Comput. and Syst. Sciences*, 13:335–379, 1976.

[4] D.G Corneil, S. Olariu, and L. Stewart. The ultimate interval graph recognition algorithm. Extended abstract, 1997.

[5] E. Dahlhaus, J. Gustedt, and R.M. McConnell. Efficient and practical modular decomposition. In *Proc. of SODA*, 1997.

[6] M. Habib, R. McConnell, C. Paul, and L. Viennot. Lex-bfs and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. *Theoritical Computer Science*, 1997. to appear.

[7] J.E. Hopcropft. A nlogn algorithm for minizing states in a finite automaton. *Theory of Machine and Computations*, pages 189–196, 1971.

[8] W.L. Hsu. A simple test for the consecutive ones property. In *LNCS 650*, pages 459–468, 1992.

[9] N. Korte and R. Möhring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. of Comp.*, 18:68–81, 1989.

[10] A. Lubiw. Doubly lexical orderings of matrices. *SIAM Journ. of Algebraic Disc. Meth.*, 17:854–879, 1987.

[11] R.M. McConnell and J.P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of comparability graphs. In *Proc. of SODA*, pages 536–545, 1994.

[12] R.M. McConnell and J.P. Spinrad. Linear-time modular decomposition and efficient transitive orientation of undirected graphs. In *Proc. of SODA*, 1997.

[13] R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journ. of Comput.*, 16(6):973–989, 1987.

[14] Donald J. Rose, R. Endre Tarjan, and George S. Leuker. Algorithmic aspects of vertex elimination on graphs. *SIAM J. of Comp.*, 5(2):266–283, June 1976.

[15] J.P. Spinrad. Graph partitioning. preprint, 1986.

[16] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. of ACM*, 22:215–225, 1975.