Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 273 (2025) 86–93

Procedia Computer Science

www.elsevier.com/locate/procedia

XIII Latin American Algorithms, Graphs, and Optimization Symposium (LAGOS 2025)

A polynomial-time algorithm recognizing exact cubes of trees

Laurent Beaudou[a], Henry Echeverría[b,c], Florent Foucaud[a], Andrea Jiménez[b,c], Nikita Manuylenko[d], Anirudh Rachuri[a]

[a]*Université Clermont Auvergne, CNRS, Clermont Auvergne INP, Mines Saint-Étienne, LIMOS, 63000 Clermont-Ferrand, France*
[b]*Instituto de Ingeniería Matemática, Universidad de Valparaíso, Chile.*
[c]*CIMFAV, Universidad de Valparaíso, Chile.*
[d]*Faculty of Computer Science, Higher School of Economics, Moscow, Russia*

Abstract

We prove that the recognition of exact cubes of trees can be done in polynomial time. More precisely, the exact distance power of a graph is a refinement of the more usual notion of graph power. Given a graph G and a positive integer p, the exact distance p^{th} power of G is the graph $G^{\#p}$ on the same vertex set where two vertices are adjacent if their distance is exactly p in G. Recently Bai et al. [Y. Bai, P. P. Cortés, R. Naserasr and D. A. Quiroz. Characterizing and recognizing exact-distance squares of graphs. *Discrete Mathematics* **347**(8). 2024] proved that the recognition of exact squares of trees is polynomially tractable. In order to extend this result to exact cubes of trees, we first test whether there is a caterpillar as an exact cubic root and, if not, proceed with a general tree. Both algorithms rely on the observation that the knowledge of a fixed number of vertices is roughly enough to deduce the whole structure of the tree we aim for.

© 2025 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0)
Peer-review under responsibility of the Program Committee of LAGOS 2025.

Keywords: Exact distance power graphs; Trees; Matchings
2000 MSC: 68R10; 05C85; 05C12

1. Introduction

One may argue that graph theory is nothing but using local knowledge (the edges between pairs of adjacent vertices) in order to draw global conclusions (for example, all vertices are connected). When knowing neighbours is not enough, one may ask for the list of not-too-far vertices: this would make a rough definition of graph power. Given a graph G and a positive integer p, the *p^{th} power of G*, denoted G^p, is the graph on the same vertex set with an edge between any pair of vertices whose distance in G is p or less. When p is 2 or 3, we use the natural denomination: G^2 is the square of G and G^3 is its cube.

A natural question arises when such an operation is described: is it easy to recognize whether a given graph is the p^{th} power of some other graph? The actual notion of graph square was formally introduced by Ross and Harary [19] in 1960. In their paper, they characterize the squares of trees and present a linear-time algorithm to recognize them. More recently, Chang, Ko and Lu [5] in 2015 gave, for any positive integer p, a linear-time algorithm to recognize the

1877-0509 © 2025 The Authors. Published by Elsevier B.V.
This is an open access article under the CC BY-NC-ND license (https://creativecommons.org/licenses/by-nc-nd/4.0)
Peer-review under responsibility of the Program Committee of LAGOS 2025.
10.1016/j.procs.2025.10.284

p^{th} power of a tree. Seven years after the work of Ross and Harary, Mukhopadhyay [14] gives a characterization of graphs that admit a square root (not necessarily a tree). This characterization, however, does not provide a polynomial time algorithm, and, in 1994, Motwani and Sudan [13] prove that recognizing graphs admitting a square root is NP-complete. This result motivated the introduction of the recognition problem for squares of specific classes of graphs. Given a class of graphs $\mathscr{C}$, the $\mathscr{C}$-root-recognition problem asks whether a given graph G can be obtained as the p^{th} power of some graph H in $\mathscr{C}$. This problem has been studied in different settings [7, 10]. Other than recognition, graph powers have attracted much attention for a range of other problems like coloring [1, 12, 11], or structural properties like hamiltonicity [22, 15, 16].

In the graph G^p, a vertex is adjacent to all vertices at distance at most p in G without any distinction. A natural refinement of this notion is the *exact distance p graph* denoted $G^{\#p}$. Given a graph G on vertex set V, the exact distance p graph $G^{\#p}$ has the same vertex set V and two vertices u and v are adjacent in $G^{\#p}$ if they are at distance exactly p in G. When p is 2 or 3, we speak of the *exact square* or *exact cube* of a graph, and *exact square root* or *exact cubic root* when they exist.

Similar questions naturally emerge with this notion of exact distance. Recently, it was studied in the context of coloring [3, 24, 8, 17, 18], and structural properties [9, 4]. We focus on the recognition problem. This paper takes root in the work of Bai et al. [2]. They prove that recognizing graphs admitting an exact square root can be done in polynomial time, contrasting with the usual square case. In fact, they prove that if a graph G admits an exact square root, then its complementary graph $\overline{G}$ is among those exact roots [2, Theorem 1.1]. In addition, they prove that when restricting the class $\mathscr{C}$ to bipartite graphs, the problem is NP-complete [2, Theorem 1.2]. Finally, they prove that when $\mathscr{C}$ is the family of trees, the problem is again polynomially tractable [2, Theorem 1.3]. So we have this interesting progression P, NP-complete, P based on the class $\mathscr{C}$ being trees, bipartite, or all graphs. We started wondering if the problem for trees remained polynomial for higher exact powers. We naturally picked the first open case, $p = 3$. In his Master's thesis, Cortés [6] describes algorithms to decide whether a disconnected graph admits a tree as an exact cubic root. Note that except for small diameter trees (like stars), the exact cube of a tree has either one, two or three connected components.

Our contribution. In this paper, we prove that the exact cubes of trees can be recognized in polynomial time.

Theorem 1. *There exists a polynomial time algorithm which takes a graph G as input and decides whether there exists a tree T such that $T^{\#3} = G$. Moreover, if yes, it provides such a tree.*

The particularity of our approach is to observe that few fixed vertices dictate the adjacency between most vertices. In addition to this rather simple roadmap, we use induction on the tree structure and get help from matching theory.

Organization of the paper. The rest of the paper follows a natural progression. In Section 2, we treat the case of caterpillars; namely, we prove that deciding if a given graph admits an exact cubic root in the form of a caterpillar is polynomially tractable. In Section 3, we describe the algorithm to follow when the given graph does not admit a caterpillar as an exact cubic root. In this case, we use the observation that any tree which is not a caterpillar has seven vertices inducing a subdivided claw. Together, these two algorithms prove Theorem 1.

2. Recognizing exact cubes of caterpillars

A *caterpillar* is a tree such that when leaves are removed, the resulting graph is a path. Our first result is the recognition of their exact cubes.

Lemma 2. *There exists a polynomial-time algorithm to decide whether a given graph G can be expressed as the exact cube of a caterpillar. Moreover, this algorithm provides an exact cubic root of G as a caterpillar when it exists.*

Proof. In order to explain the recognition of such graphs, we need to observe some simple facts about the exact cube of a caterpillar. As said before, a caterpillar C contains a path P from which leaves are pending. Thus, there exist an integer k as well as $k + 1$ vertices $v_0, v_1, \ldots, v_k$ such that $P = v_0 v_1 \ldots v_k$ (we may assume that no leaves are pending from v_0 and v_k). The integer quotient of k by 3 will be denoted q and the remainder r (so $0 \leq r \leq 2$). Thus $k = 3q + r$.

If k is 0, then the caterpillar C is just a single vertex (and so must be its exact cube). If k is 1, it is just K_2 (we have no pending leaves on the extremities of the path), in which case its exact cube is a stable set on two vertices.

Suppose now that k is 2 or more. We may thus safely speak of vertices v_0, v_1 and v_2. To each of them, we associate the furthest vertex of P at a distance multiple of 3. This means that v_0 is associated to v_{3q}, v_1 is associated to v_{3q-2} or v_{3q+1} (depending on remainder r) and v_2 is associated to v_{3q+2} or v_{3q-1} (also depending on r). Let us call u_0, u_1 and u_2 the vertices associated to v_0, v_1 and v_2 (observe that u_i and v_i might be the same vertex).

In the exact cube of caterpillar C, there is a unique shortest path from v_i to u_i (namely the one following the path P with jumps of length 3). Moreover, these three shortest paths differ in length by at most 1. They describe the whole path P by taking the next vertex consecutively and cyclically in each of those three paths. Any other vertex of C is a pending leaf on some vertex v_i of the path P. It is thus adjacent to v_{i-2} and v_{i+2} (provided that they exist) and all the pending leaves of v_{i-1} and v_{i+1}. These adjacencies can be decided unequivocally once the path P is fixed. Figure 1 illustrates this paragraph with an example. The upper drawing depicts a graph G with guessed vertices $v_0, v_1, v_2, u_0, u_1, u_2$, bold edges are the shortest paths between those pairs. The bottom drawing is the caterpillar obtained after the identification process previously described. Afterwards, one can check that the cube of this caterpillar is indeed the given graph G.

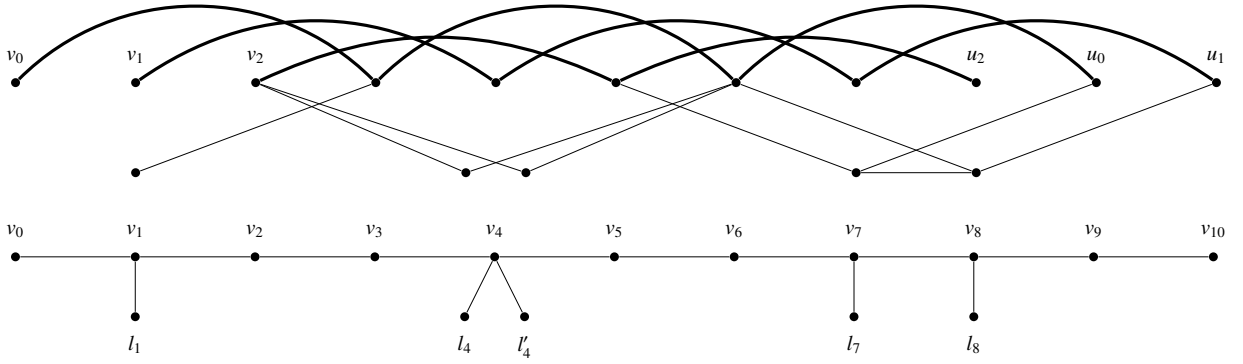


Fig. 1. A graph with fixed $v_0, u_0, v_1, u_1, v_2, u_2$, and one of its cubic root as a caterpillar.

This discussion allows us to derive a straightforward algorithm to recognize exact cubes of caterpillars:

1. Guess the six vertices $v_0, u_0, v_1, u_1, v_2, u_2$ ($O(n^6)$ possible choices):
2. Compute shortest paths (in G) between those pairs ($O(n^2)$ through BFS),
3. Deduce path P (linear time),
4. For all remaining vertices, decide where it is attached on P (constant time for each vertex, so linear as a whole),
5. Check that the obtained caterpillar is indeed an exact cubic root of G (quadratic for trees through LCA methods, for example, see Tarjan [21]).

If all 6-tuples fail the test, we may safely assert that G admits no caterpillar as an exact cubic root. If one of them passes the test, we obtain both a positive answer and an exact cubic root for G. In the end, we obtain a procedure running in time $O(n^8)$ which recognizes whether a graph G is the exact cube of a caterpillar. Moreover, if the answer is positive, it provides such a caterpillar. □

3. Recognizing the exact cube of a general tree

After having tested if the given graph G is the exact cube of a caterpillar (see Section 2), when the answer is negative, we are left with deciding whether G is the exact cube of a tree which is not a caterpillar. Observe that such a tree admits a vertex r of degree at least 3 such that three of its neighbours have degree at least 2 (r is the center of a subdivided claw). In the same flavour as for caterpillars, we shall identify some specific vertices and observe that, when fixed, the whole tree is more or less deduced from G. The specific vertices will be exactly the seven vertices described above. The vertex r has been so named because we want to look at T as a tree rooted in r.

3.1. Family relations are odd

In the remainder of this paper, we extensively use genealogy terms to characterize the vertices of the rooted tree that we build. In addition to the usual *father/children* terminology, we use specific terms that deserve a subsection to be introduced.

Given a rooted tree (T, r), we obviously call r the *root*. For any other vertex v, the *father* of v is its unique neighbour u on the unique path from v to r. Conversely, v is a *child* of u. If two vertices share a common father, we call them *brothers*. We also use the obvious *uncle/nephew* relationship (a brother of my father is an uncle). The *generation* of a vertex v is its distance to vertex r. Its *ancestors* are all the vertices on its unique path to r, and its *descendants* are all the vertices u such that v is an ancestor of u. When jumping more than one generation, we use the terms i-ancestor (and i-descendant). Given a vertex v at distance k from r, and given an integer i between 0 and k, the *i-ancestor* of v is the unique vertex at distance i from v on its path to r. Note that a 1-ancestor is nothing but a father. Reciprocally, we define the notion of *i-descendant*. Finally, among descendants of a vertex v, we sometimes want to isolate those whose distance to v is a multiple of 3. We call them *third descendants*. The third descendants of some vertex v are all $3k$-descendants of v for any integer k (note that it includes v itself).

3.2. Seven vertices to rule them all

As mentioned above, we are now in the case when the given graph G admits no caterpillar as an exact cubic root. A tree which is not a caterpillar necessarily has a vertex r of degree at least 3 such that three of its neighbours have degree at least 2. Let us name x_1, x_2, x_3 three neighbours of r and y_1, y_2, y_3 their other neighbours (at distance 2 from r). We shall see that deciding whether such a 7-tuple extends to a full exact cubic root of G can be done in polynomial time. It will, then, be easy to just test all potential 7-tuples; we already use time $O(n^8)$ just to decide if G admits a caterpillar as an exact cubic root so it should be clear to the reader that we aim for a polynomial time algorithm but not a fast one, thus testing all 7-tuples is nothing to be afraid of.

Deciding the generation of all vertices. Let $\mathcal{L}_i$ denote the set of vertices of generation i. By definition, $\mathcal{L}_0$ is nothing but the singleton $\{r\}$. The first generation consists of exactly those vertices that have at least two neighbours (in G) among y_1, y_2 and y_3. Let $N_G(v)$ denote the set of neighbours of a vertex v in graph G.

$$\mathcal{L}_1 = \{v : |N_G(v) \cap \{y_1, y_2, y_3\}| \geq 2\}.$$

Similarly, the second generation is the set of those vertices having at least two neighbours in $\{x_1, x_2, x_3\}$.

$$\mathcal{L}_2 = \{v : |N_G(v) \cap \{x_1, x_2, x_3\}| \geq 2\}.$$

For $i \geq 3$, generation i is characterized by the fact that its vertices have a 3-ancestor (thus a neighbour in $\mathcal{L}_{i-3}$) while not being in $\mathcal{L}_{i-1}$ or $\mathcal{L}_{i-2}$.

$$\mathcal{L}_i = N_G(\mathcal{L}_{i-3}) \setminus (\mathcal{L}_{i-1} \cup \mathcal{L}_{i-2}).$$

This whole procedure allows us to assign a generation to every vertex of G. Now it remains to assign a father to every vertex other than r.

Early fathers. For every vertex of $\mathcal{L}_1$, its father is none other than r. Now, observe that vertices of $\mathcal{L}_2$ are at distance 1 from their respective fathers and distance 3 from all other vertices of $\mathcal{L}_1$ (in the tree). Thus, in G, they should be adjacent to all but one vertex of $\mathcal{L}_1$, which will be their father. Therefore, fathers for the first two generations are easily identified.

3.3. Assigning fathers for further generations

Now starts the more involved part of the algorithm. We shall assign fathers to the remaining vertices iteratively, one generation after the other. To do this, we distinguish two types of vertices, the *forced vertices* and the *free vertices*. Let v be a vertex of G. Since we know the generations of all vertices, it is easy to deduce the tree of the third descendants

of v (by looking at the edges of G and the generation partition). Let T_v denote the tree rooted in v and containing all its third descendants. Keep in mind that T_v is a subgraph of G and not a subgraph of the tree that we are constructing.

We partition the vertex set into two sets:

- *Forced vertices* $\mathcal{A} = \{v \in V(G) :$ there exists a third descendant of v that has a nephew or an uncle$\}$,
- *Free vertices* $\mathcal{B} = V(G) \setminus \mathcal{A}$.

Observe that by definition, if a vertex is free, then all its third descendants are also free. Forced vertices are so named because their father can be unequivocally determined (see 3.3.1) while free vertices may have several potential fathers.

We recursively process one generation after the other. Let i be a generation such that all fathers of vertices in lower generations are already fixed (by Paragraph 3.2, $i \geq 3$). We first assign a father to all forced vertices of $\mathcal{L}_i$ (see 3.3.1) and then we seek an assignment of fathers for free vertices of $\mathcal{L}_i$ (see 3.3.2).

3.3.1. Large families help genealogy: forced fathers

Let v be a forced vertex in $\mathcal{A} \cap \mathcal{L}_i$. Observe that if v has neighbours in $\mathcal{L}_{i-1}$ then they should all be its uncles, and their unique brother which is not adjacent (in G) to v must be its father. This is the easiest case for assigning fathers to forced vertices. For the next two paragraphs, we assume that v has no uncle.

Nephews can help too. Now, if v has a nephew u (meaning a neighbour in $\mathcal{L}_{i+1}$), observe that the 3-ancestor of u must be the 2-ancestor of v. This 2-ancestor has a unique son (otherwise v would have an uncle), which in turn has to be the father of v.

From the future to the past. A generalization of the previous paragraph is as follows. Observe that, knowing G and the partition of V into generations, it is easy to know all the third descendants of v. If there is an integer k and a $3k$-descendant of v that has an uncle or nephew (that is a neighbour in $\mathcal{L}_{i+3k-1}$ or $\mathcal{L}_{i+3k+1}$) we can easily find its ancestor in generation $\mathcal{L}_{i-1}$ or $\mathcal{L}_{i-2}$ which in turn must be the 1-ancestor or 2-ancestor of v (recall that v does not have an uncle, thus knowing its 2-ancestor allows us to find its father). Therefore, we can identify and assign the father to v.

3.3.2. Lonely branches: multiple choice

We are left with the last and most problematic case: free vertices. These vertices are such that none of their third descendants has an uncle or nephew. In G, this translates to a bridge between the free vertex and its 3-ancestor, separating v and its third descendants (which form the tree T_v in G) from the rest of G.

After having assigned fathers to forced vertices, we group the remaining unassigned vertices (the free vertices in $\mathcal{B} \cap \mathcal{L}_i$) with respect to their 3-ancestor. With a bit of extra work, we shall solve not only the father assignment for just those vertices, but also for the entire subtree of their descendants.

Some structural observations. Let a be a vertex in $\mathcal{L}_{i-3}$ and X the set of its third descendants that have not received a father after previous inspection (see 3.3.1). We denote X^* as the set X augmented by all the third descendants of vertices in X.

Observation 1. Potential fathers of vertices in X are the 2-descendants of a that are lonely children, whose forced sons are either leaves or do not exist.

Proof. Let x be a vertex in X. Genealogy ensures that its father f must be a 2-descendant of a. Moreover, if f was not a lonely child, then x would have an uncle, and we would have decided its father long ago. Furthermore, if f has already been assigned as a father of some other vertex (let us call it y) which has a non-empty descendance, then x would have at least one nephew, and we would also have decided its father in the previous inspection. □

We let F denote the set of potential fathers for X, and F^* denote the set F augmented with all third descendants of vertices in F.

Observation 2. Vertices in F^* do not have nephews.

Proof. Let f be a potential father for the set X. If one of its third descendants has a nephew, then this nephew is a third descendant of some vertex of generation $\mathcal{L}_i$ which would have been assigned as a son of f while not being a leaf, contradicting Observation 1. □

The father of a potential father is henceforth called a *potential grandfather*.

Observation 3. Given a third descendant of a potential grandfather, all its 3-descendants are brothers.

Proof. Let g be a third descendant of a potential grandfather, and assume that two of its 3-descendants g_1 and g_2 have distinct fathers (call them x_1 and x_2). By Observation 1, x_1 and x_2 are third descendants of vertices in X (since they have non-empty descendance). Now, if g_1 and g_2 share a common grandfather, then x_1 would have a nephew and its ancestor in $\mathcal{L}_i$ would be a forced vertex. If g_1 and g_2 have different grandfathers, then x_1 would have an uncle, which would also qualify its ancestor in $\mathcal{L}_i$ as a forced vertex. This contradicts the definition of X, since its elements must be free vertices. □

Lemma 3. *The father of any vertex in F^* can be determined.*

Proof. We shall prove it by induction on the generation of the third descendants of potential fathers. We know the fathers of potential fathers since the father assignment is known up to generation $i-1$. Now, let f be a third descendant of a potential father at generation $i+3k-1$ and assume that we know the father g' of its 3-ancestor f' (by induction hypothesis). By Observation 3, all 3-descendants of g' are brothers. Therefore:

- if g' has only one 3-descendant g, it must be the father of f,
- if g' has several 3-descendants, all but one must be uncles of f and thus should be adjacent to f in G; the remaining 3-descendant is the father of f.

In this way, we know all mandatory relations between third descendants of potential fathers and potential grandfathers. □

Take a breath and look around. We have fixed a vertex a in $\mathcal{L}_{i-3}$ and X is the set of all its free 3-descendants. In G, each vertex x of X is the root of a tree (its third descendants) which is attached to the rest of G only through the bridge xa. So we have a group of pending trees. Similarly, the third descendants of potential fathers and potential grandfathers form pending structures. What we need is to find a way to accommodate those structures all together. The main thing to observe is that, due to the fact that these structures are pending away from the rest of the graph, if there exist several ways to assign fathers in F^* to all vertices of X^*, then these different ways are locally interchangeable and have no implication on the rest of the procedure (in which case the tree cubic root is not unique).

Fathers of none, fathers of one, fathers of many. Let v be a vertex in F^*. We distinguish the following cases:

- if v has a 3-descendant, then it has a 2-descendant. All its 2-descendants are 3-descendants of the father of v (identified by Lemma 3). By Observation 3, all of them should form a brotherhood. Their common father should be a child of v. Now, observe that if v has two or more non-forced children, then there is an uncle/nephew involving a free child (i.e. a free vertex that is a child of v), which is a contradiction. So v has exactly one free child.
- if v has no 3-descendant, but has a brother v', then v is not in F (it is made only of lonely children). Thus, v' is also in F^* and would have a nephew if v had a child (contradicting Observation 2). Therefore, v has no children. We name these vertices *dead-ends*.
- if v has no 3-descendant and is a lonely child:
  - if the father of v has some 3-descendant, they must all be 2-descendants of v, and v must have exactly one free child (to accommodate the 3-descendants of the father of v).
  - if the father of v has no 3-descendant, then v can have many children as long as they have no 3-descendant. Such a vertex v is called a *fecund vertex*.

Vertices of F^* which require exactly one free child are called *essential vertices*. Observe that fecund vertices may have any number of children (starting from zero), so we focus on essential vertices.

Testing compatibility with essential fathers. Given a vertex x in X^* (at generation j) and an essential vertex v (at generation $j-1$), we want to determine if x can be the unique free son of v (we say f and x are *compatible* if this is the case). To test this, we use induction on the tree structure of T_x.

- if x is a leaf, then it is compatible with v if and only if v itself is a leaf or if its 3-descendants are fecund or dead-ends.
- if x is not a leaf, let $x_1, x_2, \dots, x_k$ be its 3-descendants. We arrange them so that the first p ones are not leaves (that is $x_1, x_2, \dots, x_p$ are not leaves and $x_{p+1}, \dots, x_k$ are leaves). Now consider $v_1, v_2, \dots, v_l$ the essential 3-descendants of v. Now, either v has a fecund 3-descendant or not.
  - if v has no fecund 3-descendants, then each v_i must have exactly one son, and thus, we should find a one-to-one compatibility between $\{x_1, x_2, \dots, x_k\}$ and $\{v_1, v_2, \dots, v_l\}$. This translates as a perfect matching in the bipartite graph where an edge represents compatibility,
  - if v has one or more fecund 3-descendant, then we must have a matching saturating the non-leaves $\{x_1, \dots, x_p\}$ and essential vertices $\{v_1, \dots, v_l\}$ in their compatibility bipartite graph between $\{x_1, \dots, x_k\}$ and $\{v_1, \dots, v_l\}$ (as each essential vertex requires exactly one free child, and each non-leaf x_i requires a son in the tree). The remaining 3-descendants of x (which are leaves) can be safely assigned to any fecund 3-descendant of v.

Finding a perfect matching or matchings that saturate specific sets of vertices in bipartite graphs can be accomplished in polynomial time (see [20, Chapter 16] for an extensive treatment of matchings in bipartite graphs). In fact, our compatibility testing algorithm uses similar concepts to classical tree isomorphism testing algorithms (see [23, Chapter 4]). As we do not seek the most optimal algorithm, we are satisfied with a complexity of $O(n^3)$. Starting from the leaves of T_x, we can determine the compatibility of x with any of its potential fathers in polynomial time. The final step involves checking whether a matching exists between the vertices of X and the potential fathers (once again distinguishing if there is a fecund potential father). Any positive result provides an assignment of fathers for our isolated branches, thereby completing the assignment not only for generation i, but also for all descendants of these vertices in X.

3.4. Time complexity

We now have a polynomial-time algorithm to check whether a given graph G is the exact cube of a tree, thus proving Theorem 1:

1. Test if G is the exact cube of a caterpillar (time $O(n^8)$, Section 2).
2. Fix seven vertices $r, x_1, x_2, x_3, y_1, y_2, y_3$ (n^7 possible choices).
   - 2.1 Determine the generation of every vertex (quadratic through BFS, Section 3.2)
   - 2.2 Determine the fathers for generations 1 and 2 (linear, Section 3.2)
   - 2.3 For each generation i starting from 3:
     - 2.3.1 Determine forced fathers (linear with DFS in T_v, Section 3.3.1)
     - 2.3.2 Solve compatibility for unassigned children (overall $O(n^5)$, Section 3.3.2)
   - 2.4 Routine check that the obtained tree is a cubic root of G (quadratic through LCA methods)
3. Return YES (and the obtained tree) or NO.

Conclusion and open questions

In this paper, we prove that exact cube of trees can be recognized in polynomial time relying on the fact that a few fixed vertices help reveal (almost) the whole structure of the root. Thanks to the work of Bai et al. [2] we know that exact squares of trees are also polynomially recognizable. Thus, it is natural to pose the following open question.

Question 1. Is it true that for any fixed integer p, there exists a polynomial-time algorithm deciding whether a graph G is the exact distance p graph of a tree?

One may question our algorithm's exponent, given that usual graph powers of trees can be recognized in linear time (Chang et al. [5]). Notably, exact cubes of bipartite graphs are bipartite; thus, so are exact cubes of trees. These observations may help improve the time complexity.

Question 2. Is there a more efficient algorithm for recognizing exact cubes of trees than the one proposed? In particular, can this recognition be performed in linear or quadratic time?

Acknowledgements

The research presented here took birth in a meeting in Valparaiso in December 2022, gathering first and fourth author among others under the patronage of ANR project HOSIGRA (ANR-17-CE40-0022). It was then pursued during the visit of fourth author in Clermont-Ferrand thanks to the help of MATH-AMSUD project CODICIS (MATH-AMSUD MATH230035) in July 2024 and of the second author later the same year. Florent Foucaud was supported by the ANR project GRALMECO (ANR-21-CE48-0004) and the International Research Center "Innovation Transportation and Production Systems" of the I-SITE CAP 20-25 (16-IDEX-0001). Henry Echeverría is supported by ANID BECAS/DOCTORADO NACIONAL 21231147. Nikita Manuylenko is supported by the HSE University Basic Research Program. The authors are grateful to Reza Naserasr, Suchismita Mishra and Daniel Quiroz for fruitful conversations on this question.

References

[1] Agnarsson, G., Halldórsson, M.M., 2003. Coloring powers of planar graphs. SIAM Journal on Discrete Mathematics 16, 651–662.
[2] Bai, Y., Cortés, P.P., Naserasr, R., Quiroz, D.A., 2024. Characterizing and recognizing exact-distance squares of graphs. Discrete Mathematics 347.
[3] Bousquet, N., Esperet, L., Harutyunyan, A., de Joannis de Verclos, R., 2019. Exact distance colouring in trees. Combinatorics, Probability and Computing 28, 177–186.
[4] Brešar, B., Gastineau, N., Klavžar, S., Togni, O., 2019. Exact distance graphs of product graphs. Graphs and Combinatorics 35, 1555–1569.
[5] Chang, M.S., Ko, M.T., Lu, H.I., 2015. Linear-time algorithms for tree root problems. Algorithmica 71, 471–495.
[6] Cortés, P.P., 2023. Potencias y potencias exactas: Algoritmos y coloreos. Master's thesis. Universidad de Chile.
[7] Ducoffe, G., 2019. Finding cut-vertices in the square roots of a graph. Discrete Applied Mathematics 257, 158–174.
[8] Foucaud, F., Hocquard, H., Mishra, S., Narayanan, N., Naserasr, R., Sopena, É., Valicov, P., 2021a. Exact square coloring of subcubic planar graphs. Discret. Appl. Math. 293, 74–89.
[9] Foucaud, F., Mishra, S., Narayanan, N., Naserasr, R., Valicov, P., 2021b. Cliques in exact distance powers of graphs of given maximum degree. Procedia Computer Science 195, 427–436. Proceedings of the XI Latin and American Algorithms, Graphs and Optimization Symposium.
[10] Golovach, P., Kratsch, D., Paulusma, D., Stewart, A., 2018. Finding cactus roots in polynomial time. Theory Comput Syst 62, 1409–1426.
[11] Kierstead, H., Yang, D., Yi, J., 2020. On coloring numbers of graph powers. Discrete Mathematics 343, 111712.
[12] Molloy, M., Salavatipour, M.R., 2005. A bound on the chromatic number of the square of a planar graph. Journal of Combinatorial Theory, Series B 94, 189–213. doi:10.1016/j.jctb.2004.12.005.
[13] Motwani, R., Sudan, M., 1994. Computing roots of graphs is hard. Discrete Applied Mathematics 54, 81–88.
[14] Mukhopadhyay, A., 1967. The square root of a graph. Journal of Combinatorial Theory 2, 290–295.
[15] Paoli, M., 1984. Powers of connected graphs and hamiltonicity. Discrete Mathematics 52, 91–99.
[16] Park, J.H., Ihm, I., 2014. Disjoint path covers in cubes of connected graphs. Discrete Mathematics 325, 65–73.
[17] Priyamvada, Panda, B., 2022. Exact square coloring of certain classes of graphs: Complexity and algorithms. Theoretical Computer Science 932, 84–101.
[18] Quiroz, D.A., 2020. Colouring exact distance graphs of chordal graphs. Discrete Mathematics 343, 111769.
[19] Ross, I.C., Harary, F., 1960. The square of a tree. Bell Labs Technical Journal 39, 641–647.
[20] Schrijver, A., 2003. Combinatorial Optimization: Polyhedra and Efficiency. Springer.
[21] Tarjan, R.E., 1979. Applications of path compression on balanced trees. J. ACM 26, 690–715.
[22] Underground, P., 1978. On graphs with hamiltonian squares. Discrete Mathematics 21, 323.
[23] Valiente, G., 2002. Algorithms on trees and graphs. Springer, Berlin; New York.
[24] van den Heuvel, J., Kierstead, H., Quiroz, D.A., 2019. Chromatic numbers of exact distance graphs. Journal of Combinatorial Theory, Series B 134, 143–163.