Ibn Sina School for Computer Science

Continuity Project

Detecting Graph Symmetries

Saucy_Basic

Students:
Ahmed Jazzar
Bassam Jaber

Supervisor:
Prof. Karem Sakallah

December 19, 2012

1 Introduction

The applications of the graph are not limited on computer science needs, as we will find applications in SAT, we will find other applications in another fields like modelling of chemical compounds, representing migration path or movement between the regions, modelling transport networks, logistic optimization, and other applications.
There are many shapes that we see everyday, but we didn't even think to see it in a symmetrical way. This symmetry that surrounding us starting from human body reaching to our small hummer; gives us a completely different way of thinking. Graph symmetry isn't that different, it's basically a permutation of the nodes of a graph that leaves the edges of the graph unchanged.

What's described below is our work in finding graph symmetries using `Saucy` tool.

2 Step by Step

2.1 Input Parser

Input parser deals with a `text` file consisting of a header followed by a listing of the graph edges. The parser analyses the first line and saves the basic information of the graph like edges number, vertices number,.. after that, reading edges and save it will be its job.[1] Using `input parser` is important to save the graph correctly so that we can deal with it easily later.

2.1.1 Data Structure

The data structure used to represent the `graph` is an `array` of `linked lists`; the `array` represents the `graph nodes`, and the `linked lists` represent the edges of each node. Figure 1 below describing graphically this data structure.

Data Structure advantages Using an `array` in representing the graph vertices is more comfortable in implementing ,and accessing[2] the nodes than

[1]An output file will be exported in both `text` and `dot` file depending on the data that parser read.

[2]the index of a vertex in the array is the same as its number

a `linked list`. What's helped us in this is that we also knew the number of vertices[3].
In another hand, we can't determine the `number of edges` for each vertex , so the most suitable way to represent it, is a `linked list`, meaning that a new edge discovered a new node connecting to the `list`.

Complexity If we have `n` vertices, then each node will connect to `n` vertices in worst case, that means we will read the `graph` in the worst case in a complexity of $O(n^2)$.

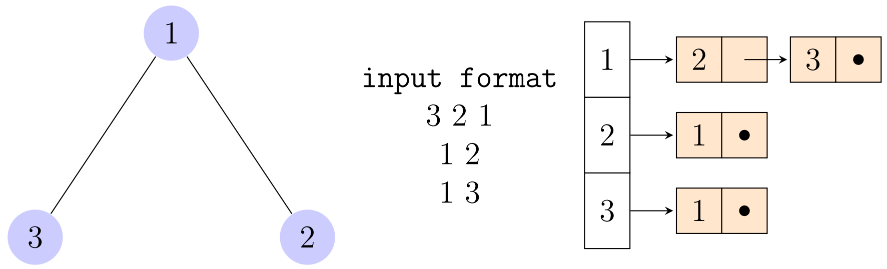

Figure 1: Example graph of an input format and its data structure

2.1.2 Algorithm

```
begin
  open input_file
  read header
  create graph[vertices_number]
  set color_change
  for  i to vertices_number do
      give graph[i] its color
  od
  comment: edges part
  while $\neg fileEnd$ do
        read i  j
        list j to graph[i]
        list i to graph[j]
  od
```

[3]the number of vertices is the first number that appears in the header

```
    close input_file
    make statistics_file ∧ dot_file
end
```

2.2 Ordered Partition

`Ordered Partition Pair` (OPP) is a generalization of tabular notation that allows for the implicit encoding of a set of permutations.[4]

2.2.1 Helper Arrays

We implement `helper` arrays in order to keep track of cell boundaries and lengths. The `helper` arrays are four:

- Vertex[i] : the number of the vertex in position i.
- Position[v] : the position of the vertex v.
- cellFront[v] : the position in the Vertex array corresponding to the first vertex in the cell containing vertex v.
- cellSize[f] : the size of the cell that starts with cellFront[v] minus 1.

The `flag` that will appear in `ordered partition algorithm` is a variable in each node tells us if the `cell` starts at the same node or not.

[4]Actually there are two ways to express graph permutations; tabular form and cycle notation.

2.2.2 Algorithm

```
begin
  comment: vertex_array builder
  for  j to colors_number do
     for  i to vertices_number do
        if  graph[i]  color  $= j$ then
                                        vertex_array[]  :=  graph[i]  vertex
        fi
     od
  od
  comment: position_array builder
  for  i to vertices_number do
     position[vertex_array[i]]  :=  i            just the inverse of vertex_array
  od
  comment: cell_front builder
  for  i to vertices_number do
     while  position_array[$vertex$] $\neg = -1$ $\wedge$  temp_color  $=$  color  do
          if  index $\neg = -1$
            then
                 temp_color  :=  graph[$vertex\_array$[i] $color$]
          fi
           index  :=  index  $+1$
     od
     cell_front[i]  :=  index
  od
  comment: cell_size builder
  while  i  $<$  size  do
       if  graph[$vertex\_array$[i]] color $\neg =$  begin_color  $\wedge$  i  $<$  size
         then
              cell_size[]  flag  $:= 1$
              cell_size[]  size  :=  counter
        else
              counter  $:= counter + 1$
       fi
       cell_size[]  flag  $:= 1$
       cell_size[]  size  :=  counter
  od
end
```

2.3 Partition Refiner

`partition refinement` is the main mechanism for distinguishing non-symmetric vertices in the graph. This distinguishing will be on color-relative vertex degree of each vertex.

2.3.1 Data Structure

The main data structure used in the partition refiner is `Queue`, and its used to process the cells in the partition refiner one by one, until emptying itself. The `Queue` implemented using a `linked list` instead of `array`. The idea behind this, is the unknown number of cells[5], so we will be free to add another cells if we need to.

The Challenge Deciding to choose a `linked list` instead of `array` costs us a constant time in normal case for popping and pushing. To reduce the cost of O(`n`) we look at the `Queue` as an object that has 3 components; `number of nodes` in the `Queue`, `pointer` to the last node in the `Queue`, and another `pointer` points to the `linked list`. And we used this object as a head of the `linked list`. The advantage will appear in using the pointer that points to the last node because it makes popping from the `Queue` faster since we already have the last node address.

[5]we know the initial number of cells, but in refinement procedure there's a probability to have another cells

2.3.2 Algorithm

```
begin
  refine (OP P){
  while ¬empty(Q) do
        I := pop(Q);
        for each cell T with connections to I do
            (OP P, Cell List L) := refineCell(OP P, cell T, cell I)
            if ¬empty(L)
              then
                   if T is in Q
                     then remove it
                   fi
                   for each cell T in L do
                       push(T, Q)
                   od
            fi
        od
  od
  }
  comment: start function refineCell
  refineCell(OP P, cell T, cell I)
  find the color relative degree for each vertex in T
  divide the vertices in T depending on thier degrees
  update position_array
  return OP P
end
```

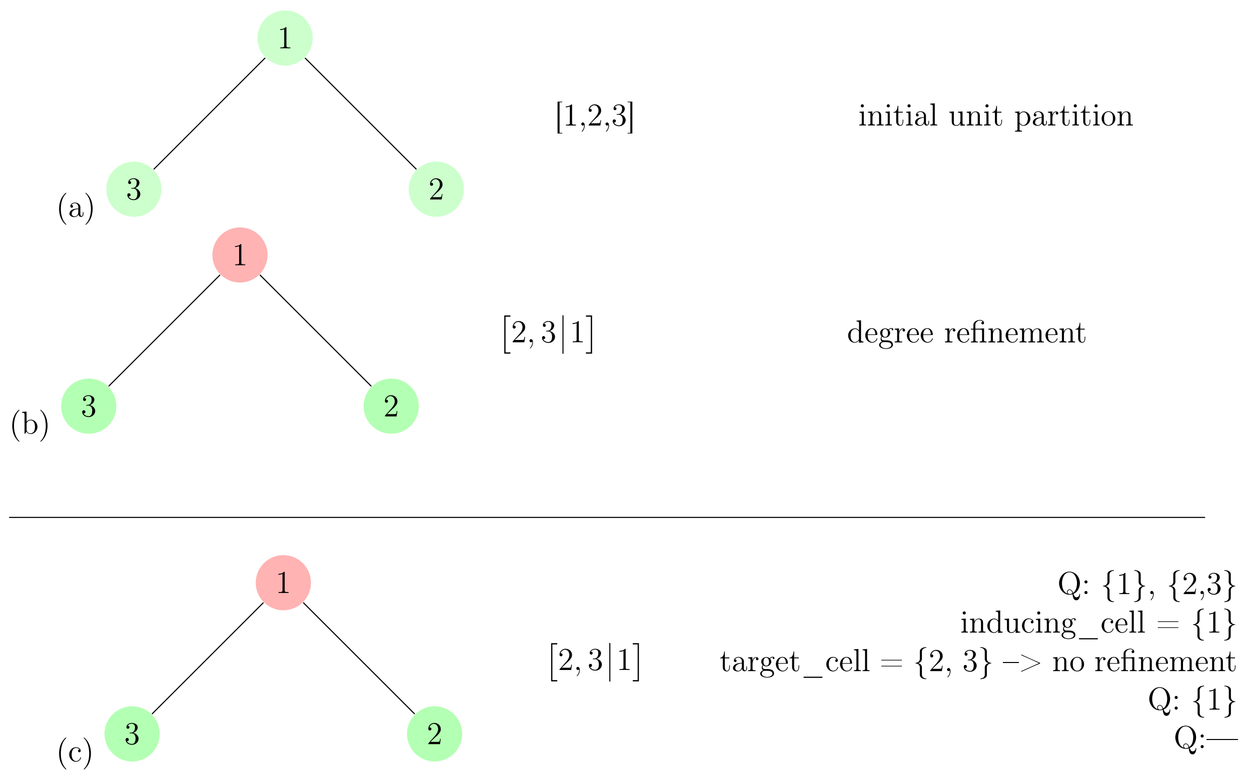

Figure 2: Example of Partition Refinement

2.4 Search Tree

The last part of the project is to find the symmetries from the whole permutations, and to generate the permutations we used a `basic search tree`.

2.4.1 Data Structure

The basic element in a tree is its node, our `tree` node here contains mainly an `OPP` and another data that helps the search doing its job perfectly like `number of children`, `array` of pointers to the children, and a colored `graph` that represents the `OPP` of the node.

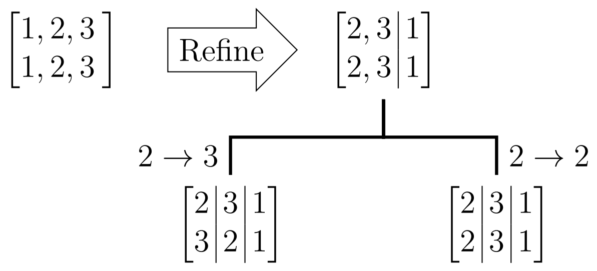

Figure 3: Example of search tree after refinement

2.4.2 Algorithm

```
begin
  T := createTree(OP P)
  searchTree( T, top, bottom)
  comment: start function searchTree
  searchTree( T, top, bottom){
  if singleton
    then
         comment: check if symmetry
         verticesPasser( top, bottom, top_graph, bottom_graph)
         return NULL
    else
         get number_of_children
         for i to number_of_children do
             mapVertices
             create T node
             searchTree( T, top, bottom)                    recursion
         od
         return T
  fi
  }
end
```

2.5 Checker

The `checker` is a procedure that checks if the given graph is symmetry or not by accepting two `OPP`s, top and bottom. The top represents the original graph vertices, and the second one represents where it should permute to. `Checker` recognizing the symmetries by checking the graph edges, if the edges are the same then the result will be symmetry, otherwise; it won't be symmetry.

2.5.1 Algorithm

```
begin
  verticesPasser( top, bottom, top_graph, bottom_graph){
  for i *to* vertices_number do
      swap the edges of bottom_graph[$i$] *with* top_graph[$i$]
  od
  for i *to* vertices_number do
      while next edge $\neg = NULL$ do
            *change* current_edge *to* mapped_edge
      od
  od
  for i *to* vertices_number do
      if $\neg$ check( i, top_graph, bottom_graph)
        **return** false
      fi
  od
  **return** true
  comment: check function compares the edges of i in the bottom wiht ones in the top
  }
end
```

3 Conclusions

The approach is not scalable because the number of symmetries can be exponential in the number of vertices and that in practice symmetry detectors produce a set of generators of the symmetry group that has at most n-1 generators.

4 Some Graphs Results

This section shows some results that come after applying some graphs on `saucy_basic`. The table below contains three columns, `input format` column shows the inputFormat of the graph, or graph name, `symmetries number` column gives the number of symmetries that discovered, and `average time` (AT) column which shows the time that our program took to find symmetries on two different operating systems; Linux and Ubuntu[6].

The tested graphs in the table tested on `Windows 8`, and `Linux Ubuntu 12.1` operating systems, `6 GB` Ram, and `AMD A6 Quad Core 1.6 GHz` processor.

[6]Note that the average time took based on 25 times running on both operating systems.

input format	symmetries number	AT(W)	AT(L)
5 0 1	120	0.4900s	0.0000s
3 2 1			
0 1			
0 2	2	0.0424s	0.0000s
4 6 1			
0 1			
0 2			
0 3			
1 2			
1 3			
2 3	24	0.1816s	0.0000s
8 14 3 3 4			
0 1			
0 2			
0 3			
1 2			
1 3			
2 3			
3 4			
3 5			
3 6			
3 7			
4 5			
4 7			
5 6			
6 7	48	0.3484s	0.0036s
cube graph(x = 2)	48	1.7406	1.018s