Finding and listing induced paths and cycles

Chính T. Hoàng* Marcin Kamiński† Joe Sawada‡ R. Sritharan §

January 13, 2012

Abstract

Many recognition problems for special classes of graphs and cycles can be reduced to finding and listing induced paths and cycles in a graph. We design algorithms to list all P_3's in $O(m^{1.5} + p_3(G))$ time, and for $k \geq 4$ all P_k's in $O(n^{k-1}+p_k(G)+k\cdot c_k(G))$ time, where $p_k(G)$, respectively, $c_k(G)$, are the number of P_k's, respectively, C_k's, of a graph G. We also provide an algorithm to find a P_k, $k \geq 5$, in time $O(k!! \cdot m^{(k-1)/2})$ if k is odd, and $O(k!! \cdot nm^{(k/2)-1})$ if k is even. As applications of our findings, we give algorithms to recognize quasi-triangulated graphs and brittle graphs. Our algorithms' time bounds are incomparable with previously known algorithms.

1 Introduction

Many recognition problems for special classes of graphs can be reduced to finding and listing induced paths and cycles in a graph. For example, if we can efficiently list all P_3's and their complements of a graph, then we can recognize quasi-triangulated graphs efficiently (definitions are given in Section 2). Also, recognizing brittle graphs is reduced to listing the P_4's of a graph. In this paper, we provide polynomial time algorithms for finding and listing induced paths of given length. The problems of finding a triangle and listing all triangles of a graph are well known. Our results show an intimate connection between listing triangles and listing P_3's.

Let $p_k(G)$ and $c_k(G)$ denote the number of P_k's and C_k's of a connected graph G. Our main results are:

- ▷ a proof that listing the P_3's is as difficult as listing triangles,
- ▷ an algorithm to list all P_3's in $O(m^{1.5} + p_3(G))$ time,
- ▷ an algorithm to list all P_k's in $O(n^{k-1} + p_k(G) + k \cdot c_k(G))$ time where $k \geq 4$,
- ▷ an algorithm to list all C_k's in $O(n^{k-1} + p_k(G) + c_k(G))$ time, and
- ▷ an algorithm to find a P_k, $k \geq 5$, in time $O(k!! \cdot m^{(k-1)/2})$ if k is odd, and $O(k!! \cdot nm^{(k/2)-1})$ if k is even,

*Physics and Computer Science, Wilfrid Laurier University, Canada. Research supported by NSERC. email: choang@wlu.ca

†Algorithms Research Group, Département d'Informatique, Université Libre de Bruxelles O8.114, CP 212, Bvd. du Triomphe, 1050 Bruxelles email: Marcin.Kaminski@ulb.ac.be

‡School of Computer Science, University of Guelph, Canada. Research supported by NSERC. email: jsawada@uoguelph.ca

§Computer Science Department, The University of Dayton, Dayton, OH 45469, email: srithara@notes.udayton.edu, Acknowledges support from The National Security Agency, USA

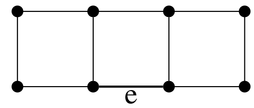

Figure 1: A non-brittle graph with no C_k, $k \geq 5$

where $k!!$ denotes the product $k(k-2)(k-4)\dots$. As applications of our findings, we give algorithms to recognize quasi-triangulated graphs and brittle graphs. Our algorithms' time bounds are incomparable with previously known algorithms.

2 Definitions and background

Throughout this paper we assume the graphs are without isolated vertices unless otherwise stated. Let $G = (V, E)$ be a graph. Then co-G denotes the complement of G. Let x be a vertex of G. $N(x)$ denotes the set of vertices adjacent to x in G. For a set S of vertices not containing x, we say x *is non-adjacent to* S if x is not adjacent to any vertex of S. Let P_k denote the induced path on k vertices, and $v_1v_2\cdots v_k$ the P_k with vertices $v_1, v_2, \dots, v_k$ and edges v_iv_{i+1} for $i = 1, 2, \dots, k-1$. $p_k(G)$ denotes the number of P_k's of G, and co-$p_k(G)$ denotes the number of co-P_k's of G. C_k, $k \geq 3$, denotes the chordless cycle with k vertices and $c_k(G)$ denotes the number of C_k's of G. K_t denotes the complete graph on t vertices. The graph co-C_4 is usually denoted by $2K_2$ and we call co-P_5 a *house*. C_3 is referred to as a *triangle*. The *paw* is the graph with vertices a, b, c, d and edges ab, bc, ac, ad. The *diamond* is the K_4 minus an edge. Let $k_3(G)$, respectively, $c_4(G)$, *co-*c_4*(G), paw(G),* $\diamond(G)$*,* $k_4(G)$, denote the number of triangles, respectively, C_4's, co-C_4's, paws, diamonds, K_4's, of G.

As usual, n denotes the number of vertices and m denotes the number of edges of the input graph. A vertex x is *simplicial* if its neighbors form a clique. A graph G is *quasi-triangulated* if each of its induced subgraphs H contains a simplicial vertex in H or in co-H. A graph is *chordal* if it does not contain an induced C_k for $k \geq 4$. Chordal graphs are well studied (for more information, see [8].) It is well known that a chordal graph contains a simplicial vertex. Thus, all chordal graphs are quasi-triangulated. The C_4 is quasi-triangulated but not chordal.

For a P_k $v_1v_2\cdots v_k$, $k \geq 4$, the edges $v_1v_2, v_{k-1}v_k$ are the *wings* of the P_k; the vertices v_1, v_k are the endpoints of the P_k. For a P_4 $abcd$, the edge bc is the *rib* of the P_4; b, c are the *midpoints* of the P_4. A vertex of a graph G is *soft* if it is not a midpoint of any P_4 or is not an endpoint of any P_4. A graph is *brittle* if each of its induced subgraphs contains a soft vertex. Since a simplicial vertex is soft and P_4 is self-complementary, quasi-triangulated graphs are brittle. Observe that any graph with a C_k where $k > 4$ is not brittle. The graph G in Figure 1 (with one edge labeled e) is a non-brittle graph containing no C_k, $k > 4$; $G - e$ is brittle and not quasi-triagulated. Quasi-triangulated graphs and brittle graphs are subclasses of perfectly orderable graphs [2] and are well studied (see [7, 9]). While quasi-triangulated and brittle graphs (see [9]) can be recognized in polynomial time, to recognize a perfectly orderable graph is an NP-complete problem [10].

Let O(n^α) be the current best complexity of the algorithm to multiply two $n \times n$ matrices. It has been long known that $\alpha < 2.376$ [3] and there is a recent unpublished improvement $\alpha < 2.3727$ [14]. It is well known that finding a triangle in a graph can be reduced to matrix multiplication. Thus, a triangle can be found in $O(n^\alpha)$ time. It follows from [1] that a triangle can be detected in time $O(m^{1.4})$ and all triangles can be listed in $O(m^{1.5})$ time.

3 Listing induced paths

In this section we discuss the problem of efficiently listing all induced P_k's in G. Naïvely, all P_k's of a graph G can be listed in $O(k^2 \cdot n^k)$ time by considering the $O(n^k)$ sequences of k distinct vertices and testing whether or not each sequence forms a P_k in $O(k^2)$ time. Taking k to be a constant, this is the best bound we can hope for when $p_k(G) = \Theta(n^k)$. For example, if the vertices of G are partitioned into k equal sized sets $V_1, V_2, \ldots, V_k$ where each V_i, V_{i+1} form a complete bipartite graph for $i = 1, 2, \ldots, k-1$ and there is no other edges, then $p_k(G) = \Theta(n^k)$. However, for many graphs we may expect a much smaller number for $p_k(G)$. In particular, for sparse graphs we can obtain a much better bound focusing on the number of edges m:

Theorem 1 *For $k \geq 1$, a graph G has*

$$p_k(G) = \begin{cases} O(nm^{(k-1)/2}) & \text{if } k \text{ is odd} \\ O(m^{k/2}) & \text{if } k \text{ is even.} \end{cases}$$

Moreover, for $k \geq 1$, all P_k's of a graph G can be listed in time $O(k!! \cdot nm^{(k-1)/2})$ if k is odd, or $O(k!! \cdot m^{k/2})$ if k is even.

Proof. By induction on k. Clearly $p_1(G) = O(n)$ and $p_2(G) = O(m)$. Since each P_{k-2} of the form $p_1p_2 \cdots p_{k-2}$ can be extended to a P_k in at most m ways, this proves the first part of the theorem. Specifically, we consider each edge xy and test whether or not either $p_1p_2 \cdots p_{k-2}xy$ or $p_1p_2 \cdots p_{k-2}yx$ is a P_k. This test will take $O(k)$ time for each edge, hence proving that we can list all P_k's in the given time bound. □

Ideally, we would like to list all P_k's in time $O(p_k(G) + m)$; that is it should cost only constant time per P_k. This seems a very challenging task, even for $k = 3$. Thus, perhaps it is more reasonable to look for an algorithm with running time $O(p_k(G) + n^t)$, where $t \leq k - 1$.

To begin, we consider a simple recursive algorithm to list all P_k's for a graph G. Assume that G is represented by the adjacency lists $adj[v]$ for each vertex $v \in V(G)$. If we consider each vertex as the starting point of a P_k, then we can generate all P_k's with the algorithm ListPath shown below:

```
function ListPath( int t )

    for each u ∈ adj[p_{t-1}] do vis[u] := vis[u] + 1
    for each u ∈ adj[p_{t-1}] do
        if vis[u] = 1 then
            p_t := u
            if t < k then ListPath(t+1)
            else if p_1 < p_t then Process(p_1 p_2 ··· p_t)
    for each u ∈ adj[p_{t-1}] do vis[u] := vis[u] − 1
end.
```

At each recursive call, we attempt to extend the induced path $p_1p_2 \cdots p_{t-1}$. To make the algorithm more efficient, for each vertex u, we maintain the value $vis[u]$ which is the number of vertices in $p_1p_2 \cdots p_{t-1}$ that are adjacent to u. This allows us to test whether or not a neighbour u of p_{t-1} can extend the induced path in constant time by checking if $vis[u] = 1$. To initialize the algorithm we set $vis[v] := 0$ for each vertex $v \in V(G)$. Then, for vertex $v \in V(G)$ we set $p_1 := v$, set $vis[v] := 1$, make the call ListPath(2), and finally reset $vis[v] := 0$. To make sure we do not list the same path twice, we only visit an induced path $p_1 p_2 \cdots p_t$ if $p_1 < p_t$ (given an initial ordering on the vertices). The function Process(P_k) is an application specific function to process the path. In particular, it can be written to simply output the vertices of the P_k.

A naïve analysis of the algorithm ListPath yields a worst case running time of $O(n^t)$. However, for sparse graphs this bound is not tight. A more detailed analysis for small values of k will be done in the following subsections, which measure the running time for a graph G with respect to the number of occurrences of certain induced subgraphs.

3.1 Listing P_3's

When $k = 3$, the algorithm ListPath is equivalent to considering each edge uv and then checking all neighbours w of u and v to see if we obtain a P_3 of the form uvw or vuw. If a neighbour w does not form a P_3 then it must be adjacent to both u and v and hence forms a triangle. Thus, an upper bound on the running time can be expressed as $O(m + k_3(G) + p_3(G))$. Each P_3 will be found twice and the algorithm can be optimized so that each triangle will be found exactly 3 times (by visiting the ordered adjacency lists for u and v at the same time). Since $k_3(G) = O(m^{1.5})$, the above discussion proves the following:

Theorem 2 *There is an $O(m + k_3(G) + p_3(G))$ = $O(m^{1.5} + p_3(G))$ algorithm to list all P_3's of a graph G.* □

Observe that this bound is incomparable with the $O(nm)$ approach from Theorem 1. Thus, the algorithm ListPath will be an improvement of this simpler approach in sparse graphs where $p_3(G) = o(m^2)$. We now mention a class of graphs for which Theorem 2 gives a better time bound than $O(nm)$. Consider the class P_n of induced paths on n vertices. With n and m being the number of respectively vertices and edges of P_n, we have $m = O(n), p_3(P_n) = O(n)$. So, Theorem 2's time bound is $O(n^{1.5})$ which is better than $O(nm) = O(n^2)$.

Note that some sparse graphs do admit $\Theta(m^2)$ P_3's as observed in the n-star, i.e., a graph on n vertices and $n - 1$ edges with a vertex of degree $n - 1$ (every other vertex has degree one). Thus, we consider one more approach that does not visit any triangles, as follows.

Theorem 3 *For any graph G, there is a $O(n + m + p_3(G)+$ co-$p_3(G))$ algorithm to list:*

- ▷ *all P_3's of G,*
- ▷ *all co-P_3's of G,*
- ▷ *all P_3's and co-P_3's of G.*

Proof. We first, for each isolated vertex, generate the co-P_3's containing that vertex at the cost of constant time per generated co-P_3. (This is the only time in the paper when we need to consider isolated vertices.) We remove all isolated vertices and apply the following algorithm.

```
for each x ∈ V do
    for each y ∈ V − N(x) − {x} do
        for each z ∈ N(y) do
            if xz ∈ E and y < x then Process(yzx)
            if xz ∉ E and y < z then Process({x, y, z})
```

In the above algorithm, the innermost **for** loop always iterates at least once and for each iteration either finds a P_3 (if $xz \in E$) or a co-P_3 (if $xz \notin E$). To remove duplicates, we only process those where $y < x$ or $y < z$ for an initial vertex ordering. □

For certain special classes of graphs, we can list P_3's more efficiently. For example, the following result applies to (diamond, house)-free graphs.

Theorem 4 *There is an $O(m^{\frac{2}{3}}\, n + p_3(G))$-time algorithm to list all P_3's of a (diamond, house)-free graph G.*

Proof. From [5] all maximal cliques of a (diamond, house)-free graph G can be listed in $O(m^{\frac{2}{3}}\, n)$ time. For a vertex v, $N(v)$ is the union of disjoint cliques $C_1, \ldots, C_k$ and $C_i \cup \{v\}$ is a maximal clique of G. Observe that xvy is a P_3 of G if and only if $x \in C_i, y \in C_j$ where $i \neq j$. If each vertex is given a list of pointers to the cliques that contain it (built up as the cliques are found) and each clique is given a list of vertices it contains, then the result follows. □

3.1.1 Lower bound

The following result shows, with an $O(n^2)$ reduction, listing P_3's is as hard as listing triangles, a well-known problem.

Theorem 5 *If there is an $f(n,m)$-time algorithm to list all P_3's of a graph, then there is an $O(n^2 + f(n,m))$-time algorithm to list all triangles of a graph.*

Proof. Given $G = (V, E)$, construct the bipartite graph $H = (X, Y, E')$ as follows: $X = \{w_1 \mid w \in V\}$, $Y = \{w_2 \mid w \in V\}$ (that is, for each vertex $w \in V$, put its copy w_1 in X, and w_2 in Y), and $E' = \{w_1 z_2 \mid wz \in E\} \cup \{w_2 z_1 \mid wz \in E\}$. It is clear that abc is a triangle in G if and only if $a_1 b_2 c_1$ is a P_3 of H such that a is adjacent to c in G. As H can be constructed from G in linear time, listing all the P_3 's of H in $f(2n, 2m)$ time enables us to list all the triangles of G in $O(n^2 + f(n,m))$ time. □

3.2 Listing P_4's

In addition to analyzing the algorithm for ListPath when $k = 4$, we will also consider three other approaches for listing all P_4's of a graph G. These three approaches can be summarized as follows:

1. Consider pairs of edges as potential wings of a P_4 and test if exactly one endpoint of each edge is adjacent to some endpoint of the other edge.

2. Consider all edges as a potential ribs of a P_4 and visit the neighbourhoods of each endpoint.

3. Consider each vertex as an endpoint of a P_4, and perform a BFS (Breadth-First Search) 4 levels deep. By keeping track of cross edges use the BFS tree to find all P_4's.

For the algorithm ListPath for $k = 4$, we consider the possible subgraphs at each level of computation. After three vertices have been added, we have exactly the analysis for generating P_3's: $O(m + k_3(G) + p_3(G))$. When adding a fourth vertex there are four possible induced subgraphs: a P_4, C_4, paw, or diamond. Thus, an upper bound on the running time can be expressed as:

$$O(m + k_3(G) + p_3(G) + c_4(G) + paw(G) + \diamond(G) + p_4(G)).$$

For the first alternate approach, we consider all pairs of edges uv and wx. If the two edges share a common endpoint, then we have either a P_3 or a triangle. If all four vertices are distinct, then we will obtain one of the following induced subgraphs on the four vertices: a $2K_2$, P_4, C_4, paw, diamond, K_4. A simple analysis yields $\Theta(m^2)$, but using occurrences of these subgraphs we get:

$$O(k_3(G) + p_3(G) + co\text{-}c_4(G) + c_4(G) + paw(G) + \diamond(G) + k_4(G) + p_4(G)).$$

Note this alternate approach gives a bound worse than that of the ListPath algorithm.

For the second alternate approach, we consider each edge uv as potential rib of a P_4, and then consider elements of the Cartesian product of the adjacency lists of u and v as the potential endpoints of a P_4. For a P_4 to be possible, the size of each adjacency list must be greater than 1 to account for their shared edge. If this condition is satisfied, then the possible subgraphs are: P_4, C_4, paw, diamond, or K_4. However, we can make a small improvement by first scanning the two adjacency lists concurrently (assuming the lists are ordered) and removing common vertices. Each vertex removed will correspond to a triangle. If each adjacency list still has more than one vertex, then we charge the non-removed vertices to the Cartesian product operation which will yield either a P_4 or a C_4. If after removing all shared vertices one of the adjacency lists has size 1, then we need to account for the vertices visited that are not shared. Since each such vertex forms a P_3, we can use the number of P_3's as an upper bound. Once this step is completed, we must make one more pass through the adjacency lists to insert back the removed vertices. This algorithm will take:

$$O(m + k_3(G) + p_3(G) + c_4(G) + p_4(G)) = O(m^{1.5} + p_3(G) + c_4(G) + p_4(G)) = O(nm + c_4(G) + p_4(G)).$$

Thus, for C_4-free graphs, we have the following result:

Theorem 6 *There is an $O(nm + p_4(G))$ algorithm to list all P_4's of a C_4-free graph G.* □

The final approach uses some pre-processing that will be beneficial in a paw-free graph. We consider each vertex u as the potential endpoint of a P_4. Then we perform a BFS starting from u up to 4 levels that constructs a parent list for each vertex v visited containing all neighbors of v in the previous level. Clearly by following any path from a vertex in the 4th level back to u we obtain a P_4, and all such paths can be easily found recursively using the parent pointer. In addition to such P_4's, a P_4 may also exist with two vertices in the third level. Thus when we perform our BFS, we must also keep track of all edges whose endpoints are both in the third level. For each such edge xw we scan the parent lists of each vertex. If they share a parent then we have a paw, but for each parent of one vertex that is not in the parent list of the other, we find a P_4. To handle equivalent P_4's, we only list those where the first vertex is greater than the last vertex for some given vertex ordering. This algorithm will take:

$$O(nm + paw(G) + p_4(G)).$$

Thus, we have the following result:

Theorem 7 *There is an $O(nm + p_4(G))$ algorithm to list all P_4's of a paw-free graph G.* □

If a graph is either paw-free or C_4-free then we have algorithms that run in time $O(nm + p_4(G))$. An open question is whether or not there exists an algorithm that runs in time $O(n^3 + p_4(G))$ to list all P_4's for an arbitrary graph G.

3.3 Listing P_k's

In this subsection, we consider two general approaches to list all P_k's of a graph G. First, recall that the algorithm ListPath for $k = 3$ runs in time $O(m^{1.5} + p_3(G))$. Extending this algorithm to $k = 4$, observe that in the worst case we perform $O(n)$ extra work for each P_3. Thus, we can generate all P_4's in $O(m^{1.5} + n \cdot p_3(G))$. The following generalizes this observation for larger k.

Theorem 8 *There is an $O(m^{1.5} + n^{k-3} \cdot p_3(G))$ algorithm to list all P_k's of a graph G.* □

Our second approach extends the *rib* approach we used to list all P_4's; however instead of a rib we start by considering a P_{k-2}. If L is a list of all P_{k-2}'s in G ($k \geq 4$), the following approach will list all P_k's:

---

for each $P := p_1 p_2 \cdots p_{k-2} \in L$ **do**
  A := set of vertices adjacent to p_1 and non-adjacent to $P - \{p_1\}$
  B := set of vertices adjacent to p_{k-2} and non-adjacent to $P - \{p_{k-2}\}$
  for each $(a, b) \in A \times B$ **do**
    if $ab \notin E$ **then** Process(aPb)

To analyze this algorithm, observe that A and B can be computed in $O(kn)$ time and the nested **for** loop either generates a P_k or a C_k (when $ab \in E$). Also note that each C_k will be generated k times. Thus, using the upper bound of $O(n^{k-2})$ for the number $p_{k-2}(G)$, we obtain an overall running time bound of $O(kn^{k-1} + p_k(G) + k \cdot c_k(G))$ for this algorithm. The factor k in front of the term n^{k-1} is somewhat undesirable; however, this factor can be eliminated by modifying the algorithm to start with P_{k-4}'s. In the following algorithm, L is a list of all P_{k-4}'s in G and $k \geq 6$:

---

for each $P := p_1 p_2 \cdots p_{k-4} \in L$ **do**
  A := set of vertices adjacent to p_1 and non-adjacent to $P - \{p_1\}$
  B := set of vertices adjacent to p_{k-4} and non-adjacent to $P - \{p_{k-4}\}$

```
C := set of vertices non-adjacent to P
for each (a, b) ∈ A × B do
    if ab ∉ E then
        A' := subset of C adjacent to a but not b
        B' := subset of C adjacent to b but not a
        for each (u, v) ∈ A' × B' do
            if uv ∉ E then Process( uaPbv )
```

In the case when $k = 4$ and $k = 5$ we can consider k to be constant. Thus, we obtain the following theorem.

Theorem 9 *There is an $O(n^{k-1} + p_k(G) + k \cdot c_k(G))$ algorithm to list all P_k's of a graph G, where $k \geq 4$.*

Proof. When $k = 4$ and $k = 5$ we can consider k to be constant. Thus, as discussed the first algorithm presented in this section attains the time bound. When $k \geq 6$, the factor of k in front of the term n_{k-1} is handled by the latter algorithm. In that algorithm we note that each C_{k-2} will be considered $k-2$ times (when $ab \in E$); however, this work is contained in the term n^{k-1}. □

Corollary 1 *There is an $O(n^{k-1} + p_k(G))$ algorithm to list all P_k's of a C_k-free graph G.* □

Note that the bounds in Theorem 9, Corollary 1, and the upcoming Theorem 10, can be tightened for sparse graphs by applying Theorem 1.

3.4 Listing C_k

Observe that many of the algorithms for listing P_k's can easily be modified to list all C_k's. In particular, to apply the algorithm immediately preceding Theorem 9, observe that we will find all C_k's if $uv \in E$ in the final statement of the algorithm. However, one extra challenge for cycles is to easily identify duplicates. This can be done by ensuring that we only list cycles of the form $C = c_1c_2 \cdots c_k$ where c_1 is the smallest vertex in C and $c_2 < c_k$. There are three steps to doing this efficiently. First, when we list the P_{k-4}, we must keep track of the smallest vertex in the induced path as the path gets generated. This can easily be done in constant time (per added vertex). Second, we do not consider a P_{k-4} unless the smallest vertex is p_1. Finally, when we consider (u, v) in the final **for** loop, we will find a representative C_k of the form $Pbvua$ if and only if p_1 is smaller than each of b, v, u, a and $p_2 < a$. Observe that each C_k will be tested at most a constant number of times. Thus, the algorithm for listing C_k's yields an improved bound compared to the one we give to list P_k's.

Theorem 10 *There is an $O(n^{k-1} + p_k(G) + c_k(G))$ algorithm to list all C_k's of a graph G.* □

For the special case of $k = 4$, observe that the algorithm used to prove Theorem 6 can also be adapted to obtain the following result (which is an improvement for sparse graphs).

Theorem 11 *There is an $O(nm + p_4(G) + c_4(G))$ algorithm to list all C_4's of a graph G.* □

For sparse graphs, we can also make simple modifications to the proof of Theorem 1 to obtain similar results for cycles. Using the techniques just described, we can easily obtain only the representative cycles with constant time testing.

Theorem 12 *All C_k's of a graph G can be listed in time $O(k!! \cdot nm^{(k-1)/2})$ if k is odd, or $O(k!! \cdot m^{k/2})$ if k is even.*

4 Finding induced paths

One can find a P_3 in linear time since a graph contains a P_3 if and only if it has a component that is not a clique. It is also known that finding a P_4 (if one exists) in a graph can be done in linear time [4]. In this section, we give an algorithm for finding a P_k, $k \geq 5$, that is better than the naïve $O(n^k)$ algorithm. We first consider an auxiliary problem:

Problem P1. Let a be a vertex of G and let B be a subset of $V - \{a\}$. Is there a P_4 of the form $abcd$ where $b, c, d \in B$?

This problem can be answered in $O(nm)$ time using the following approach: for each $b \in N(a)$ and each $cd \in E$ with $b, c, d \in B$, test if $abcd$ is a P_4. However, it is possible to achieve a more efficient algorithm for Problem **P1** by computing certain components. In particular, if B' is the subset of vertices in B that are not adjacent to a, then compute the components $C_1, C_2, \ldots, C_t$ for the subgraph of G induced by B'. The desired P_4 exists if and only if there is a vertex $b \in N(a) \cap B$ that is adjacent to some but not all vertices of some C_i. The following provides a detailed explanation of how we efficiently test each b:

$B' := B - N(a)$
$C_1, C_2, \ldots, C_t$:= components of the subgraph of G induced by B'
initialize counters $c_1, c_2, \ldots, c_t$ to 0
$L :=$ an empty list
for each $b \in N(a) \cap B$ **do**
    for each $c \in N(b) \cap B'$ **do**
        j := index of component containing c
        $c_j := c_j + 1$
        if $c_j = 1$ **then** add j to L
    for each $j \in L$ **do**
        if $c_j < |C_j|$ **then return** "yes"
        $c_j := 0$
        remove j from L
return "no"

The components can be computed in $O(m)$ time and it is easy to maintain which component each vertex belongs to. The first nested **for** loop visits a unique edge bc, and the second nested **for** loop is executed at most the same number of times as the first **for** loop. Thus, the overall running time is $O(m)$. Observe that if we did not maintain the list L , the second nested **for** loop could be altered to test if $0 < c_j < |C_j|$ for each $1 \leq j \leq t$.

However, in the worst case there may be $O(n)$ components and hence the running time would be $O(n^2)$. When the algorithm returns "yes", we can produce a P_4 as follows. Retrieve the current vertex b and component C_j. Let X, respectively, Y, be the set of vertices of C_j adjacent, respectively, non-adjacent, to b. Find an edge xy with $x \in X$, $y \in Y$. Since C_j is connected, such an edge exists. Then, $abxy$ is the desired P_4. The above discussion establishes the following theorem:

Theorem 13 *Problem* **P1** *can be solved in* $O(m)$ *time.* □

Theorem 14 *There is an* $O(m^2)$ *algorithm to find a* P_5 *in a graph* G *if one exists.*

Proof. For each edge uv, we test if uv extends into a P_5 of the form $uvwxy$ or $vuwxy$ for some vertices w, x, y. This is done by solving Problem **P1** with $a = v$ and $B = V - N(u)$ (respectively, $a = u$ and $B = V - N(v)$). □

Theorem 15 *For* $k \geq 5$*, a* P_k*, if one exists, can be found in a graph* G *in time*

(i) $O(k!! \cdot m^{(k-1)/2})$ *if* k *is odd,*

(ii) $O(k!! \cdot nm^{(k/2)-1})$ *if* k *is even.*

Proof. We have seen the theorem holds for $k = 5$. Suppose $k > 5$. We list all P_{k-3}'s. For each of these paths, we test if it can be extended into a P_k. Consider a path $v_1 v_2 \cdots v_{k-3}$. We test in time $O(m)$ that this path can be extended into a P_k $v_1 v_2 \cdots v_{k-3} bcd$ by solving Problem **P1** with $a = v_{k-3}$ and $B = V - (N(v_1) \cup N(v_2) \cup \ldots \cup N(v_{k-4}))$. Using the bound to list all P_{k-3}'s given in Theorem 1, the result follows. □

4.1 A note on finding C_4 and C_5

The purpose of this section is to show that finding C_4 and C_5 are related to finding certain P_4's in a graph. It is known a C_k, $k \geq 4$, can be found in $O(n^{k-3+\alpha})$ time [12]. In particular, a C_4, respectively, C_5, can be found in $O(n^{3.376})$, respectively, $O(n^{4.376})$, time. Intuitively, finding a C_k (respectively, P_k) should be at least as hard as finding a C_{k-1} (respectively, P_{k-1}), for $k \geq 4$. But this seems to be a challenging problem. We do not even have a solution in the case $k = 4$. We will show that finding a C_5 is at least as hard as finding a triangle. First, consider the following

Problem P2. Let the vertices of a graph G be partitioned into two sets A, B. Is there a P_4 of the form $abcd$ where $a, d \in A$, and $b, c \in B$?

We can find a C_5 as follows: For each vertex x, define $A = N(x)$, $B = V - A - \{x\}$. Then x belongs to a C_5 if and only if Problem **P2** has a positive answer on the sets A, B. So, if we can solve Problem **P2** in $O(n^3)$ time, then we can find a C_5 in $O(n^4)$ time.

Theorem 16 *If there is an* $f(n, m)$*-time algorithm for Problem* **P2***, then there is an* $O(n^2 + f(n, m))$*- time algorithm to find a* C_4*.*

Proof. Let G be an instance of the problem of finding a C_4. We will construct an instance H of Problem **P2**. Let B be a copy of G and A be a copy of the complement of G. For a vertex $x \in A$ and a vertex $y \in B$, add the edge

xy if $x = y$ (x and y are the same vertex in G), or xy is an edge in G. Suppose G contains a C_4 $abcd$. Then H contains a P_4 $abcd$ with $a, d \in A, b, c \in B$ (actually, H contains four such P_4's). Now, suppose H contains a P_4 $abcd$ with $a, d \in A, b, c \in B$. a and b cannot be the same vertex in G, for otherwise a would be adjacent to c in H, a contradiction. Similarly, c and d are different vertices in G. Thus, G contains the C_4 $abcd$. □

The above shows it will be difficult to solve Problem **P2** in $O(n^3)$ time since finding a C_4 in $O(n^3)$ time is a well-known open problem. Can we prove that finding a C_5 is at least as hard as finding a C_4? Consider the following problem:

Problem P3. Given a graph G and a vertex x. Is there a C_5 of G containing x?

Note that Problems **P2** and **P3** are linear-time equivalent.

Theorem 17 *If there is an $f(n,m)$-time algorithm for Problem* **P3***, then there is an $O(n^2 + f(n,m))$- time algorithm to find a triangle.*

Proof. Given G with vertex set $V = \{1, 2, \ldots, n\}$, construct H as follows. Make four copies H_1, H_2, H_3, H_4 of V. For $t = 1, 2, 3$, and for vertices $i \in H_t$, $j \in H_{t+1}$, if ij is an edge of G, then add the edge between the copy of $i \in H_t$ and the copy of $j \in H_{t+1}$. For vertices $i \in H_1, j \in H_4, i \neq j$, add the edge between the copy of $i \in H_1$ and $j \in H_4$. Observe that the graph constructed so far is bipartite. Add a vertex x adjacent to all vertices in $H_1 \cup H_4$. Call the resulting graph H.

Suppose there is a $C_5 = xpqrs$ containing x in H. It is easy to see that each of $\{p, q, r, s\}$ is in a distinct H_i. Without loss of generality, we may assume $p \in H_1, s \in H_4$. p and s must be copy of the same vertex of G. Therefore, p, q, r form a triangle in G.

Suppose G contains a triangle pqr. Then H contains the C_5 $xpqrp$. □

Note that in the above proof, any C_5 of H must contain x. So, testing for a C_5 is at least as hard as testing for a triangle. Now, one may want to prove the converse of Theorem 16. But this would mean that finding a C_4 is at least as hard as finding a triangle, which is a long-standing open problem.

A modification of the construction in [12] shows that deciding whether there is a C_5 containing a given edge can be solved in $O(n^\alpha)$ time. Therefore Problem **P3** can be solved in time $O(n^{\alpha+1})$ by testing, for each edge incident to x, if there is a C_5 containing it.

5 Applications

The ability to recognize whether or not a graph G belongs to a class C has been widely studied for many graph classes. Many classes of graphs, including chordal graphs, strongly chordal graphs, quasi-triangulated graphs and brittle graphs have particular "special vertices" that can be used to solve the recognition problem. For such a class C and its corresponding special vertex definition, the following generic approach can be used to determine if G belongs to C:

L:= list of "special vertices"
while L not empty **do**
    remove a vertex v from L
    remove v from G
    update L
if G is empty **then return** $G \in C$
return $G \notin C$

The overall running time of this approach is dependent on the time to initialize and update L. For quasi-triangulated graphs and brittle graphs, we will use listings of appropriate P_k's that will optimize these steps.

5.1 Recognizing quasi-triangulated graphs

For quasi-triangulated graphs, the "special vertices" are those that are either simplicial (not a middle vertex of any P_3) or co-simplicial (not an isolated vertex in any co-P_3). Using the generic recognition algorithm, we can optimize the initialization and maintenance of these special vertices as follows:

- ▷ List and store all P_3's and co-P_3's in a table T,
- ▷ for each vertex v let $Q[v]$ be a list of pointers to all items in T containing v,
- ▷ for each vertex v let $mid[v]$ hold the number of P_3 with v as a midpoint
- ▷ for each vertex v let $iso[v]$ hold the number of co-P_3 with v as the isolated vertex
- ▷ store all special vertices v (those with $mid[v] = 0$ or $iso[v] = 0$) in a list L.

While constructing the table T it is easy to update the values for Q, mid, and iso in constant time per table element. Observe that this pre-computation runs in time proportional to the time it takes to list the P_3's and co-P_3's. To maintain these data structures as a special vertex s is removed from L and G, remove all the elements from T pointed to by any element of $Q[s]$. When removing a P_3 avb, the value $mid[v]$ is decremented; when removing a co-P_3 with v as the isolated vertex, the value $iso[v]$ is decremented. If either $mid[v] = 0$ or $iso[v] = 0$ after an update, insert v into L. Observe that each item in T is removed at most once, and thus quasi-triangulated graphs can be recognized in the time it takes to list all P_3's and co-P_3's. Thus, Theorem 3 leads to the following result:

Theorem 18 *There is a $O(n + m + p_3(G)+$ co-$p_3(G))$ algorithm to recognize quasi-triangulated graphs.* □

The above result gives a time bound for recognition of quasi-triangulated graphs that is incomparable to the bound of $O(n^{2.77})$ given in [13]. We now mention a specific class of graphs for which Theorem 18's time bound is better than O($n^{2.77}$). Consider the class P_n of induced paths on n vertices. With n and m being the number of respectively vertices and edges of P_n, we have $m = O(n), p_3(P_n) = O(n), co\text{-}p_3(D) = O(n^2)$. So, Theorem 18's time bound of $O(n^2)$ is better than $O(n^{2.77})$.

5.2 Recognizing brittle graphs

The following simple approach, which is folklore ([11], page 31), can be used to recognize brittle graphs:

If there exists a soft vertex, remove it. Repeat this process until there is no soft vertex or the graph is empty. If the graph is empty, the original graph was brittle; otherwise, it was not brittle.

The key to this algorithm is to detect soft vertices. The following method uses significant pre-computation:

- ▷ List and store all P_4's in a table,
- ▷ for each vertex v let $Q[v]$ be a list of pointers to all the P_4's containing v,
- ▷ for each vertex v let $mid[v]$ hold the number of P_4's with v as a midpoint,
- ▷ for each vertex v let $end[v]$ hold the number of P_4's with v as an endpoint,
- ▷ store all soft vertices v (those with $mid[v] = 0$ or $end[v] = 0$) in a list L.

While computing the P_4's, it is easy to update the values for Q, mid and end in constant time per P_4. Observe that this pre-computation runs in time proportional to the time it takes to list the P_4's. To maintain these data structures as a soft vertex s is removed from L and the graph, remove all the P_4's pointed to by any element of $Q[s]$ from the table. When removing a P_4, the values $mid[u]$, $end[u]$ are updated for each u (not equal to s) in the P_4. If u becomes soft, it is inserted into L. Observe that each P_4 is removed at most once, and thus brittle graphs can be recognized in the time it takes to list all P_4's. The results from Section 3 lead to the following three theorems:

Theorem 19 *A brittle graph G can be recognized in O(*$m^{1.5} + n \cdot p_3(G)$*) time.* □

Theorem 20 *A brittle graph G can be recognized in O(*$nm + p_4(G) + paw(G)$*) time.* □

Theorem 21 *A brittle graph G can be recognized in O(*$nm + p_4(G) + c_4(G)$*) time.* □

The above results give time bounds for recognition of brittle graphs that are incomparable to the bounds of $O(n^{3.376})$ or $O(n^3 \log n \log n)$ given in [6], and the bound $O(m^2)$ given in [11]. Consider the class P_n of induced paths on n vertices. With n and m being the number of respectively vertices and edges of P_n, we have $m = O(n), p_3(P_n) = O(n), p_4(P_n) = O(n), paw(P_n) = 0, c_4(P_n) = 0$. So the bounds of Theorems 19, 20, 21 are $O(n^2)$ which is better than those of [6, 11].

6 Open Problems

In our discussion, we have mentioned the following 3 open problems:

1. Does there exist an algorithm that runs in time $O(n^3 + p_4(G))$ to list all P_4's for an arbitrary graph G?
2. Is finding a C_5 at least as hard as finding a C_4 in an arbitrary graph G?
3. Is finding a C_4 at least as hard as finding a C_3 in an arbitrary graph G?

The last question is a long-standing open problem.

References

[1] N. Alon, R Yuster, U. Zwick, Finding and counting given length cycles, *Algorithmica* **17** (1997) 209–223.

[2] V. Chvátal, Perfectly orderable graphs, in: C. Berge, V. Chvátal, *Topics in Perfect Graphs*, Annals of Discrete Mathematics, 21 (1984), Amsterdam: North-Holland, pp. 63–68.

[3] D. Coppersmith, S. Winograd, Matrix multiplication via arithmetic progressions, *Journal of Symbolic Computation* **9**:3 (1990) 251–280.

[4] D. G. Corneil, Y. Perl, L. K. Stewart, A linear recognition algorithm for cographs, *SIAM Journal on Computing* **14** (1985) 926–934.

[5] E. M. Eschen, C. T. Hoàng, J. P. Spinrad, R. Sritharan, On graphs without a C_4 or a diamond, to appear in *Discrete Applied Mathamatics* **159**(7) (2011) 581–587.

[6] E. M. Eschen, J. L. Johnson, J. P. Spinrad, R. Sritharan, Recognition of some classes of perfectly orderable graphs, *Discrete Applied Mathamatics* **128** (2003) 355–373.

[7] I. Gorgos, C. Hoàng, V. Voloshin, A note on quasi-triangulated graphs, *SIAM J. Discrete Math.* **20**:3 (2006) 597–602.

[8] M.C. Golumbic. Algorithmic Graph Theory and Perfect Graphs. Academic Press, 1980.

[9] C. T. Hoàng. Perfectly orderable graphs: a survey. In: Perfect Graphs. J. L. Ramirez-Alfonsin and B. A. Reed (Editors), pp 145–169. Wiley, 2001.

[10] M. Middendorf, F. Pfeiffer, On the complexity of recognizing perfectly orderable graphs, *Discrete Math.* **80** (1990) 327–33.

[11] A. A. Schäffer, Recognizing brittle graphs: remarks on a paper of Hoàng and Khouzam, *Discrete Applied Mathematics* **31** (1991) 29–35.

[12] J. P. Spinrad, Finding large holes, *Information Processing Letters* **39** (1991) 227–229.

[13] J. P. Spinrad, Recognizing quasi-triangulated graphs, *Discrete Applied Mathematics* **139** (2004) 203–213.

[14] Virginia Vassilevska Williams, Breaking the Coppersmith-Winograd barrier, `http://www.cs.berkeley.edu/~virgi/matrixmult.pdf`