Partition Refinement:

a meaningful technique for Graphs

Qinna Wang*
qinna.wang@ens-lyon.fr

ABSTRACT

The `partition refinement` is a meaningful technique for applications in graphs. Currently, the applications in aim of the graph problems like chordal graphs [7], permutation graphs [6] and modular decomposition [3] focus on increasing the efficiency. In those propositions, the partition refinement plays an important role for the resolutions. Therefore, we describe this efficient technique which overcomes the constraints such as the unreasonable complexity and ambiguous understanding of certain applications that are used for graph problems in this paper. Through the illustration with various examples, we will show how efficient the algorithm become according to the partition refinement.

Keywords

graph theory, partition refinement

1. INTRODUCTION

The *graph theory* has been developed for several decades. Since the first graph theory paper [1] was written by *Leonhard Euler* on the *Seven Bridges of Konigsberg* and published in 1736, the graph theory has been used for resolving multiple problems in different domains such as *computation science*, for example, [5] has shown that the hyper graph partition is considerable for parallel scientific computing.

Generally, a good graph algorithm should be efficient and simple to understand while it is used to resolve complicated problems. The focus of this paper is the *partition refinement* which is a meaningful technique to simplify and optimize the applications in several domains such as in *graphs*.

In traditional, *pivots* are used to split the classes. The most classical algorithm with respect to the pivot rule is *Quick Sort* which is used for sorting integers. In this case, the final partition is the ordering elements. Note the complexity of `Quick Sort` which is really considerable. Therefore, a similar conceptual technique is then generated for developing algorithms, which is the *partition refinement*.

In the domain of the graph, the partition refinement is developed to split a class of partition vertices into disjoint subclasses according to the adjacencies between vertices. In this way, the underlying information of graphs is able to mined and particular graph problems can be resolved. For example, there are several proposed partition refinement applications for *twin vertex calculation* [6], *chordal graphs recognition* [7] and *permutation graphs recognition* [6].

For the valuable contributions of the partition refinement, we introduce it in this paper which is organized as: in section 2, we present the partition refinement in detail; in section 3, we describe several classical partition refinement applications based on the ordering of elements; we then list the categories of the partition refinement applications according to different citations in section 4; we finally provide the discussion on the partition refinement and give a conclusion of this paper.

2. THE PARTITION REFINEMENT

Explicitly, all applications referred to the partition refinement can be described in a general framework which is shown in this section.

2.1 definition and notation

Firstly, we introduce the basic definitions and notations related with the partition refinement.

DEFINITION 1. *The* `partition` $\mathcal{P}$ *of a set* E *is the collection of disjoint subsets* $\{\chi_1, \chi_2, ..., \chi_k\}$ *where* $\cup_{1 \leqslant i \leqslant k} \chi_i = E$. *We call the subset collection as* `classes`.

DEFINITION 2. *The set* S `intersects strictly` *with the set* S' *if and only if* $S \cap S' \neq \emptyset$ *and* $S \nsubseteq S'$. *Note that* $S' \subseteq S$ *is possible when the set* S *intersects strictly with the set* S'.

DEFINITION 3. *The* `refined partition` $\mathcal{P}' = refine(\mathcal{P}, S)$ *where* $\mathcal{P} = \{\chi_1, \chi_2, ..., \chi_k\}$ *and* S *is the subset of* E, *is obtainned by replacing the classes* χ_i *with* χ_{ia} *and* χ_{ib} *where* $\chi_{ia} = \chi_i \cap S$ *and* $\chi_{ib} = \chi_i \setminus S$.

*ENS Lyon, Laboratoire de l'information du ParallÃl'lisme,69364 Lyon cedex, France

DEFINITION 4. *The partition $\mathcal{P}$ is* `stable` *for the subset S, if and only if any class χ_i doesn't intersect strictly with the set S. i.e,$\mathcal{P} = refine(\mathcal{P}, S)$.*

DEFINITION 5. *In case of an* `ordering partition` *$\mathcal{P} = \{\chi_1, \chi_2, ..., \chi_k\}$, elements in each class χ_i are ordered.*

DEFINITION 6. *The ordering partition $\mathcal{Q}$ is* `compatible` *with another ordering partition $\mathcal{P}$ if and only if $\mathcal{Q} \preceq \mathcal{P}$ which means that for each class χ in $\mathcal{Q}$ there exists another class γ in $\mathcal{P}$ and $\chi \subseteq \gamma$ and the order of elements in χ respects for the order in γ.*

We then introduce the data structure in the partition refinement: the double chain list $L(E)$ is used to store the element in the set $E = \{x_1, x_2, ..., x_k\}$. Each element x has a pointer to its partition class χ and each class χ has two pointers to its minimal and maximal elements in $L(E)$. Hence, the insetting and deleting operations are easy to realize in the partition refinement algorithms. Therefore, the partition refinement which deletes the class $\chi_a = \chi \bigcap S$ and inserts it to the right of the class χ has reasonable computational time with the pointer structure. Further, marque the ordering of the list $L(E)$ which is related with the ordering of the partition classes and the compatibility of the classes. Hence, the partition refinement is useful for applications concerning with the ordering of elements.

2.2 the partition refinement

Secondly, we show the total algorithm of the partition refinement. In order to well understand it, we present a general algorithm at the beginning. Then we describe its basic procedures with explications. At end, we provide the algorithm of the partition refinement in global view.

Input: the partition $\mathcal{P} = \{\chi_1, \chi_2, ..., \chi_k\}$
Output: the refined partition $\mathcal{P}' = \{\chi'_1, \chi'_2, ..., \chi'_m\}$
begin
  pivots $= \emptyset$
  while *the InitPartition ($\mathcal{P}$) doesn't stop the loop* **do**
    InitPartition ($\mathcal{P}$)
    while *the Pivots* $\neq \emptyset$ **do**
      select an element $\mathcal{E}$ from the *Pivots*
      ClassPivot ($\mathcal{E}$)
    end
  end
end
Algorithm 1: a general algorithm for the partition refinement

Now, we illustrate the general algorithm which is shown in Algorithm 1.

- *InitPartition ($\mathcal{P}$) is essential for the total algorithm. It is used to start the following procedures and indicates the end of the total algorithm. One of its important contribution is used to add the known pivots into Pivots. Additionally, the InitPartition ($\mathcal{P}$) can be considered as the beginning of the recursive process. Therefore, the procedure of InitPartition ($\mathcal{P}$) is important. Note that some applications arrives their strategy only by executing InitPartition ($\mathcal{P}$) one time such as twin vertex calculation.*

- *ClassPivot ($\mathcal{E}$) is a procedure to produce the subset S according to the pivot p which is implied by the element E. It is related with the strategy of the application. Hence, there are two versions of the procedure ClassPivot: only one pivot p is used for the procedure $refine(\mathcal{P}, S)$ or several pivots are used for $refine(\mathcal{P}, S)$. For the first version of ClassPivot, the operation of refine($\mathcal{P}, S$) is executed several times. However, the refine($\mathcal{P}, S$) is only run one time. Therefore, we consider that the first version of ClassPivot saves the space for pivots.*

We then discuss the procedure of *ClassPivot* according its two versions:

Input: an element $\mathcal{E}$ in *Pivots*
Output: the stable partition for the subsets S which are generated by $\mathcal{E}$
for *each element* $x \in \mathcal{E}$ **do**
  $p \leftarrow (x, \mathcal{E})$
  $S \leftarrow PivotSet(p)$
  refine($\mathcal{P}, S$)
end
Algorithm 2: $ClassPivot(\mathcal{E})$

Input: an element $\mathcal{E}$ in *Pivots*
Output: the stable partition for the subset S which is generated by $\mathcal{E}$
begin
  $S \leftarrow \emptyset$
  for *each element* $x \in \mathcal{E}$ **do**
    $p \leftarrow (x, \mathcal{E})$
    $S \leftarrow S \cup PivotSet(p)$
  end
  refine($\mathcal{P}, S$)
end
Algorithm 3: *ClassPivot* ($\mathcal{E}$)

In the procedure $ClassPivot(\mathcal{E})$, p is the pivot to produce the subset S where $S \subseteq E$ and one element $x \in \mathcal{E}$ is used to produce an unique pivot p.

Now, we observe the Algorithm 2 and Algorithm 3: the Algorithm 2 executes the operation *refine($\mathcal{P}, S$)* with one subset S. Distinct from it, the other Algorithm 3 runs the operation *refine($\mathcal{P}, S$)* with several subsets S. The second version of *ClassPivot* ($\mathcal{E}$) is used to minimize the determined determined automate.

Next, we present the key algorithm $refine(\mathcal{P}, S)$ which is used to refine the current partition with the set S.

- The operation *refine($\mathcal{P}, S$)* is to split the partition class χ into the class χ_a and χ_b, where χ_a=$\chi \bigcap S$ and $\chi_b = \chi \backslash S$. Then we insert the new class χ_a next to the partition class χ, where the current class $\chi = \chi_b$. Note

Input: the partition $\mathcal{P} = \{\chi_1, \chi_2, ...\chi_k\}$ for the set E and subset S, where the class χ_i intersect strictly with S
Output: the partition $\mathcal{P}' = \{\chi'_1, \chi'_2, ...\chi'_m\}$,which is stable for S

```
for each class χ do
    χ_a=χ ⋂ S
    if χ_a ≠ ∅ and χ\S ≠ ∅ then
        remove χ_a from χ
        if InsertRight(χ, χ_a, p) then
            insert χ_a to the right of χ
        end
        else
            insert χ_a to the left of χ
        end
        AddPivot(χ, χ_a)
    end
end
```

Algorithm 4: refine($\mathcal{P}, S$)

that the class χ_a implies the ordering of the list $L(E)$ and the position of the new class χ_a depends on the strategy of applications.

- *InsertRight*(χ, χ_a, p) is used to determine whether insert the class χ_a in the right of χ. It depends on the strategies of applications and the expectation of the operator. If the return value is true, we then insert χ_a to the right of χ; otherwise, insert χ_a to the left of χ.

- *AddPivot*(χ, χ_a) is used to update the *pivots*. With respect to the new partition $\mathcal{P}'$ and the new class, the set *pivots* must be updated for the relations with the subset $S_a = \{x \in \chi_a : p_a \leqslant x\}$ and $S_b = \{x \in \chi_b : p_b \leqslant x\}$.

- The computational complexity referring to the data structure: all elements in E is stored in a double chain list and each partition class " touch " its elements in $L(E)$ with two points, the total operation of the partition refinement can be considered to delete all elements of χ_a in $L(E)$ and add an interval χ_a next to the interval χ simultaneously(Once one element x is in χ_a, add a pointer to the interval χ_a). And the class χ_b is presented by the ancient χ. Therefore, the total computational complexity is in time $|S|$ which presents the operation time on the interval χ_a. Note the position of element x in the list $L(E)$ doesn't change, thus the ordering of elements in χ_a respects for the ordering of elements in $L(E)$. We state that the final partition implies the ordering.

- In the domain of graph theory, all elements in the set E can be considered as the vertex set $\mathbb{V}$ in graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$. The partition classes $\chi_1, \chi_2, ..., \chi_k$ provide the adjacency list of the graph. As a result, the algorithm of the partition refinement is meaningful for the problems in graphs such as recognizing chordal graphs and permutation graphs.

At the end of this section, we show the global algorithm of the partition refinement in the following:

Input: a partition $\mathcal{P} = \{\chi_1, \chi_2, ..., \chi_k\}$ for the set E
Output: the refined partition $\mathcal{P}' = \{\chi'_1, \chi'_2, ..., \chi'_m\}$

```
begin
    Pivots← ∅
    while the InitPartition (P) doesn't stop the loop do
        InitPartition (P)
        while Pivots ≠ ∅ do
            select an element E from the Pivots
            ClassPivot (E)
            for each element x ∈ E do
                p ← (x, E)
                S ← PivotSet(p)
                refine(P, S)
                if the current partition P isn't stable for S
                then
                    Let the set of classes χ which intersect
                    strictly with S be denoted as M
                    for each class χ ∈ M do
                        χ_a = χ ⋂ S
                        remove χ_a from χ
                        if InsertRight(χ, χ_a, p) then
                            insert χ_a to the right of χ
                        end
                        else
                            insert χ_a to the left of χ
                        end
                        AddPivot(χ, χ_a)
                    end
                end
            end
        end
    end
end
```

Algorithm 5: the global algorithm of the partition refinement

Algorithm 5 shows the global algorithm for the partition refinement which is used for applications in aim of different problems. Depending on the basic procedures such as the *InitPartition($\mathcal{P}$), ClassPivot($\mathcal{E}$)* and *refine($\mathcal{P}, S$)*, it splits the partition classes into the singleton classes or the congruent classes[1].

We then study the complexity of the global algorithm for the partition refinement. We firstly note the size of *Pivots* which contains the element $\mathcal{E}$. We have learned that the element $\mathcal{E}$ is used to produce the subset S. When the size of *Pivots* unknown, it's difficult to estimate the running time. Note the case such as the *twin vertex calculation*, all elements are considered as the pivots during the total procedure. In this case, the size of *Pivots* is limited to the number of the elements in set E. Then the computational time on pivot selection is in $O(n)$ where n is the number of elements in set E. We secondly consider the time for the partition refinement. Reviewing the time for the procedure *refine($\mathcal{P}, S$)*, it is in time $O(|S|)$. Considering the total procedures in *refine($\mathcal{P}, S$)*, we define the running time is $m = \sum |S|$. In conclusion, the algorithm for the partition refinement costs $O(n + m)$ in time. Note that the above analysis is based on the known *Pivots* which depends on the size of the set E. In case of unknown pivots, the total computational time may be in $O(n + mlogn)$ which is proposed by Hopcroft [4] for "process the small half" pivot rule.

2.3 invariant

In [8], it proposes the invariant to prove the correctness of the partition refinement:

invariant: the partition verifies the property A; if the partition fails to verify the property B, there are partition classes which intersect strictly with the subset S, where S is related with the current *Pivots*.

- **A**: some properties imply the existence of a resolution which is compatible with the partition $\mathcal{P}$
- **B**: some properties indicate the partition $\mathcal{P}$ corresponding to the resolution

The **invariant** will be used to prove the correctness of the partition refinement applications. Here, we take the `Quick Sort` as an example.

PROOF. **invariant**

- **A**: for each element x which is in front of y in the resultant list E, then $x < y$.
- **B**: all classes in the final partition are singleton.

The inputting data is the ordering partition $\mathcal{P}$ for the set E. And the *Pivots* can be defined in the *InitPartition* ($\mathcal{P}$). According to the procedure *ClassPivot($\mathcal{E}$)*, the partition $\mathcal{P}$ is divided into the singleton classes. □

[1]The classes in the partition are equal to each other. It also implies that elements in each class are ordered independently

With respect to the other partition refinement applications, we then analyze the proof related with **invariant**: once the property A is verified by the partition then the procedure *InitPartition* is executed with iterative until the partition maintains the property B. It implies the contribution of the procedure *InitPartition* for the practical application for the correctness.

3. APPLICATIONS

Based on the algorithm of the partition refinement, we then present several partition refinement applications with different strategies. Considering the differences between applications, we present them according to their problems: the partition classes are congruent or the classes are sorted. Additionally, the invariant is also used to prove the correctness of applications.

3.1 unordered partition refinement

The applications like twin vertex calculation [8], deterministic automation and modular decomposition [3] imply the unordered partition refinement. In this case, each partition class doesn't care the order of its elements and the total partition classes implies the congruent relations. So we consider that such applications resolve the problems by the congruent classes or congruent relations between objects.

We then take the *twin vertex calculation* as an example. Firstly we introduce the twin vertices.

DEFINITION 7. *`twin vertices` are defined to be two vertices in one graph have the same neighbours. It is obvious that being twin vertices corresponds to an congruent relation. Thus, the calculation is a procedure of unordered partition refinement.*

DEFINITION 8. *If two vertices in one graph have $N(u) = N(v)$ where $N(u)$ is the open neighbours of vertex u that excludes the vertex u itself, they are defined to be `fake twin vertices`*

DEFINITION 9. *Two vertices are defined to be `true twin vertices`, if and only if they have $N[u] = N[v]$ where $N[u]$ is the close neighbours of the vertex u that contains the vertex u itself.*

Secondly, we introduce the algorithm for calculating the twin vertices in the following:

- In this case, one execution of the procedure *InitPartition* ($\mathcal{P}$) is enough for the calculation where the pivots correspond to the adjacency lists of vertices. After the *InitPartition* ($\mathcal{P}$), the partition classes then are divided into the singleton classes according to the procedure $refine(\mathcal{P}, S)$. Therefore, the algorithm stops when it starts *InitPartition* ($\mathcal{P}$) in second time. Additionally, the procedure *AddPivot(χ, χ_a)* does nothing because the calculation doesn't need more pivots.

Input: the vertex set $\mathbb{V}$ in a graph $\mathbb{G}$ with the adjacency lists of vertices
Output: the partition $\mathcal{P} = \{\chi_1, \chi_2, ..., \chi_k\}$ where each class is composed by twin vertices
begin
  $\mathcal{P} \leftarrow \{\mathbb{V}\}$
  execute the Algorithm 5 with the following procedures
end
procedure *InitPartition* $(\mathcal{P})$ **begin**
  add all vertices in $\mathbb{V}$ to *pivots* at the first execution
  exit and return the partition $\mathcal{P}$ at the second execution
end
procedure *PivotSet* $(p = (x, \chi))$ **begin**
  the adjacency list of the vertex x
end
procedure *InsertRight*(χ, χ_a, p) **begin**
  true
end
procedure *AddPivot*(χ, χ_a) **begin**
  do nothing
end

Algorithm 6: calculating the twin vertices

- **invariant:**the partition verifies the property A; if the partition fails to verify the property B, there are partition classes which intersect strictly with the subset S, where S is related with the *Pivots*.
  - **A**: if vertex x and y are twin vertices, they belong to the same class.
  - **B**: the partition $\mathcal{P}$ is stable for the $N(u)$ where the u is a vertex in graph $\mathbb{G}$ [2]

  proof: It's obvious to observe the correctness of such an algorithm: if the vertex x and y are twin vertices, no vertex in $N(u)$ can separate them to different classes. And if the property B can't be verified, there exist two vertices x and y belonging to the same class which can be separated by one element in $N(u)$. The reason is that the pivot p which split the vertex x and y hasn't been selected and the algorithm is in the execution.
- This algorithm 6 is in linear time. For each *PivotSet* $(p = (x, \chi))$, it is related with the pivot vertices. According to the number of pivot vertices is limited by the number of vertices, we estimate the computational time for *PivotSet* $(p = (x, \chi))$ as $O(n)$. For the procedure of *refine*$(\mathcal{P}, S)$, it depends on the adjacency lists which is implied by the edges in the graph. Therefore, the total *twin vertex calculation* costs $O(n+m)$ in time where n is the number of vertices and m is the number of edges in graph $\mathbb{G}$.

Therefore, the twin vertices can be calculated by the partition refinement application. And the partition refinement application has the reasonable running time and it isn't complicated for the understanding. We also note that the partition classes aren't single for the existences of the twin vertices.

[2]If the algorithm is used to calculate the true twin vertices, replace the $N(u)$ by $N[u]$. It means the partition $\mathcal{P}$ is stable for the $N[u]$ where the u is a vertex in graph $\mathbb{G}$ in case of true twin vertex calculation.

3.2 ordered partition refinement

Some applications such as Lex-BFS concerns on the ordering of the partition. In this case, each class in refined partition yields the ordering of the inputting data.

We have introduced the unordered partition refinement with an example of the twin vertex calculation. In fact, we can consider the twin vertex calculation as an ordered algorithm which respects for the order of the inputting data. Here we define the original partition as $\mathcal{P}$ and the refined partition as $\mathcal{P}'$. Therefore, the partition P' is compatible with the partition $\mathcal{P}$: i.e. if $x <_P y$, then $x <_{L(E)} y$.

Now, we discuss how to use the partition refinement for lexicographical literature.

DEFINITION 10. *the* `lexicographical order` *comes from the order of letters in dictionary: for a sequence of letters* $a1a2...ak$ *appear before* $b1b2...bk$ *in a dictionary, if and only if there is a letter* j *where* $x(j) < y(j)$ *in alphabet order and* $x(i) = y(i)$ *for the other letters* i *where* $i < j$.[3]

DEFINITION 11. *Assume* $\sum$ *is a sequence of letters in alphabet order and* $\sum^*$ *is a set of strings which come from* $\sum$. *Additionally, the* i*-th letter which is in the string* x *belonging to the set* $\sum^*$ *is denoted to be* $x(i)$. *Under this condition, we consider the string* x *is in front of* y *according to the lexicographical order:* $x <_{lex} y$*,if and only if there is a letter* j *where* $x(j) < y(j)$ *in alphabet order and* $x(i) = y(i)$ *for the other letters* i *where* $i < j$ *or* x *is the strictly prefix of* y.

Apparently, the algorithm which is used to sort the lexicographical strings is classified into the ordered partition refinement. Hence, we take it as an example in this section.

Input: n strings $x1, x2, ..., xn$
Output: the partition $\mathcal{P} = (\chi_1, \chi_2, ..., \chi_n)$ of strings
begin
  $\mathcal{P} \leftarrow \{x1, x2, ..., xn\}$
  execute the Algorithm 5 with the following procedures
end
procedure *InitPartition* $(\mathcal{P})$ **begin**
  precompute the sets $S_{i,a}$ where $S_{i,a} \neq \emptyset$ and each $S_{i,a}$ has a fixed letter a at the fixed position i.
  add all sets $S_{i,a}$ in *pivots* at the first execution
  exit and return the partition $\mathcal{P}$ at the second execution
end
procedure *PivotSet* $(p = (x, \chi))$ **begin**
  the p
end
procedure *InsertRight*(χ, χ_a, p, M, N) **begin**
  true
end
procedure *AddPivot*(χ, χ_a) **begin**
  do nothing
end

Algorithm 7: lexicographical strings sorting

- The procedure *InsertRight*(χ, χ_a, p, M, N) refers to the M, N that implies the a, i.

[3]The string x is also possible to be the strictly prefix of y

- The complexity of this algorithm is in $O(n+k+m)$ where n is the number of strings, k is the size of the $Pivots$ and m is the $\sum |S_{i,a}|$. According to *precompute the sets* S_{ia}, the total procedure has to estimate the cost for pivot calculations which is $O(k)$. For $O(n)$ is far less than $O(m)$, the total complexity can be estimated in $O(k+m)$
- **invariant:**the partition verifies the property A; if the partition fails to verify the property B, there are partition classes which intersect strictly with the subset S, where S is related with the $Pivots$.
  - **A**: a lexicographical sort of the strings is compatible with $\mathcal{P}$
  - **B**: two different strings don't appear in the same class

  proof: It's obvious to observe the correctness of such an algorithm according to the *Pivots* and the property of the partition refinement.

Focus on the ordered partition refinement, the partition is generally refined until all classes yielding the order of their elements. It means that each final refined class is not necessary to be a singleton. The congruent classes are possible to exist.

4. LEX-BFS

According to the standard bread-first search failing to obtain an ordering list for vertices that are visited in a graph $\mathbb{G}$, the Lex-BFS is proposed to visit vertices in a graph in order with respect to certain properties. Therefore, the Lex-BFS is a resolution for graph problem. Here, we will present the Lex-BFS algorithm in detail.

- Lex-BFS [2] is used to recognize triangulated graphs in traditional.

  DEFINITION 12. *`Triangulated` graphs are also called `chord` graphs which are subsets of the perfect graphs. If each cycle composed by at least of four nodes has a chord, we call the graph is `chordal`, where `chord` is an edge joining two nodes that are not adjacent in the cycle.*

  [2] firstly indicates how to use the Lex-BFS for the problems of graphs. It numerates all vertices according to the visiting order. Each vertex has a label: $label(x)$. $label(x)$ is related with the adjacency lists. Once a visited vertex x is numbered i, the number i will be added into the labels of the non-visited neighbours of x. i.e. i will be added into the $label(y)$ where the y is the neighbour of x and y hasn't been visited. In the following execution, the size of $label(x)$ is considered to number the vertex x. In this way, the result $\mathcal{P} = \{\chi_1, \chi_2, ..., \chi_n\}$ implies the perfect elimination ordering of the graph $\mathbb{G}$.

  DEFINITION 13. *$\sigma = [x_1, x_2, ..., x_n]$ is a `perfect elimination ordering` of a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ if the neighbours of each vertex x_i is a clique of the induced subgraph $G_i = G[x_1, x_2, ..., x_n]$. For example, the σ =*

Input: a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ with its adjacency lists for vertices
Output: an ordering Lex-BFS list $\mathcal{P}$ for the vertices in set $\mathbb{V}$

begin
  $\mathcal{P} \leftarrow (\mathbb{V})$;
  execute the algorithm 5 with respect to the following procedures.
end
procedure *InitPartition ($\mathcal{P}$)* with respect to the i-th execution **begin**
  if $i = n$ **then**
    return $\mathcal{P}$: the end of the *procedure*
  end
  else
    select the vertex x from the class χ_i where χ_i is composed by the non-visited vertices and is located in the most right in the partition $\mathcal{P}$
    i.e.$\chi_{max}^{non-visit} \in \mathcal{P}$
    if $|\chi_i| \neq 1$ **then**
      replace the class χ_i by $\chi_i \backslash \{x\}$
      add $\{x\}$ to $\mathcal{P}$
    end
    add $\{x\}$ to *Pivots*
  end
end
procedure *PivotSet($p = (x, \chi)$)* **begin**
  return the adjacency list of the vertex x
end
procedure *InsertRight(χ, χ_a, p)* **begin**
  return *true;*
end
procedure *AddPivot(χ, χ_a)* **begin**
  nothing to do
end

Algorithm 8: Lex-BFS

$[f, g, a, e, c, b, d]$*, then the subgraph* $G_i[e, c, b, d]$ *may be a cycle.*

As known, the chordal graph can be specified by the perfect elimination ordering. So, the Lex-BFS is really useful for the graph problems such as chordal graph recognition.

- Reviewing the Lex-BFS algorithm, it is easier to understand with the framework of the partition refinement. Unlike the previous algorithms, pivots are unknown in this case. In fact, pivots are produced in *PivotSet(p)*. It implies one type of partition refinement applications that produce pivots during the execution. So the procedure of *ClassPivot(E)* corresponds to the Algorithm **??** in this case.

- We then discuss the implication through this algorithm. During the partition refinement process, each class implies the lexicographical ordering of its elements. i.e. if the vertex x and y have belonged to the same class χ, they may have the same prefix. And if the vertex x is split before the vertex y, then the $x <_{Lex} y$.

- We then discuss the complexity of the algorithm. The time for operations is related with the size of *pivots* which is the vertices in $\mathbb{G}$. And the procedure *refine(P,S)* costs $O(|\sum S|)$ in total. Note that the S depends on the edges between vertices. Therefore, we conclude that the total computational time for the algorithm of Lex-BFS is in $O(n + m)$ where n is the number of vertices and the m is the number of edges in graph.

- As before, we verify the correctness of this algorithm through the **invariant:**the partition verifies the property A; if the partition fails to verify the property B, there are partition classes which intersect strictly with the subset S, where S is related with the $Pivots$.

  - **A**: there is an lexicographical ordering of the vertices in graph $\mathbb{G}$ which is compatible with the *Pivots*.
  - **B**: all classes are in singleton.

  proof: The lexicographical ordering is compatible with the *Pivots* for the property of the partition refinement and the partition refinement operation: delete the subset S from the class χ and add it to the right of χ and each element which is used to produce the pivot p for the next operation is selected from the class χ where $\chi_{max}^{non-visit} \in \mathcal{P}$. Additionally, the process is ended when $i = n$, where n is the size of the vertex set V. From the procedure of the algorithm, one no-visited vertex x is selected to $InsertRight(\chi, \chi_a, p = (x, \chi))$. After n executions of *InitPartition* ($\mathcal{P}$), each class is a singleton.

Here, we only show the Lex-BFS algorithm. In fact, it provides a perfect elimination ordering. According to its speciality, it has been applied for diverse problems. [**?**] uses it to recognize permutation graphs which are the comparability graph. [7] uses it to recognize the chordal graph. And [7] even develops it for co-chordal graphs and co-interval graphs. Therefore, the algorithm of partition refinement is really interesting and meaningful for graphs.

5. CATEGORIES

We have presented the algorithm of the partition refinement according to diverse applications that are classified by the ordering of elements in partition classes.

In fact, the partition refinement applications can be classified to diverse categories according to different critations. We have introduced the ordering. Next we will introduce the *pivot rules*.

According to the descriptions on different partition refinement applications, pivots play the key role in each algorithm, especially it is related with the complexity of the algorithm. If the pivots are known in advance. Then the application is a linear time algorithm. Otherwise, the computational time is complicated to estimate. Therefore, pivot rules are used as a citation for classify the categories of applications:

- *simple rule* is used to describe the applications that execute with the known pivots. In this case, there doesn't exist any rule to select pivots. Probably, the applications like Lex-BFS use simple rules such as arbitrary choice to determine pivots. Therefore, the complexity of applications in this category is linear.

- *Hopcroft* is a popular rule for pivots. We have presented that pivots can be generated during the procedure of the partition refinement. *Hopcroft* is proposed by Hopcroft [4] which applies the "process the smaller half" strategy. It benefits to applications which produce pivots during the procedure such as *coarse partition computation*. With the help of *Hopcroft*, the pivot selection becomes easier and the complexity is more efficient. Generally, applications based on such a pivot rule is in time $O(mlogn)$ where m is the size of inputting data and n is the number of elements.

Furthermore, the applications also can be classified according to "tie-break rule" which is related with the execution of *InitPartition*. We have observed that some applications like *twin vertex calculation* finish their partition refinement procedures when the procedure *InitPartition($\mathcal{P}$)* is recalled in second time. However, some applications such as Lex-BFS execute the procedure of *InitPartition($\mathcal{P}$)* several times to execute the procedure *InitPartition($\mathcal{P}$)* until the current partition classes are congruent even single. Therefore the "tie-break rule" is generated which focus on breaking the recursive process.

6. CONCLUSIONS

We conclude that the partition refinement is a meaningful technique in graph theory. It can be applied by different applications for diverse strategies. It is composed by several procedures: *InitPartition($\mathcal{P}$)*, *PivotSet*$(p = (x, \chi))$,*InsertRight* (χ, χ_a, p) and *AddPivot* (χ, χ_a). For each process *refine($\mathcal{P}, S$)*, each new class $\chi_a = \chi \bigcap S$ is removed from the class χ and inserted next to the class χ in the partition P. According to the data structure that all elements in E are stored in a

double chain list and the partition refinement is related with the pointers which are used for the partition class χ "touch" its elements in $L(E)$. Thus, it preserves the initial ordering of elements. So it is used to resolve the problems in graphs, such as twin vertex calculation, chordal graph recognition and interval graph recognition.

In this paper, we introduce the different categories of partition refinement applications according to different citation. We also present some classical partition refinement applications for graph problems such as twin vertex calculation and Lex-BFS, the results are considerable with reasonable running time. It implies that the partition refinement give insights for graph problems.

Note the properties of the partition refinement, it is efficient and simple to understanding. With respect its contributions such as Lex-BFS, it is meaningful for graph calculations.

While the partition refinement is used for graph algorithms, the element set E is usually the vertex set $\mathbb{V}$, each class χ implies the properties of the graph $\mathbb{G}$ and the *pivots* are related with the adjacency lists. Additionally, the execution of the partition refinement application is based on the inputting data. If the inputting data is subgraphs such as clique as the element E, the algorithm is also accepted for the calculation. Therefore, we conclude that the partition refinement has a good prospective in the domain of graphs.

7. ACKNOWLEDGMENTS

8. ADDITIONAL AUTHORS

9. REFERENCES

[1] *Graph Theory*. Oxford University Press, 1986.

[2] G. S. L. D.J. Rose, R.Endre Tarjan. Algorithmic aspects of vertex elemination on graphs. *SIAM J.of Comp.*, 5(2):266–283, June 1976.

[3] R. M. E.Dahlhaus, J.Gustedt. Efficient and practical modular decomposition. *Proc. 7th Ann ACM-SIAM Symp.*, 1997.

[4] J. Hopcroft. An *nlogn* algorithm for minimizing states in a finite automaton. In *Theory of machines and computations*, pages 189–196. Academic Press, 1971.

[5] R. H. R. B. U. C. KD Devine, EG Boman. Parallel hypergraph partitioning for scientific computing. *20th International Parallel and Distributed Processing*, 2006.

[6] C. P. Michel Habib and L. Viennot. A synthesis on partition refinement: a useful routine for strings, graphs, boolean matrices and automata. *15th annual Symposium on Theoretical Aspects of Computer Science*, 1998.

[7] C. P. Michel Habib, Ross McConnell and L. Viennot. Lex-bfs and partition refinement with applications to transitive orientation, interval graph recognition and consecutive ones testing. *theoretical computer science*, 2000.

[8] C. PAUL. *Parcours en Largeur Lexicographique: un algorithme de Partitionnement Application aux Graphes et Generalisations*. PhD thesis, UNIVERSIT MONTPELLIER II, 1998.