UNIVERSIDADE FEDERAL FLUMINENSE

Matheus S. D'Andrea Alves

Recognition of (r, ℓ)-partite Graphs

Niterói

2020

UNIVERSIDADE FEDERAL FLUMINENSE

Matheus S. D'Andrea Alves

Recognition of (r, ℓ)-partite Graphs

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Algoritmos e Otimização

Orientador:
Uéverton dos Santos Souza

Niterói

2020

Abstract

The complexity to recognize if a graph has an (r, ℓ)-partition, i.e. if it can be partitioned into r cliques and ℓ independent sets, is well defined(1). However, as we will demonstrate, the literature-stabilished values for those on the P class can be improved. The following work provides a set of strategies and algortihms that pushes the previous results for the $(2, 1)$-partite (from n^4 to $n * m$), $(1, 2)$-partite (from n^4 to $n * m$) and $(2, 2)$-partite (from n^{12} to $n^2 * m$) recognition.

Keywords— (r, ℓ)-graphs, (r, ℓ)-partitions

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Current results

In this section we will explore the current state of the complexity analysis as r and ℓ grows. As we fulfill the following table we expose the strategies and how those can be used to enlighten the more complex results.

The most trivial result is the recognition of the $(1, 0)$-graphs, as in order to recognize it we just need to know if $|E(G)| > 0$. Therefore, it's complexity is $\mathcal{O}(1)$.

$r \backslash \ell$	0	1	2	3	4	...
0	-	?	?	?	?	...
1	$\mathcal{O}(1)$	?	?	?	?	...
2	?	?	?	?	?	...
3	?	?	?	?	?	...
4	?	?	?	?	?	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Table 1.1: Incomplete complexity analysis of the (r, ℓ)-partite recognition problem

1.1.1 m-bounded results

Naturally, we wish to find the complexity for those problems of small partition cardinality.

At (2), König showed that it takes $\mathcal{O}(m)$ steps to recognize a bipartite graph. For complete graphs, is enough to check if $|E(G)| = n(n-1)/2$, if it doesn't then we have an answer; If it does, checking vertex by vertex if it's neighborhood contains all other vertex is $\mathcal{O}(m)$.

For co-bipartite graphs recognition, we need only to verify if it's complement is a bipartite graph. The recognition of a split graph can be done using their vertex degrees , obtaining all vertices degrees is $\mathcal{O}(m)$ therefore, the recognition of split graphs is $\mathcal{O}(m)$

1.1.2 $\boldsymbol{NP}$-Complete results

An adequate strategy at this moment is to find when the recognition problem gets NP-Complete.

At (3) is shown that 3-coloring a graph (i.e. assign a color between three possibles to each vertex such that no neighborhood repeats a color) is NP-Complete, it's trivial to see how the 3-coloring of a graph can be reduced to the problem of finding if a graph is a $(3, 0)$-graph, therefore the recognition of $(3, 0)$-graphs is NP-Complete.

It's noticeable that the recognition of (r, ℓ)-graphs is monotonic, and therefore if the recognition of $(3, 0)$-graphs are NP-Complete then the recognition of any $(r, 0)$-graph or $(3, \ell)$-graph is NP-Complete for $r > 3$ and $\ell > 0$.

We can extrapolate these findings and argument that the recognition of a $(0, 3)$-graph is also NP-Complete, as it is the same as recognize it's complement as a $(3, 0)$-graph, and use the property of monotonicity to state that the recognition of any $(r, 3)$-graph or $(0, \ell)$-graph is NP-Complete for $r > 0$ and $\ell > 3$.

$r \backslash \ell$	0	1	2	3	4	$\ldots$
0	-	$\mathcal{O}(m)$	$\mathcal{O}(m)$	NPc	NPc	$\ldots$
1	$\mathcal{O}(1)$	$\mathcal{O}(m)$	?	NPc	NPc	$\ldots$
2	$\mathcal{O}(m)$	?	?	NPc	NPc	$\ldots$
3	NPc	NPc	NPc	NPc	NPc	$\ldots$
4	NPc	NPc	NPc	NPc	NPc	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Table 1.2: Incomplete complexity analysis of the (r, ℓ)-partite recognition problem

1.1.3 Frontier results

Finally, the frontier cases $(1, 2)$,$(2, 1)$ and $(2, 2)$ were subject of studies by Brandstädt(1, 4). He's findings show that:

- Recognition of $(1, 2)$-graphs are $\mathcal{O}(n^4)$.

- Recognition of $(2,1)$-graphs are $\mathcal{O}(n^4)$.
- Recognition of $(2,2)$-graphs are $\mathcal{O}(n^{12})$.

$r \backslash \ell$	0	1	2	3	4	...
0	-	$\mathcal{O}(m)$	$\mathcal{O}(m)$	NPc	NPc	...
1	$\mathcal{O}(1)$	$\mathcal{O}(m)$	$\mathcal{O}(n^4)$	NPc	NPc	...
2	$\mathcal{O}(m)$	$\mathcal{O}(n^4)$	$\mathcal{O}(n^{12})$	NPc	NPc	...
3	NPc	NPc	NPc	NPc	NPc	...
4	NPc	NPc	NPc	NPc	NPc	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Table 1.3: Current complexity analysis of the (r,ℓ)-partite recognition problem

1.2 Brandstädt's recognition of $(2,1)$-graphs

In this section we will describe the algorithm designed by Brandstädt to recognize a $(2,1)$-graph.

Let G be a graph. In order for G to accept a $(2,1)$-partition a necessary condition is that for any vertex $v \in V(G)$:

- (N1) $N(v)$ induces a split graph.
- (N2) Or, $\bar{N}(v)$ should induces a bipartite graph

More specifically, if G has a $(2,1)$-partition I_1, I_2, C then (N2) holds for every vertex $v \in C$ and (N1) holds for the vertices $v \in I_1 \cup I_2$

Let A be the set of vertices that is satisfied by (N1), let B be the set o vertices satisfied by (N2) then if $R = A \cap B = \emptyset$ then we already have a $(2,1)$-partition of G. Otherwise

This conditions leads us to the following algorithm:

1: Brandstadt

```
def Get_21(Graph G) 2-1-Partition
  for v in V(G) do
    N = N(v) # open neighboorhood of v
    K = N̄(v) # V(G) - N
    if (!N.is_Split() and !K.is_Bipartite()) then
```

```
    return None; # G ∉ (2,1)
  end
  cl = { v in V(G) : !N(v).is_Split() and N̄(v).is_Bipartite() }
  bi = { v in V(G) : N(v).is_Split() and !N̄(v).is_Bipartite() }
  if ( !cl.is_Clique() or !bi.is_Bipartite()) then
    return None; # G ∉ (2,1)
  R = {v in V(G) : N(v).is_Split() and N̄(v).is_Bipartite()}
  if R.is_Empty() then
    return new 2-1-Partition {
      I_1: bi.I_1,
      I_2: bi.I_2,
      C: cl
    }
  else return AllocateR(bi,cl,R)
end
```

2: AllocateR

```
def AllocateR(bi Bipartite, cl Clique, R Graph)
  if R.is_Split() then
```

Chapter 2

Our strategy

2.1 Finding the 3,1 partition

As seen on the Brandstädt's algorithm every vertex of a 2,1 partition must at least be compliance to one of the neighborhood rules:

- (N1) $N(v)$ induces a split graph.
- (N2) Or, $\bar{N}(v)$ should induce a bipartite graph.

If every vertex match exactly one of these rules, then the Graph is known to be a 2,1 Graph. On the other hand if any vertex does not match N1 or N2 then the Graph is bound to not have a 2,1 partition. However, when a vertex matches both N1 and N2 the graph could have a 2,1 partition but clearly has a 3,1 partition as the union of the induced split of it's open neighborhood, and the bipartite induced by its complement describe such partition.

Therefore, the following algorithm can be declared.

3: Get 2-1-Partition

```
def Get_21(Graph G) 2-1-Partition
  cl = new Graph #intended clique
  bi = new Graph #intended bipartite
  for v in V(G) do # O(n)
      N = N(v) # open neighborhood of v
      K = N̄(v) # V(G) - N
      if N.is_Split() then # O(m)
        bi.Add(v)
      if K.is_Bipartite() then # O(m)
```

```
            cl.Add(v)
        if (!bi.contains(v) and !cl.contains(v)) then
          return None; # G ∉ (2,1)
    end # O(n*m)
    if cl.V ∩ bi.V = ∅ then
      if (!cl.is_Clique() or !bi._is_Bipatite()) then
        return None; # G ∉ (2,1)
      else return new 2-1-patition{
        C: cl
        I_1: bi.I_1,
        I_2: bi.I_2
      }
    else
      using any v ∈ cl.V ∩ bi.V
      N = N(v) # open neighborhood of v
      K = N̄(v) # V(G) - N
      tr = new 3-1-Partition{
            C: N.Clique,
            I_1: N.Independent,
            I_2: K.I_1,
            I_3: K.I_2,
      }
      return 3-1-to-2-1(tr)
    end
end
```

2.2 Odd Cycle transversal

The Odd Cycle transversal can be described as the problem of finding a subset of vertices of a graph such that when removed from said graph makes it a bipartite graph. This problem is known to be NP-Complete, however, Reed, Smith and Vetta presented a fixed-parameter tractable algorithm for this problem. Such algorithm obtain the desirable subset running in time $3^k n^{\mathcal{O}(1)}$ (5)

Is imperative we further analyze this algorithm, as it's strategy will be relevant for our next steps.

2.3 The critical paths

2.4 Finding the viable movements on the critical paths

2.5 The Algorithm

2.5.1 Pseudo-Code

4: 3-1-to-2-1

```
def 3-1-to-2-1(Graph-3-1 G)
  tri = G.I₁ ∪ G.I₂ ∪ G.I₃
  oct = Odd-Cycle-Transversal(tri,3)
  case oct is None then
    return None; # There's more then 3 vertices in the tripartite that should move.
  case oct = 1 then
    for v in tri.v do # O(n)
      inter = N̄(v) ∩ G.C
      if size(inter) > 2 then
        break loop;
      cl = G.C - inter + v
      bi = tri - v + inter
      if (cl.is_Clique() and bi.is_Bipartite()) then # O(m)
        return new 2-1-patition{
          C: cl
          I₁: bi.I₁,
          I₂: bi.I₂
        }
      end
    end
    return None; # There's no vertex that can be moved.
  case oct = 2 then
    return Transform2(tri,G.C,G)
  case oct = 3 then
    for v in oct do
      inter = N̄(v) ∩ G.C
      if size(inter) > 2 then
        break loop;
      cl = G.C - inter + v
      neoTri = tri - v + inter
      result = Transform2(neoTri,cl,G)
      if result is not None then
```

```
        return result;
    end
    return None;
end
```

5: Transform2

```
def Transform2(Tripartite,clique,G)
  P_1, P_2 = CriticalPaths(Tripartite)
  # Table
  # rank \ aux
  rank1 = Table[Label][Int][Int]
  for v in V(G[P_1]) do # Rank 1
    max_out = 0
    for out in E^+(v) ∈ G[P_1] do
      max_out = Max(out,max_out)
    end # max_out is the closest vertex to T that can be reached by an out edge
    previous = v.prev
    rank = v.position
    if previous is not None then
      # rank is either v position or the foremost
      # vertex that can be reached by the previous vertex
      rank = Max(rank, rank1[previous][1])
      # max_out is the max between the farthest
      # vertex that can be reached by v or by the previous vertex
      max_out = Max(max_out,rank1[previous][1])
    end
    rank1[v][0] = rank
    rank1[v][1] = max_out
  end
  rank2 = Table[Label][Int][Int]
  for u in V(G[P_2]) do # Rank 2
    max_out = 0
    for out in E^+(u) ∈ G[P_2] do
      max_out = Max(out,max_out)
    end # max_out is the closest vertex to T that can be reached by an out edge
    previous = u.prev
    rank = u.position
    if previous is not None then
      # rank is either u position or the foremost
      # vertex that can be reached by the previous vertex
      rank = Max(rank, rank2[previous][1])
      # max_out is the max between the farthest
```

```
    # vertex that can be reached by u or by the previous vertex
    max_out = Max(max_out,rank2[previous][1])
  end
  rank2[u][0] = rank
  rank2[u][1] = max_out
end
for v in $V(G[P_2])$ # Rank 1.2
  max_out = 0
  for out in $E^+(v) \in G[P_1 \cup v]$ do
    max_out = Max(out,max_out)
  end # max_out is the closest vertex to T that can be reached by an out edge
  previous = v.prev
  rank = v.position
  if previous is not None then
    # rank is either v position or the foremost
    # vertex that can be reached by the previous vertex in $P_1$
    rank = Max(rank, rank1[previous][1])
    # max_out is the max between the farthest
    # vertex that can be reached by v or by the previous vertex
    max_out = Max(max_out,rank1[previous][1])
  end
  rank1[v][0] = rank
  rank1[v][1] = max_out
end
for u in $V(G[P_1])$ # Rank 2.1
  max_out = 0
  for out in $E^+(u) \in G[P_2 \cup u]$ do
    max_out = Max(out,max_out)
  end # max_out is the closest vertex to T that can be reached by an out edge
  previous = u.prev
  rank = u.position
  if previous is not None then
    # rank is either u position or the foremost
    # vertex that can be reached by the previous vertex in $P_2$
    rank = Max(rank, rank2[previous][1])
    # max_out is the max between the farthest
    # vertex that can be reached by v or by the previous vertex
    max_out = Max(max_out,rank2[previous][1])
  end
  rank2[v][0] = rank
  rank2[v][1] = max_out
end
```

```
for z in V(clique) do # for all vertex in the clique
  # take all edges on the tripartite that have an end on z
  for e(z,i) in E(Tripartite ∩ N[z]) do
    z.left_u = +∞
    z.left_v = +∞
    z.right_u = 0
    z.right_v = 0
    # Find which are the closest and farthest vertex in P_1 z can reach
    if i in V(G[P_1]) then
      if i.position < z.left_u then
        z.left_u = i.position
      end
      if i.position > z.right_u then
        z.right_u = i.position
      end
    end
    # Find which are the closest and farthest vertex in P_2 z can reach
    if i in V(G[P_2]) then
      if i.position < z.left_v then
        z.left_v = i.position
      end
      if i.position > z.right_v then
        z.right_v = i.position
      end
    end
  end
end
for e(u,v) ∈ E(G) where u ∈ V(G[P_1]) and u ∈ V(G[P_1]) do
  # let N̄_k(v) be the not-neighborhood of a vertex in the clique
  S = N̄_k(u) ∩ N̄_k(v)
  if size(S) <=2 then
   if rank1[u][0] <= u.position then
     if rank2[v][0] <= v.position then
       if rank2[u][0] <= v.position and rank1[v][0] <= u.position then
         if size(S) = 0 then
           c = clique ∪ {u,v}
           bi = Bipartition_Of(Tripartite \ {u,v})
           return new 2-1-patition{
             C: c
             I_1: bi.I_1,
             I_2: bi.I_2
           }
```

```
            end
            max_right_u = 0
            max_right_v = 0
            min_left_u = +∞
            min_left_v = +∞
            c = clique ∪ {u, v} \ S
            bi = Bipartition_Of(Tripartite ∪ S \ {u, v})
            h = new 2-1-patition{
                C: c
                I_1: bi.I_1,
                I_2: bi.I_2
              }
            for z in V(S) do
             if z.right_u > max_right_u then
               max_right_u = z.right_u
             end
             if z.right_v > max_right_v then
               max_right_v = z.right_v
             end
             if z.left_v < min_left_v then
               min_left_v = z.left_v
             end
             if z.left_u < min_left_u then
               min_left_u = z.left_u
             end
            end
            if max_right_u < u.position and max_right_v < v.position then
              return h;
            end
            if min_left_u > u.position and min_left_v > v.position then
              return h;
            end
          end
        end
      end
     end
  end
  return None;
end
```

2.5.2 Correctness

Bibliography

1 BRANDSTÄDT, A. *Partitions of graphs into one or two independent sets and cliques.* 1984.

2 KÖNIG, D. *Theorie der endlichen und unendlichen Graphen: Kombinatorische Topologie der Streckenkomplexe.* [S.l.]: Akademische Verlagsgesellschaft mbh, 1936. v. 16.

3 GAREY, M. R.; JOHNSON, D. S. *Computers and intractability.* [S.l.]: wh freeman New York, 2002. v. 29.

4 BRANDSTÄDT, A. Partitions of graphs into one or two independent sets and cliques. *Discrete Mathematics*, Elsevier, v. 152, n. 1-3, p. 47–54, 1996.

5 REED, B.; SMITH, K.; VETTA, A. Finding odd cycle transversals. *Operations Research Letters*, Elsevier, v. 32, n. 4, p. 299–301, 2004.