Computing H-Joins with Application to 2-Modular Decomposition

Michel Habib, Antoine Mamcarz, Fabien de Montgolfier

Noname manuscript No.
(will be inserted by the editor)

Computing H-joins with application to 2-modular decomposition

Michel Habib · Antoine Mamcarz · Fabien de Montgolfier

the date of receipt and acceptance should be inserted later

Abstract We present here a general framework to design algorithms that compute H-join. For a given bipartite graph H, we say that a graph G admits a H-join decomposition or simply a H-join, if the vertices of G can be partitioned in $|H|$ parts connected as in H. This graph H is a kind of pattern, that we want to discover in G. This framework allows us to present fastest known algorithms for the computation of P_4-join (aka N-join), P_5-join (aka W-join), C_6-join (aka 6-join). We also generalize this method to find a *homogeneous pair* (also known as *2-module*), a pair $\{M_1, M_2\}$ such that for every vertex $x \notin (M_1 \cup M_2)$ and $i \in \{1, 2\}$, x is either adjacent to all vertices in M_i or to none of them. First used in the context of perfect graphs [Chvátal Sbihi 87], it is a generalization of *splits* (a.k.a 1-joins) and of *modules*. The algorithmics to compute them appears quite involved. In this paper, we describe an $O(mn^2)$-time algorithm computing all maximal homogeneous pairs of a graph, which not only improves a previous bound of $O(mn^3)$ for finding only one pair [Everett Klein Reed 97], but also uses a nice structural property of homogenous pairs, allowing to compute a canonical decomposition tree for sesquiprime graphs.[1]

Keywords algorithms · graph · graph decompositions · H-join · homogeneous pairs

1 Introduction

In the following, $G = (V, E)$ always denotes a simple, finite, undirected graph with $|V| = n$ vertices and $|E| = m$ edges. Two sets X and $Y \subseteq V$ are said to be **adjacent** (resp. non adjacent) if $\forall x \in X\ \forall y \in Y, xy \in E$ (resp. $xy \notin E$). A vertex x is said to be a **splitter** of $X \subset V$ if $\exists y, z \in X$ s.t. $xy \in E$ and $xz \notin E$.

LIAFA · CNRS · Université Paris Diderot - Paris 7
E-mail: {habib, mamcarz, fm}@liafa.univ-paris-diderot.fr

[1] A preliminary version of this work was presented at Latin 2012 [15].

Let H be a $l \times r$ matrix filled over $\{0,1\}$. We say that G admits a H-**join** if V can be partitioned into $l+r$ subsets $L_1...L_l$ and $R_1..R_r$ such that, for all $i \in [1..l]$ and all $j \in [1..r]$

- If $H[i,j] = 1$ then for all $v \in L_i$ and for all $v' \in R_j$ $vv' \in E$. In other words, there is a complete join (all possible edges exist) between L_i and R_j.
- Else ($H[i,j] = 0$) then for all $v \in L_i$ and for all $v' \in R_j$ $vv' \notin E$. In other words, there is no edge between L_i and R_j.

We note $L = \bigcup_{i=1}^{i=l} L_i$ the **left side** of the H-join, and $R = \bigcup_{j=1}^{j=r} R_j$ the **right side** of the H-join. But sometimes L and R are identified with $\{L_1...L_l\}$ and $\{R_1...R_r\}$. Note that there are no constraints on the adjacency of two vertices on the same side of the join. The problem we deal with is:

H-join Problem:
Input: A graph $G = (V,E)$, and H a $l \times r$ matrix filled over $\{0,1\}$.
Question: Does G admit a H-join?

This problem was first defined under this name in [4,20], and it is a particular case of Matrix Partition problems as defined in [**?**]. Notice that equivalently, a bipartite graph B_H may be used instead of H to encode this problem. It is defined in the obvious way: l white vertices $u_1...u_l$, r black vertices $v_1...v_r$, and edge u_iv_j iff $M_{i,j} = 1$.

For a given H, a H-join is **surjective** if no set R_i or L_j is empty. The left (resp right) side is trivial if $\forall i |L_i| \leq 1$ (resp. if $\forall i |R_i| \leq 1$). The join is **non-trivial** if neither side is trivial.

Furthermore H is **twin-free** if it has no two identical rows, nor two identical columns. Clearly, if two rows i and i' are identical, then vertices of G may be put indifferently in L_i or in $L_{i'}$. Using twins could be important in some application since this allows to formulate cardinality constraints, for the surjective case. In fact two identical rows i and i' have a cardinality constraint meaning: $|V_i \cup V_{i'}|$ must be at least 2. Section 2.8 explains how to deal with these twins.

This formalism can be used to express a broad variety of problems, among which we can find connectedness of a graph (H is made up with 2 isolated vertices), but also many classical graph decompositions such as modular decomposition [16]. Indeed, finding a module is equivalent to find a H-join in which H is reduced to an edge and an isolated vertex. Using the same formalism one can search for 1-joins (aka splits) [8] (H is reduced to one edge and 2 isolated vertices one on each side), or 2-join (H is reduced to 2 parallel edges and 2 isolated vertices one on each side). For these decompositions, efficient ad-hoc algorithms exist. But for other decompositions like P_4-join, P_5-join, C_6-join, (see Figures 1,2, 3), this is the only known method. These decompositions were also defined and used in [5].

In this paper, we first propose a general technique for computing non-trivial H-join in the surjective case and then we apply it to several particular

cases, including the detection of homogenous pairs. In section 4, focusing on this homogeneous pair decomposition we show how to compute a kind of tree decomposition for sesquiprime graphs using our techniques.

2 Algorithms for the surjective H-join problem

Let us now show, given G and H, how to compute efficiently a surjective, non-trivial H-join.

2.1 Seed sets

In the following we will refer to the sets L_i and R_j as **destination**.

Definition 1 Let H denotes an $l \times r$ matrix filled over $\{0,1\}$. A subset $S = \{l_1 \dots l_l, r_1 \dots r_r\} \subseteq V$ is called a *seed set*.

If for given seed set S and matrix H we have that, for all $i \in [1 \dots l]$ and $j \in [1 \dots r]$ $l_i r_j \in E \iff H[i,j] = 1$, then S is **compatible** with H.

If there exists a seed set S and a H-join $\{L_1, ...L_l, R_1, ..R_r\}$ such that for all $1 \leq i \leq l$ and $1 \leq j \leq r$ we have $l_i \in L_i$ and $r_j \in R_j$, then we say that S **realizes** H (or realizes this H-join) and that this H-join **extends** S.

It is easy to check if a seed-set is compatible with a matrix H. Clearly a seed-set realizing a H-join is compatible with H, but the converse is not true. Our brute force method to find H-decomposition is roughly the following:

1. **Enumerate all possible seed sets,**
2. **For every generated seed set check if it could be extended as a H-join.**

So there are two independent steps, the first one has to define how to efficiently generate all the possible seed sets and the second step is concerned by how to extend a seed set. Let us start by studying this first step.

2.2 Generating the seed sets

At first, it seems that we must consider $O(n^{l+r})$ seed sets. However, we can low down a little this number, using techniques inspired from [10].

Proposition 1 *If G admits a H-join, every spanning tree of G contains at least one edge of the H-join.*

Proof The special case with H filled with zeros can be trivially solved using any connected components algorithm. And if H contains at least one 1, it induces an edge cut of G, i.e. the join contains an edge of every path between vertices of L and vertices of R. □

A spanning tree of G contains $O(n)$ edges, including one that crosses the join. Therefore, by considering every edge of the spanning tree, and $l + r - 2$ vertices, we have:

Proposition 2 *It is enough to consider only $O(n^{l+r-1})$ seed sets.*

For some particular H this bound can be improved as it is shown in the next sections.

2.3 Computing a H-join that extends a given seed set

Let us first assume that H has no false twin (H is twin-free for short).

Definition 2 Given $x \in V \backslash S$, L_i is a **potential destination** of x if for every $R_j \in R$, for every $y \in R_j$ $xy \in E \iff H[i,j] = 1$. Dually R_j is a **potential destination** of x if for every $L_i \in L$, for every $y \in L_i$ $yx \in E \iff H[i,j] = 1$.

Given a seed set S, for every $x \in V \backslash S$, let $list(x) \subseteq \{L_1, \ldots, L_l, R_1, \ldots R_r\}$ denote the set of its potential destinations.

Proposition 3 *Given a seed set S, for every $x \in V \backslash S$ there are at most two sets to which x can belong. Moreover, at most one of these sets belongs to R, and at most one belongs to L.*

Proof Since H is twin-free, all neighborhoods in H are different and therefore for every vertex $x \in V \backslash S$ at most one set L (resp. R) is compatible with the adjacency to the vertices of S that belong to R (resp. L). □

In other words, starting from the seed set S, $list(x)$ has size at most 2. If it has size 0 for some x then that seed set cannot realize H. If it has size 1, x can safely be placed to the unique destination set it may belong to. It remains to address the case when $list(x)$ has size 2. In [13] the problem is solved using a 2-SAT solver [3]. Previous results, applied to homogeneous pairs, can be found in [12] using that idea.

Theorem 1 *[13] Given a seed set S, it is possible to build a 2-SAT instance that extends S to a H-join of G, if such partition exists, in $O(n^2)$ time.*

Although 2-SAT can be solved using a linear time algorithm, the n^2 term in the complexity comes from the size of the 2-SAT instance which has $O(n^2)$ clauses. Furthermore, enumerating all possible seed sets yields an algorithm in $O(n^{|S|+2})$ for computing a H-join.

Now, we will show that it is possible to use partition refinement techniques in order to extend a seed set to a H-join, faster than the 2-SAT method. Partition refinement is a standard technique for graph algorithm design, see for example [19,17].

2.4 Destination rules

Let us now give some **Forcing Rules** describing how a vertex can be placed to a destination L_i or R_j. Later we shall explain how to implement them using partition refinement techniques. We say a vertex $x \in V$ is *placed* if $|list(x)| = 1$ (this is especially the case when $x \in S$) and *ambiguous* if $|list(x)| = 2$

Proposition 4 *Let us consider a partial solution with placed vertices set into* $\{L_1 \dots L_l, R_1, \dots R_r\}$*, and with all other vertices of* V *ambiguous. Let* x *be a placed vertex, and* y *be an ambiguous vertex, such that* $list(y) = \{L_i, R_j\}$*,*

1. *If* $list(x) = \{L_{i'}\}$*,* $H[i', j] = 1$ *and* $xy \notin E$ *then there exists no solution with* y *placed to* R_j
2. *If* $list(x) = \{L_{i'}\}$*,* $H[i', j] = 0$ *and* $xy \in E$ *then there exists no solution with* y *placed to* R_j
3. *If* $list(x) = \{R_{j'}\}$*,* $H[i, j'] = 1$ *and* $xy \notin E$ *then there exists no solution with* y *placed to* L_i
4. *If* $list(x) = \{R_{j'}\}$*,* $H[i, j'] = 0$ *and* $xy \in E$ *then there exists no solution with* y *placed to* L_i

And if there is an H-join compatible with this partial solution, then there is also an H-join compatible with the partial solution in which x *has been placed to its destination.*

Proof There are eight configurations depending on whether x is on the left or right side, H has an entry 0 or 1, and xy is an edge or not. For four of these configurations, x adds no constraints to y. As for example the case in which: $list(x) = \{L_{i'}\}$, $H[i', j] = 1$ and $xy \in E$.

Only the four cases listed in the proposition force y to be placed. Let us consider the first case (the three other cases are similar): $x \in L_{i'}$, $H[i', j] = 1$ and $xy \notin E$. If $y \in R_j$ then the partial solution is not consistent with H, since $H[i', j] = 1$, $x \in L_{j'}$, and $xy \notin E$. So y necessarily belongs to L_i. □

So starting with a seed set S, we can first compute the lists, and then apply in any order these 4 forcing rules until no more can be applied. The partial solution grows, starting from S.

2.5 The extended partition refinement outline

Let us now consider how to efficiently compute the destinations of each vertex, using these forcing rules. There are two difficulties with them. First we must deal with edges but also with non edges, without having the complement graph. This can be handled by using *vertex splitting*, a standard *partition refinement* technique for graphs [19,17]. But sometimes we have to merge some classes. This is an extension of the standard partition refinement. Notice that termination is ensured because the merged classes are never split again, since they are exactly the destination sets, see below.

Let us recall how vertex splitting works. For any partition $\mathcal{P} = \{P_1, \dots P_k\}$ of the vertex set V and any $A \subseteq V$, called a *pivot-set*, we define : $Refine(\mathcal{P}, A) = \{P_1 \cap A, P_1 - A, \dots, P_k \cap A, P_k - S\}$. The action of a pivot-set over the partition $\mathcal{P}$ (called *refinement* of $\mathcal{P}$ using A) is to create a new partition $\mathcal{P}' = Refine(\mathcal{P}, A)$ where each part $P \in \mathcal{P}$ is replaced with the two parts $P \cap A$ and $P \setminus A$. However empty parts are not inserted in $\mathcal{P}'$. A part P is thus **splitted** only if $P \not\subset A$ and $P \cap A \neq \emptyset$. For various graph applications $A = N(x)$ for some *pivot-vertex* $x \in V$, and therefore the current partition is refined using some neighborhood.

Lemma 1 *[17] If a graph G is given by its adjacency lists, for every partition $\mathcal{P}$ of V, $Refine(\mathcal{P}, N(x))$ and $Refine(\mathcal{P}, \overline{N(x)})$ can be computed in $O(|N(x)|)$.*

So given a graph G, partition refinement techniques allow to compute dually (i.e., either on G or $\overline{G}$) within the same complexity. Before going into the details of the algorithm let us now sketch it, in 3 main steps:

1. **Given a seed set, compute the list of every vertex of V.**
   This can be obtained using standard partition refinement, starting from the initial trivial partition $\mathcal{P} = \{V\}$ using successively the neighborhood of every vertex of S as pivot. Then Procedure GENLISTS (see 2.6.1) assigns the lists to each part. See Section 2.6.1 for details. After this step we have a partition that is a partial solution, with nonempty destination sets containing all placed vertices. A part is a set of vertices with the same *list*.
2. **Apply all possible forcing rules.**
   To avoid generating trivial H-join, one more placed vertex is needed. If all vertices of $V \setminus S$ are ambiguous, the algorithm tries the two possible affectations of this given vertex v_0 to start the process. Note that if a solution exists, at least one of these affectations will extend into a non-trivial H-join.
   Finally Algorithm 2 implements Proposition 4 recursively, using a queue F which is the queue of the newly placed vertices that may help to affect other ambiguous vertices. Using F allows us to find the pivot vertices in constant time.
3. **Test whether the current partition can be extended as a solution.**
   When there are no more placed vertices that can be used as pivots, the process stops. Proposition 5 tells that all remaining ambiguous vertices can be put on one side of the join. We process using necessary conditions, so if the seed extends to a H-join, then the computed partition is a H-join. That must be finally checked. This test can be done in $O(n+m)$ time using a straightforward algorithm.

2.6 Algorithm details

Let us now detail the implementation. For our unconventional partition refinement we need, in addition to procedure refine(), some particular procedures.

The procedure move_vertex($\mathcal{P}$, x, P) extracts x from its current part of $\mathcal{P}$ and put it into part P. It returns the new partition. Procedure merge($\mathcal{P}$, P,Q) return a new partition in which $P \cup Q$ is a new part, replacing both P and Q.

Algorithm 1: EXTEND

```
   Data: (G, H) an instance of the H-join problem, S = {l_1 ... l_l, r_1 ... r_r} a seed set;
   Result: A H-join of G respecting S if such partition exists, ∅ otherwise;
 1 if S is not compatible with H then return ∅;
 2 foreach L_i ∈ L do L_i := {l_i} ;
 3 foreach R_j ∈ R do R_j := {r_j} ;
 4 P = {L_1, ... L_l, R_1, ... R_r, V \ S};
 5 foreach s ∈ S do
 6 |   P =refine(P, N(s)) ;
 7 Queue F := ∅;
 8 GENLISTS(P,S,H);
 9 foreach x ∈ V \ S do
10 |   if list(x) = ∅ then
11 |   |   return ∅;
12 |   else if list(x) = D_i then
13 |   |   P =move_vertex(P, x, D_i);
14 |   |   enqueue(F, x)
15 if F ≠ ∅ then
16 |   return(REFINEMENT((G, H),P, F));
17 else
18 |   pick any v_0 ∉ S;
19 |   Force v_0 inside L;
20 |   F = (v_0);
21 |   if REFINEMENT((G, H),P, F) ≠ ∅ then
22 |   |   return(REFINEMENT((G, H),P, F));
23 |   else
24 |   |   Force v inside R;
25 |   |   F = (v_0);
26 |   |   return(REFINEMENT((G, H),P, F));
```

2.6.1 Procedure GENLISTS

Procedure GENLISTS($\mathcal{P}$,S,H) computes $list(P)$ for each part $P \in \mathcal{P}$ created after the $|S|$ initial refinements. We can assume that each part P has a `neighbors` list and, if a new part $P \cap N(x)$ is created during refinement, (while the old P becomes $P \setminus N(x)$), then x is added to a `neighbors` list of the new part which inherits (in constant time using linked lists with a common tail, i.e., an in-tree) from the `neighbors` list of the former P part. The `neighbors` list of the parts allows to build the *list*s. Indeed if we use the pivots in order l_1 then l_2... then l_l then r_1... then r_r; and if the new part $P \cap N(x)$ is put just before (in a total ordering of parts) the old part $P \setminus N(x)$, then all parts are sorted in lexicographic order with respect to S. We can sort matrix H in the same order.

Algorithm 2: REFINEMENT

Data: (G, H) an instance of the H-join problem, $\mathcal{P}$ a partition with sets $L_1 \dots L_l, R_1 \dots R_r$ marked, F a non-empty queue of placed vertices;
Result: A H-join of G respecting S if such partition exists, $\varnothing$ otherwise;

```
while F ≠ ∅ do
    Pop x from F;
    P =refine(P,N(x));
    foreach Part P ∈ P do
        if P falls in one case of Proposition 4 then
            foreach y ∈ P do enqueue(F, y);
            merge P with L_i or R_j according to the case;
Let v_0 be the first vertex ever popped from F;
if v_0 ∈ R then
    put all ambiguous vertices to Side L of the join
else
    put all ambiguous vertices to Side R of the join;
if P is a H-join of G then
    return P;
else
    return ∅;
```

Therefore, for each part P, if it corresponds with one (and at most one since H is twin-free) R_j, then R_j is placed to $list(P)$, i.e. with this implementation, all vertices $x \in P$ implicitly get R_j to their *list*. This is done in one sweep with j from 1 to r.

The same is then done for the right side also (affect L_i to $list(P)$) after dropping $l_1 ... l_l$ from the `neighbors` lists and re-sorting them in lexicographic order (taking linear time).

2.7 Correctness of the EXTEND algorithm

Proposition 5 *In the execution of EXTEND algorithm, when F becomes empty,*

- *either all remaining ambiguous vertices can be put into R (right side)*
- *or into L*

Proof Let us prove it only for the R side (the proof for the other side is the same). Let P be a part of ambiguous vertices with $list(P) = \{L_i, R_j\}$. We affect all vertices from P to R_i. For all vertices currently in the left side, the constraints of H are respected (if not, some vertex would not have been ambiguous by proposition 3, and would have been inserted into F). Since we do not add further vertices to this side, so those constraints remain respected. □

Theorem 2 *The Refinement Algorithm 2, given an instance (G, H) and a seed S of $r+l$ vertices, computes a non-trivial H-join $\mathcal{P}$ that extends S if such H-join exists, and returns $\varnothing$ otherwise.*

Proof Correctness of the main loop follows from Proposition 4. If after step 2 there are no more ambiguous vertices, then the necessary conditions of Proposition 4 tell that $\mathcal{P}$ is the only H-join possible extending S (provided $\mathcal{P}$ is a genuine H-join, what is ultimately tested by the algorithm). In the other case, by Proposition 3, every ambiguous vertex has a list of size 2, with one destination for each side of the join. By Proposition 5, putting all ambiguous vertices on each side of the partition gives H-join extending S (again, if possible). To be sure that the H-join is not trivial, if v_0, the first pivot used after the vertices from S, is on the left side, then all ambiguous are put on the right side, and vice versa. □

2.8 Dealing with twins

Proposition 6 *Let (G, H) be an instance of the H-join problem, and S a seed set compatible with H. It is possible to build H' such that H' is twin-free, and that S realizes H if and only S is realizes H'.*

Proof Let H' be H in which every twin class has been contracted. In this case, S contains more than one vertex for each destination (in fact, one for each twin in the twin class).

Take any solution to the H' partition problem. Simply split the previously merged destinations. Since there are no constraints between those new sets, and since there is enough vertices in S to do so, this is always possible.

Conversely, consider any solution to the H-join problem. Assume D and D' are twins in H, just merge D and D' to obtain a solution to the H'-join problem. □

2.9 Complexity issues

Proposition 7 *$\mathcal{P}$ always contains at most $2r + 2l + rl$ parts*

Proof We create an initial partition with $|S| + 1$ parts but only one non-trivial part $V \setminus S$, and then perform $|S|$ pivoting on it. Theoretically this steps can create $O(|S| + 2^{|S|})$ parts. After Procedure GENLISTS($\mathcal{P}$,S,H) creates $lists(P)$ for each class P, two parts cannot have the same list since H is twin-free. As $list(P)$ cannot be empty either, then we have at most l parts with $lists(P) = \{L_i\}$; at most r parts with $lists(P) = \{R_j\}$; and at most lr parts with $lists(P) = \{L_i, R_j\}$. Plus the $|S|$ parts $L_1...L_l$ and $R_1...R_r$. All other parts have an empty $list$, thus if in the initialization step creates more than $2r + 2l + rl$, we can end with $list(P) = \varnothing$ and the algorithm stops.

When a refine is done by Algorithm 2 then Proposition 4 ensure that one of the two classes issued from a part P, either $P \cap N(x)$ or $P \setminus N(x)$, is merged with one of its destination set (L_i or R_j). Therefore the number of parts cannot increase and is bounded by $2r + 2l + rl$. □

Proposition 8 *Algorithm 1 requires $O(nlr + m)$ time.*

Proof Lemma 1 tells that refine$(\mathcal{P}, x)$ runs correctly and in $O(|N(x)|)$ time (the standard implementation is that vertices from $N(x)$ are moved to newly created parts $P \cap N(x)$, while the unmoved vertices stay in the old parts $P \setminus N(x)$ needing no operation). Since each neighborhood is used at most one time as a pivot, the time taken by all the calls to refine is $O(m)$.

By using lists to store every destination L_i or R_j, each merging step can be done in constant time, which implies that all merging will take $O(n)$ time (since every vertex is moved only one time). There are also $O(n)$ queue operations (each vertex is put at most once into F).

Procedure GENLISTS($\mathcal{P}$,S,H) takes $O(n + |S|)$ time since it uses linear-time lexicographic sorting, and simultaneous sweep of two sorted lists.

The only problem we may have is the line 2 of Algorithm 2. We have to check every part P to see if its vertices fall in one case of Proposition 4. This check is done in constant time per part, and Proposition 7 insures that the number of parts is always bounded by $2r + 2l + rl$. □

Let h denotes the number of ones in the H matrix.

Theorem 3 *There is a $O(n^{l+r-1}mlrh)$ time algorithm for H-join problem, i.e. that produces a surjective, non-trivial H-join if possible.*

Proof First assume G is connected. Generate the seed S by taking one edge among the $n-1$ of the spanning tree, and $n-2$ other vertices of G. Identify the edge with every 1 entry of matrix H (at most h of them). Then, use the refinement procedure presented above. And if G is not connected, all connected components except one shall be put in one destination set, so work component by component. □

Corollary 1 *If H is not considered to be a part of the input, there is a $O(n^{l+r-1}m)$-time algorithm to solve the H-Join problem.*

3 An almost-surjective case

3.1 Algorithms

The algorithm for the surjective case above may be extended to a restricted class of non-surjective case, when we allow two "trash sets" that may be empty. Each trash set R_1 or L_1 is non-adjacent with the other side of the join. This class includes almost all known H-join problem people are looking at.

More formally, given H, we say that G admits an *Almost Surjective H-join* (ASH-join) if G admits a H-join $\mathcal{P} = \{L_1 ... L_l, R_1 ... R_r\}$ such that

- For all $i \in 2 \dots l$ L_i contains at least one vertex.
- For all $i \in 2 \dots r$ R_i contains at least one vertex.
- R_1 is non adjacent to every L_i, i.e. $\forall i$ $H[i, 1] = 0$

- L_1 is non adjacent to every R_j, i.e. $\forall j\ H[1,j] = 0$
- $L = \cup L_i$ contains at least l vertices
- $R = \cup R_j$ contains at least r vertices.

Define an AS-seed set as a set $\{l_2...l_l, r_2...r_l\}$. The same definition of a *compatible* seed and a seed that *realizes* H apply. Then, the proofs and algorithms go on as in the surjective case. We only need to prove those two propositions:

Proposition 9 *Given a AS-seed set S, for every $x \in V \backslash S$ there are at most two sets to which x can belong to. Moreover, at most one of these sets belongs to R, and at most one belongs to L.*

Proof Notice that L_1 (R_1) is uniformly adjacent to R (L). Since H is twin-free, every pair of vertices of L (R) admits a splitter in $R_2 \dots R_r$ ($L_2 \dots L_l$), thus at most one vertex of L (R) can respect the adjacency to the vertices of S that belong to $R(L)$.□

Proposition 10 *If G admits a ASH-join, every spanning tree of G will contain at least one edge between to non-empty sets of the H-join.*

Proof Since R_1 and L_1 are non-adjacent to the other side of the join, no edge of the cut can be adjacent to them. All the remaining sets are non-empty. □

Finally we have:

Theorem 4 *There is a $O(n^{l+r-3} * lr * m)$ time algorithm for ASH-join problem.*

3.2 Applications

Let us now briefly describe three applications to known H-join problems for which our general framework provides the fastest known algorithms. Notice that our general procedure provides an algorithm in $O(nm)$ to find a 1-join, which is slower than state-of-the-art algorithms that provide the whole decomposition tree in linear time [8].

It may also be applied to 2-join which is an almost surjective H-join problem, and provides an algorithm in $O(n^3m)$ which, again, is slower than state-of-the-art $O(n^2m)$ algorithms [10].

3.2.1 P_4-join

A graph is said to admit a P_4-Join (or N-join) if its vertex set can be partitioned into six sets R_0, R_1, R_2, L_0, L_1, L_2 as depicted in Figure 1 (an edge L_iR_j appears if $H[i,j] = 1$, i.e L_i must be completely adjacent with R_j, and no edge is drawn if $H[i,j] = 0$, i.e no edge between L_i and R_j is allowed) [5].

Corollary 2 *There is an $O(mn^3)$ time algorithm to compute a non-trivial P_4-join if exists any.*

Proof This decomposition is quasi-surjective and we can apply theorem 4, which yields an $O(mn^3)$ time algorithm. □

Fig. 1 Structure of a P_4-Join

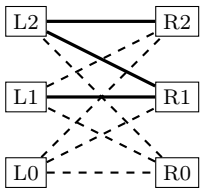
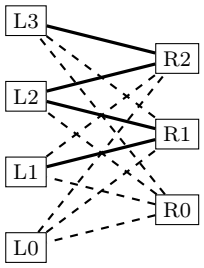


Fig. 2 Structure of a P_5-Join



3.2.2 P_5-join

A graph is said to admit a P_5-Join (or W-join) if its vertex set can be partitioned into seven sets R_0, R_1, R_2, L_0, L_1, L_2, L_3 as depicted in Figure 2 (an edge L_iR_j appears if $H[i,j] = 1$, i.e L_i must be completely adjacent with R_j, and no edge is drawn if $H[i,j] = 0$, i.e no edge between L_i and R_j is allowed) [5]. It is a particular case of quasi-surjective H-join and therefore:

Corollary 3 *There is a $O(mn^4)$ time algorithm to compute a non-trivial P_5-join, if exists any.*

3.2.3 C_6-join

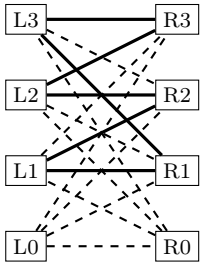
The C_6-join was introduced by [2,6] under the name of 6-join, to decompose some perfect graphs. A graph is said to admit a C_6-Join if its vertex set can be partitioned into 8 sets $L_0...L_3$ and $R_0...R_3$ as depicted in Figure 3. It is a particular case of quasi-surjective H-join and therefore:

Corollary 4 *There is a $O(mn^5)$ time algorithm to compute a non-trivial C_6-join of a graph, if exists any.*

4 Homogeneous Pairs

First used in the context of perfect graphs [11], homogeneous pairs generalize *splits* (a.k.a 1-joins) and *modules*. The algorithmics to compute them appears

Fig. 3 Structure of a C_6-Join



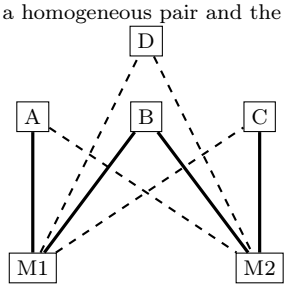
quite involved. In the following, we describe an $O(mn^2)$-time algorithm computing (if any) a homogeneous pair, which not only improves a previous bound of $O(mn^3)$ [12], but also uses a nice structural property of homogenous pairs.

4.1 Definition

A **homogeneous pair** of G is a set $P \subseteq V$ which can be partitioned into two subsets M_1 and M_2 such that neither M_1 nor M_2 has a splitter outside P (i.e. all splitters of M_i belong to M_{3-i}).

If $2 < |P| < |V|-1$, P is called a **proper** (or non trivial) homogeneous pair. The definition implies that, if such a proper pair exists, the whole vertex-set is partitioned into six sets M_1, M_2, A, B, C, D, such that $|A|+|B|+|C|+|D| > 2$ and $|M_1|+|M_2| > 2$ and there are 4 adjacencies (i.e., complete bipartite, drawn as plain lines) and four non-adjacencies (dotted lines) between the sets as depicted in Figure 4 (no edge is drawn if any linkage between sets is allowed). In the following, we will sometime refer to vertices or subsets of P as **inner**, and to vertices or subsets of $V \backslash P$ as **outer**. Clearly, the inner and outer sides are the two sides of a H-join.

Fig. 4 Relationships between a homogeneous pair and the other vertices



Even though homogeneous pair detection is not a ASH-join problem, we can generalize a bit further our method to solve it.

Let $\overline{G}$ denote the edge-complement of G. Immediately from the above definition, by swapping A and C, and also B and D, we have:

Proposition 11 *P is a homogeneous pair of G iff P is a homogeneous pair of $\overline{G}$.*

Remark 1 A homogeneous pair with $M_1 = \varnothing$ or $M_2 = \varnothing$ or $A = C = \varnothing$ is a *homogeneous set* (also called *module*).

We say a module is *trivial* if it has 0,1 or n vertices. A graph is *prime* if all its modules are trivial ones.

Homogeneous pairs were used in [11] in the context of perfect graphs. They form a natural generalization of *splits* (in this case M_1 has no neighbors in $V - (M_1 \cup M_2)$) and of *modules* (in this case M_1 is empty). Linear-time algorithms for computing the split decomposition [8] or the modular decomposition (see [16] for a survey) are known, but are quite involved. An $O(mn^3)$ algorithm was proposed in [12] and it is therefore a challenging problem to design an efficient algorithm to find homogeneous pairs. Modules of graphs can be represented via a directed tree, using the *partitive families* framework, see [9], while splits can be represented via an undirected tree through *bipartitives families* [7,18]. Unfortunately the structure of homogeneous pairs seems to be weaker than the one of splits. Nevertheless, we can prove a tree structure theorem.

Definition 3 A *sesqimodule* (introduced in [18]) is a set M such that there exists x such that $G[V \setminus \{x\}]$ is a module. It is *proper* if $|M| > 1$.

A graph is *sesquiprime* if it contains no sesquimodule.

The family of sesquimodules is thus decomposed into modules (if M is a module any x is OK) and homogeneous pairs $\{M, \{x\}\}$ where x is the only splitter of M. Notice that in this case M is also a homogeneous pair $\{M \cap N(x), M \setminus N(x)\}$. So sesquimodules are special cases of homogeneous pairs, so named because they stand between the usual 1-modules and the 2-modules.

A sesquiprime graph is also prime (it contains no proper module).

Proposition 12 *All the sesquimodules of a graph can be enumerated in time $O(nm)$.*

Proof For each $x \in V$ just run a linear-time modular decomposition algorithm on $G[V \setminus \{x\}]$.□

Since when the graphs contains a sesquimodule, it is easy to find a homogeneous pair, we will only address the case of sesquiprime graphs.

4.2 Structural properties of homogeneous pairs

Lemma 2 *Let G be a sesquiprime graph and P be a proper homogeneous pair of G. There exists only one way to partition P into $\{M_1, M_2\}$.*

Proof Consider a partition $\{M_1, M_2\}$ of P with $m_1 \in M_1$ ang $m_2 \in M_2$. Assume, by contradiction, that it exists another split $\{M_1', M_2'\}$ of P with m_1 and m_2 in the same subset (w.l.o.g., M_1'). By definition of a homogeneous pair, m_1 and m_2 have the same neighborhood outside P, which implies that A and C were empty for the partition $\{M_1, M_2\}$, i.e., that P is a homogeneous set, contradicting primality of G.□

The graphs we consider in the following are assumed to be sesquiprime, unless explicitly stated otherwise. Among other interesting properties discussed below, we gain then that it is equivalent to talk about P or about $\{M_1, M_2\}$ when dealing with a homogeneous pair. And that is why P, a single subset, may be called a "pair".

A proper homogeneous pair P is *maximal* (resp. minimal) if there is no proper homogeneous pair Q such that $P \subsetneq Q$ (resp. $Q \subsetneq P$).

Remark 2 A homogeneous pair of a graph that induces a minimal set of inner vertices (resp. outer) vertices induces a maximal set of outer (resp. inner) vertices.

Let us now consider the relationships between homogeneous pairs.

Theorem 5 *Let G be a sesquiprime graph, $P = \{M_1, M_2\}$, and $P' = \{M_1', M_2'\}$ be two proper homogeneous pairs of G. Then*

1. *either $P \cap P' = \varnothing$*
2. *or $P \cap P' = \{x\}$ and x is not a splitter of $M_1 \setminus \{x\}$ nor $M_2 \setminus \{x\}$ nor $M_1' \setminus \{x\}$ nor $M_2' \setminus \{x\}$ (i.e. both $P - \{x\}$ and $P' - \{x\}$ are homogeneous pairs).*
3. *or $P \cup P'$ is a homogeneous pair*

Proof Consider the four intersection sets $M_i \cap M_j'$ for $i, j = 1, 2$. We distinguish four cases.

1. All these four intersection sets are empty. Then we are in Case 1.
2. Exactly three intersection sets are empty. Suppose without loss of generality M_1 properly intersects M_1'. We first prove $M_1 \cap M_1'$ is a module. Let y be a splitter of $M_1 \cap M_1'$. If $y \notin P$, y would split M_1 but P is a homogeneous pair, a contradiction. If $y \notin P'$, y would split M_1' but P' is a homogeneous pair, a contradiction. So $y \in P \cap P'$ but $P \cap P' = M_1 \cap M_1'$: y is not a splitter, final contradiction. As G is prime $M_1 \cap M_1'$ is trivial, i.e. $M_1 \cap M_1' = \{x\}$: this is Case 2 of the theorem. As G is sesquiprime then M_1 also contains $u \neq x$ and M_1' also contains $u' \neq x$.
   Suppose xu is an edge. Then uu' is an edge (otherwise u is a splitter of M_1') and then $u'x$ is an edge (otherwise u is a splitter of M_1). Finally x is adjacent with every vertex of $M_1 \cup M_1'$, and is not a splitter of $M_1 - \{x\}$ nor of $M_1' - \{x\}$. And if xu is not an edge, x is non-adjacent to $M_1 \cup M_1'$ and is not a splitter either.
   Suppose x is a splitter of M_2: there exists $y, z \in M_2$ and xy is an edge but not xz. Since y does not split P' yu' is an edge, while zu' is not an

edge for the same reason. So $u' \notin P$ is a splitter of M_2, contradiction with P is a homogeneous pair. For the same reason x is not a splitter of M'_2. Consequently, if $|P| > 3$ and $|P'| > 3$, then $P - \{x\}$ and $P' - \{x\}$ are homogeneous pairs.

3. Suppose now exactly two intersection sets are non empty. We must distingish subcases:
   (a) Suppose M_1 intersects M'_1, and M_2 intersects M'_2. Then $P \cup P'$ is a homogeneous pair $\{M_1 \cup M'_1, M_2 \cup M'_2\}$. Indeed for $u_1 \in M_1 \cap M'_1$ and $x \notin P \cup P'$, if xu_1 is an edge x is adjacent with both M_1 and M'_1 (otherwise it would split M_1 or M'_1) and thus does not split $M_1 \cup M'_1$. And if xu_1 is not an edge then x is non-adjacent with $M_1 \cup M'_1$. This is Case 3 of the theorem. A similar proof shows x is not a splitter of $M_2 \cup M'_2$.
   (b) Suppose M_1 intersects M'_2, and M_2 intersects M'_1. Then $P \cup P'$ is a homogeneous pair $\{M_1 \cup M'_2, M_2 \cup M'_1\}$ (same proof than 3(a))
   (c) Suppose M_1 intersects both M'_1 and M'_2, but M_2 does not intersect P'. Then $P \cup P'$ is a homogeneous pair $\{M_1 \cup P', M_2\}$. Indeed let $u_1 \in M_1 \cap M'_1$ and $u_2 \in M_1 \cap M'_2$. Let $x \notin P \cup P'$. Suppose xu_1 is an edge. As x is not a splitter of M_1 x is adjacent with M_1. So xu_2 is an edge. As x is not a splitter of M'_1 (resp. M'_2) x is adjacent with M'_1 (resp. M'_2). As similarly if xu_1 is not an edge x is non-adjacent with $M_1 \cup P'$. This is also Case 3 of the theorem.
   (d) The three remaining cases are similar with 3(c) if M_1 is replaced by M_2, or if we swap P and P', or both.
4. Suppose now at least three intersection sets are nonempty. Then either Case 3(a) or Case 3(b) holds. Suppose w.l.o.g M_1 intersects M'_1, and M_2 intersects M'_2. As seen before $\{M_1 \cup M'_1, M_2 \cup M'_2\}$ is a homogeneous pair. But a third intersection holds! If M_1 intersects also M'_2 then a neighbor (resp. non-neighbor) of $M_1 \cup M'_1$ also is a neighbor (resp. non-neighbor) of $M_2 \cup M'_2$ and $P \cup P'$ is a module. As G is sesquiprime, $P \cup P' = V$. Same if M'_1 intersects also M'_2. It is a degenerated form of Case 3 of the theorem.

□

4.3 Homogeneous pairs decomposition

The modules of a graph are *closed under union*: for them only Cases 1 or 3 of Theorem 1 may occur. But Theorem 5 tells the homogeneous pairs are *almost* closed under union, in the following sense. Let us say that a vertex x is a *conflicting vertex* of a homogeneous pair P if there exist a homogeneous pair P' such that $P \cap P' = \{x\}$.

Proposition 13 *A proper homogeneous pair P of a sesquiprime graph has size at least 4 and contains at most two conflicting vertices.*

Proof Let $P = \{M_1, M_2\}$. If $|M_1| = 0$ M_2 is a module, contradicting primality, thus sesquiprimality, of G. And if $|M_1| = 1$ P is a sesquimodule, also impossible. Same for M_2. So P has at least four vertices.

As a consequence of Case 2 of Theorem 5, a conflicting vertex x is either fully adjacent with M_i or fully non-adjacent with M_i, for $i = 1$ or 2. Furthermore x belongs either to M_1 or to M_2. There are thus eight cases. All vertices falling in the same case form a module, and since the graph is prime there is at most one vertex per case. Suppose M_1 contains two conflicting vertices x and y. If $xy \in E$ both x and y must be fully adjacent with M_1. And if $xy \notin E$ both x and y must be fully non-adjacent with M_1. Since there are no twins in G, x has to be fully adjacent with M_2, and y fully non-adjacent with M_2. So M_2 does not contains a conflicting vertex, and M_1 can not contain a third conflicting vertex (it would be a twin of x or y). The same proof shows that if M_2 contains two conflicting vertices then there are exactly two in M_2 and zero in M_1.

Definition 4 A vertex is a *maximal conflicting vertex* if it is the conflicting vertex of two maximal (wrt inclusion) homogeneous pairs of G.

A *almost maximal* homogeneous pair (AMHP for short) of a graph G is obtained by taking a homogeneous pair of G maximal with respect to inclusion, ad removing its (at most two) conflicting vertices.

Theorem 6
- *Every vertex of G either belongs to a unique AMHP, or is a maximal conflicting vertex.*
- *Each proper homogeneous pair P of G intersects exactly one almost maximal homogeneous pair $Max(P)$.*
- *Furthermore a proper homogeneous pair P is included in exactly one maximal homogeneous pair, namely the $Max(P)$ plus its (at most two) conflicting vertices.*

Proof Consequence of Theorem 5 and of Proposition 13.□

This theorem allows to compute a canonical (i.e., independent from algorithms choices) homogeneous pair decomposition tree of a sesquiprime graphs. Let the *almost maximal homogeneous pairs partition* (AMHPP for short) of a graph the partition of the vertex-set into almost-maximal homogeneous pairs and into maximal conflicting vertices. The existence and uniqueness of this partition is ensured by the previous theorem. A tree may then be built by recursively computing the AMHPP inside the graph induced by each AMHP. The next section shows how to compute this AMHPP efficiently.

5 Homogeneous pairs algorithm

5.1 Finding a Homogeneous pair

Definition 5 A HP-seed-set is a triple $\{m_1, m_2, a\}$ of vertices of V such that $m_1 \in M_1$, $m_2 \in M_2$, and $a \in A$.

The algorithm proposed here consists in first computing the modular decomposition of the graph, using existing linear time algorithms (see [16]). So we know if the graph contains a non-trivial module (which is a homogeneous set). If this preliminary search fails, we are lead to the case of searching proper homogeneous pairs in sesquiprime graphs. We thus assume now the graph is sesquiprime. Notice that $O(n+m) = O(m)$ as G is connected.

Thanks to Remark 1 we have to search for a homogeneous pair P that is not a module, i.e. such that $|M_1| > 0$, and $|M_2| > 0$, and $|A| + |C| > 0$. We can assume, without loss of generality that $|A| > 0$.

Proposition 14 *Given $S = \{m_1, m_2, a\}$ a HP-seed set, every vertex $x \in V \backslash S$ has $|list(x)| = 2$.*

Proof If $x \in N(a)$ x may belong to M_1 but not to M_2, and otherwise to M_2 but not to M_1. Whether x is adjacent with m_1, or m_2, or both m_1 and m_2, or neither of them, then it may only belongs to (respectively) A, C, B and D.□

Given that, we can use our refinement method.

Lemma 3 *If G is prime, then in every proper homogeneous pair $\{M_1, M_2\}$ there exists a non-edge $\{x, y\} \notin E$ with $x \in M_1$ and $y \in M_2$*

Proof Assume by contradiction that no such edge exists. M_2 would then be adjacent to M_1, thus both M_1 and M_2 would be modules of G, contradicting either the assumption of the primality of G or the properness of $\{M_1, M_2\}$.□

Proposition 15 *Consider a vertex $z \in M_2$ such that $M_1 \cap \overline{N(z)} \neq \varnothing$. Its non-neighborhood contains vertices from these four sets: $M_1' = \overline{N(z)} \cap M_1$; $M_2' = \overline{N(z)} \cap M_2$; A; and D*

Let us take such a vertex z. A *spanning forest* of $G[\overline{N(z)}]$ (obtained for example by a breadth first search) will necessarily contain one edge between a vertex of A and a vertex of M_1', since it is the only way to go from $M_1' \cup M_2'$ to $A \cup D$. Indeed, $B \notin \overline{N(z)}$ and $C \notin \overline{N(z)}$, whereas M_1' and A both contain at least one vertex (by Lemma 3 and assumption). Such spanning forest contains $O(n)$ edges, including at least one that will have one endpoint in M_1 and one endpoint in A. From the above discussion, we have:

Proposition 16 *It is possible to generate $O(n^2)$ HP-seed sets for the homogeneous pair problem, such that every proper homogeneous pair extends at least one of them.*

This is implemented by Algorithm 3, who generates $O(n^2)$ triples to test, and runs in $O(mn)$ time.

Then we have, using n^2 times the EXTEND algorithm (where H is the 2x4 matrix defining homogeneous pairs):

Theorem 7 *There is a $O(mn^2)$ time algorithm for the homogeneous pair problem.*

Algorithm 3: generate seeds

```
   Data: a sesquiprime graph G = (V, E);
   Result: S ⊂ V^3;
1  S := ∅;
2  foreach x ∈ V do
3  |  Let T be a BFS tree of G[N̄(x)];
4  |  foreach {y, z} ∈ E(T) do
5  |  |  S:=S ∪ {(x, y, z), (x, z, y)};
6  return S;
```

5.2 Finding a Maximal Homogeneous pair

Given two homogeneous pairs $P = \{M_1, M_2\}$ and $P' = \{M'_1, M'_2\}$ we note $P \subset P'$ for $(M_1 \cup M_2) \subset (M'_1 \cup M'_2)$.

Lemma 4 *Let $P = \{M_1, M_2\}$ and $P' = \{M'_1, M'_2\}$ be two homogeneous pairs with $P \subset P'$.*

1. *If $(M_1 \cup M_2) \subset M'_1$, then $(M_1 \cup M_2 \cup A \cup C) \subseteq P'$.*
2. *Else, $M_1 \subseteq M'_1$ and $M_2 \subseteq M'_2$; or $M_1 \subseteq M'_2$ and $M_2 \subseteq M'_1$.*

Proof In the first case, since $P \subset M'_1$, A and C are splitters of M'_1. So by definition of a homogeneous pair, we have $A \in P'$, and $C \in P'$. If we are not in the first case, then we cannot have M_1 intersecting both M'_1 and M'_2; nor M_2 intersecting both M'_1 and M'_2, because then A' or C' would split M_1 or M_2. So we are in the second case.□

Lemma 5 *Given a non-trivial homogeneous pair $P_i = \{{M_1}^i, {M_2}^i\}$, if there exists a non-trivial homogeneous pair $P_{i+1} = \{M_1^{i+1}, M_2^{i+1}\}$ and $(M_1^i \cup M_2^i) \subset M_1^{i+1}$, then P_{i+1} can be found in $O(mn)$ time.*

Proof We are in the first case of Lemma 4. Create a set M_1^{i+1} and affect to it all $M_1^i \cup M_2^i \cup A^i \cup C^i$ vertices. For each vertex $a \in N(M_1^{i+1})$ try launching the REFINEMENT procedure using $A^{i+1} = \{a\}$ and M_2^{i+1} as the splitters of M_1^{i+1} that also belong to $\overline{N(a)}$. Indeed neighborhood of a allows to split P_{i+1}. greedily adding the splitters of $P \cup A \cup C$ ends in finding some vertex of M'_2.□

Lemma 6 *Given $x \in M_1$, $y \in M_2$, $z \in A$, and t any other outer vertex as only member of F, procedure REFINEMENT returns P, a maximal homogeneous pair such that $x \in M_1$, $y \in M_2$, $z \in A$, and $t \in V \backslash P$.*

Proof According to Proposition 14, all vertices are ambiguous, except t, the only vertex from Queue F. Proposition 4 rules only affect vertices to the side of the placed vertices used, so all placed vertices are on the side of t. Furthermore, the algorithm adds only necessary vertices to A, B, C, D (by proposition 4). Then all remaining placed vertices are put on side $M_1 \cup M_2$.□

Lemma 7 *Given a non-trivial homogeneous pair $P = \{M_1, M_2\}$, such that there is no non-trivial pair $P' = \{M_1', M_2'\}$ with $P \subset M_1'$ or $P \subset M_2'$, then a maximal non-trivial (with respect to inclusion) homogeneous pair P'' can be found in $O(mn^2)$ time.*

Proof For any homogeneous pair P' containing P, we are in the second case of Lemma 4. Try every pair $\{z, t\}$ of vertices of $V \backslash P$ as outer vertices, and launch the REFINEMENT procedure. One of the $O(n^2)$ found homogeneous pairs will be maximal, especially the larger one P'', thanks to Lemma 6. This step takes $O(mn^2)$ time as there are $O(n^2)$ calls to REFINEMENT.□

Theorem 8 *A maximal non-trivial homogeneous pair of G can be found in $O(mn^2)$ time*

Proof We show it by constructing an sequence of homogeneous pairs $P_0 ... P_k$ ordered by inclusion.

P_0 is built using Algorithm 1 with any seed $\{x \in M_1, y \in M_2, z \in A\}$ returning a non-trivial result. Lemma 6 (using as t the v_0 of Algorithm 1) tells that at least one of the $O(n^2)$ tries of Proposition 16 should work, or the graph contains no non-trivial homogeneous pair.

Using Lemma 5 recursively, we get a sequence non-trivial of homogeneous pairs $P_i = \{{M_1}^i, {M_2}^i\}$, $0 \leq i < k$, with $P_i \subsetneq M_1^{i+1}$. Notice $k \leq n$ since inclusion is strict so total time is $O(mn^2)$.

When this is no more possible, we complete this sequence with a final non-trivial pair P_k found using Lemma 7, which is guaranteed by this Lemma to be maximal. Notice $k \leq n$ since inclusion is strict.□

5.3 Computing the Almost Maximal Homogeneous Pair Partition of a graph

Lemma 8 *Given a family of $O(n^2)$ homogeneous pairs containing all maximal homogeneous pairs, Algorithm 4 computes the AMHPP in $O(n^3)$ time.*

Proof It is a simple implementation of Theorem 6. The array $mark[\,]$ contains for each vertex either the number of its AMHP or c if it is maximal conflicting. Sorting the homogeneous pairs in decreasing size order gives a linear extension of the inclusion order, therefore a maximal homogeneous pair is seen before all the ones it contains. At most two vertices of this pair then belong to already seen homogeneous pairs and they are marked as maximal conflicting. The running time is $O(n^3)$ (in fact it is linear w.r.t. input size).

Theorem 9 *It is possible to find all almost maximal homogeneous pairs of a sesquiprime graph (i.e. the AMHPP) in time $O(mn^2)$*

Proof Let P be a maximal homogeneous pair found using Theorem 8. Let $\mathcal{S}$ be the set of $O(n^2)$ triples generated by Algorithm 3. Define $\mathcal{S}^{-P}$ as the subset of triples of $\mathcal{S}$ so that at least one of their inner vertices does not belong to

Algorithm 4: almost maximal homogeneous pairs partition

```
Data: a family F of O(n^2) homogeneous pairs containing all maximal homogeneous
      pairs
Result: The partition into AMHP and into maximal conflicting vertices
1  Let mark[1..|V|] be an array initialised with 0s.;
2  num ← 0;
3  Sort F in decreasing size order;
4  foreach homogeneous pair P ∈ F taken by decreasing size order do
5      shared ← 0;
6      foreach vertex x ∈ P do
7          if mark[x] ≠ 0 then shared++;
8      if shared ≤ 2 then
9          num++;
10         Remark: If shared ≤ 2 then P is the maximal homogeneous pair
                   number num. Otherwise P would be included and shared = |P|;
11         foreach vertex x ∈ P do
12             if mark[x] ≠ 0 then mark[x] = c else mark[x] = num;
13 foreach vertex x with mark[x] = 0 do
14     mark[x] ← num++;
15 return mark[ ]
```

P, and so that they are compatible with at least one non-trivial homogeneous pair.

Theorem 5 tells that, for another maximal homogeneous pair P', it intersects P on at most one vertex.

So for any two vertices v_0 and v_1 of P we are sure one of them is out of the intersection, i.e. is in $V \setminus P'$.

Thus, by calling REFINEMENT twice for any triple $(x, y, z) \in \mathcal{S}^{-P}$, once with $F = \{v_0\}$, and the second time with $F = \{v_1\}$, according to Proposition 6 we get a family $\mathcal{F}$ of $O(n^2)$ non-trivial homogeneous pairs excluding P in $O(mn^2)$ time, each one being maximal with respect to its seed and to v_0 (or its seed and v_1). $\mathcal{F} \cup \{P\}$ therefore contains all maximal non-trivial homogeneous pairs of the graph.

Some of our homogeneous pairs might still not be maximal, but they will be properly included into one another. Thus, we only need to scan our homogeneous pairs again, deleting the ones that are included into others homogeneous pairs. This can also be done in $O(n^3)$ time using Algorithm 4, and we get the AMHPP $\mathcal{M}$ of almost maximal homogeneous pairs of G.□

6 Conclusions and perspectives

The algorithms described above share some common features and they are all particular cases of graph partition problems that can be expressed using matrices in $\{0, 1, *\}$ as defined in [14]. For many polynomial cases the techniques developed here can be used to speed up the existing algorithms. It is possible

to patch the above algorithms to output possibly trivial H-joins, or to solve the list version of the problem we consider.

In this paper for the computation of non trivial H-join, we have carefully studied how to extend a seed set and our proposal is quasi optimal, but the other step : enumerating all potential seed sets, is far from being optimal. This could be improved, at least for some particular classes of graphs H, or by finding a better way to enumerate all the seed sets.

The structure of homogeneous pairs in sesquiprime graphs is now more undersood, leading to a canonical decomposition theorem. It is not clear however when the graph contains modules or sesquimodules.

7 Aknowledgments

The authors whish to thank the referees for their very careful readings, their suggestions greatly improve our writing.

References

1. P. Aboulker, P. Charbit, M. Chudnovsky, N. Trotignon and K. Vušković, *LexBFS, structure and algorithms*, CoRR, arxiv abs/1205.2535 (2012).
2. C. Aossey and K. Vušković, $3PC(.,.)$*-free Berge graphs are perfect*, Cited by Cornuéjols but seems unpublished.
3. B. Aspvall, M. F. Plass and R. E. Tarjan, *A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas*, Inf. Process. Lett., 8(3):121-123, 1979.
4. B-M. Bui-Xuan, J. A. Telle and M. Vatshelle, *H-join decomposable graphs and algorithms with runtime single exponential in rankwidth*, Discrete Applied Mathematics, 158(7): 809-819, 2010.
5. B-M.Bui-Xuan, J.A.Telle and M.Vatshelle, *Boolean-width of graphs*, Theor. Comput. Science, 412(39): 5187-5204, 2011.
6. M. Conforti, G. Cornuéjols, A. Kapoor, and K. Vušković, *Balanced* $0, \pm 1$ *matrices, Part I: Decomposition theorem, and Part II: Recognition algorithm*, Journal of Combinatorial Theory B 81 (2001), 243-306
7. W. H. Cunningham and J. Edmonds, *A combinatorial decomposition theory*, Canad. J. Math, vol 32(3): 734-765, 1980.
8. P. Charbit, F. de Montgolfier, M. Raffinot, *A Simple Linear Time Split Decomposition Algorithm of Undirected Graphs*, to appear in SIAM J. of Discrete Mathematics.
9. M. Chein, M. Habib and M.C. Maurer, *Partitive hypergraphs*. Discrete Mathematics 37(1): 35-50, 1981.
10. P. Charbit, M. Habib, N. Trotignon, and K. Vuskovic: Detecting 2-joins faster `http://arxiv.org/abs/1107.3977`, to appear in J. of Discrete Algorithms.
11. V. Chvátal and N. Sbihi, *Bull-free graphs are perfect*, Graphs Combin. 3: 127-139, 1987.
12. H. Everett, S. Klein and B. Reed, *An algorithm for finding homogeneous pairs*, Discrete Applied Mathematics 72: 209-218, 1997.
13. T. Feder, P. Hell, D. Král and J. Sgallx, *Two Algorithms for General List Matrix Partitions*, SODA (2005) 870-876.
14. T. Feder, P. Hell, S. Klein and R. Motwani, *List Partitions*, SIAM J. Discrete Mathematics 16, (2003) 61-80.
15. M. Habib, A. Mamcarz and F. de Montgolfier, *Algorithms for some H-join decompositions*, LATIN, LNCS N 7256, 446-457, 2012.
16. M. Habib and C. Paul, *A survey of the algorithmic aspects of modular decomposition*, Computer Science Review, 4: 41-59, 2010.

17. M. Habib, C. Paul, and L. Viennot. Partition refinement techniques: an interesting algorithmic toolkit. International Journal of Foundations of Computer Science 10 (2): 147-170, 1999.
18. F. de Montgolfier, *Décomposition modulaire des graphes: théorie, extensions et algorithmes*, PhD. thesis, Université Montpellier 2, 2003.
19. R. Paige and R. E. Tarjan, *Three Partition Refinement Algorithms*, SIAM J. Computing 16: 973-989, 1987.
20. M. Rao, *Décompositions de graphes et algorithmes efficaces* PhD. thesis, Université de Metz.