







Approximate the Clique-Width of a Graph Using Shortest Paths

J. Leonardo González-Ruiz¹(✉) , J. Raymundo Marcial-Romero¹ ,
J. A. Hernández¹ , and Guillermo De-Ita² 

¹ Facultad de Ingeniería, Universidad Autónoma del Estado de México,
Cerro de Coatepec s/n, Ciudad Universitaria, 50100 Toluca, Mexico
{jlgonzalezru, jrmarcialr, xoseahernandez}@uaemex.mx

² Benemérita Universidad Autónoma de Puebla, 4 Sur 104 Centro Histórico C.P.,
72000 Puebla, Mexico
deita@cs.buap.mx

Abstract. In this paper, we present an algorithm to approximate the clique-width of a graph. The proposed approach is based on computing the shortest paths between pairs of vertices. We experimentally show that our proposal approximates the clique-width of simple graphs in polynomial time, while other methods that calculate it in an exact way, transform the problem to SAT, that is well-known as NP-Complete.

Keywords: Graph theory · Clique-width · Algorithm complexity

1 Introduction

The clique-width has been recently studied as an invariant that maintains its properties under graph isomorphism, belonging to the theory of parameterized complexity [1], which is a branch of computational complexity theory that classifies the difficulty of problems according to multiple input or output parameters, specifically in graph theory it measures the difficulty of decomposing a graph into a tree-like structure. Computing the clique-width of a graph consists of construct a finite term which represents the graph.

Courcelle et al. [2] presents a set of four operations to construct such term: 1) the creation of labels for vertices, 2) the disjoint union of graphs, 3) the creation of edges and 4) the re-labeling of vertices. The number of labels used to construct the finite term is commonly denoted by k . The minimum k number used to construct the term, also called k -expression, defines the clique-width. Finding the best combination which minimizes the k -expression is a NP-complete problem [3], furthermore, there is no constant error in the approximation algorithms for the calculation of the clique-width [3]. As the clique-width increases, the complexity of the problem in graphs also increases, in the same way the difficulty to achieve the decomposition of the graph increases, in fact, for some automata

Supported by CONACYT.

that represent certain problems in graphs (according to the main Courcelle's theorem), its computation rapidly consumes memory.

In recent years, the clique-width has been studied in different classes of graphs showing the behavior of this invariant under certain operations. For example, Golumbic et al. [4] showed that for every graph with hereditary distance (a graph in which the distances between all connected induced subgraphs is the same as in the original graph), the clique-width (*cwd* for short) is smaller or equal to 3, so the following problems have a linear solution in this type of graph: minimum dominant set, minimum connected dominant set, minimum Steiner tree, maximum weight clique, domestic number for a fixed k , vertices cover and colorability for a fixed k . Examples can also be found in [5] for graphs with *cwd* 3 or 4.

The importance of classifying these types of graphs according to their *cwd* allows us to have a set of problems in graph theory that allow solutions in polynomial time, which indicates that they are into the tractable problems category. A variant of the problem, which also belongs to the *NP*-complete class, is to decide whether a graph has *cwd* of size k , for a fixed k number. For graphs with *cwd* bounded, in [6] is shown that there is an algorithm in polynomial time ($O(n^2m)$) that recognizes graphs of *cwd* less than or equal to 3. However, the authors refer that the problem remains open for $k \geq 4$. In the other hand, there is a classification for the graphs of $cwd \leq 2$, since the graphs of $cwd = 2$ are precisely the cographs (graphs that can be generated by a graph K_1 by complementation or disjoint union).

Similarly, the same result holds for the other invariant in graphs, the *treewidth* [7], however, the *cwd* is more general in the sense that graphs with a small *tree-width* also have small *cwd*. Algorithms used to calculate the *cwd* in these graphs require a previous certificate that indicates that they have small *cwd*, however, calculating this certificate or deciding if the *cwd* is bounded by a given number is also a complicated problem in the combinatorial sense. In other hand, the *cwd* of a graph with n vertices of degree greater than 2 cannot be approximated by a polynomial algorithm with an absolute error of n^ϵ unless $P = NP$ [3].

Recently, González-Ruiz et al. [8] showed that the *cwd* for Cactus graphs is less or equal to 4 and a polynomial time algorithm was presented that calculates exactly the 4-*expression*. Furthermore, in [9] they studied graphs called Polygonal Trees, which consist of simple cycles joined by at most one edge, showing that the *cwd* of this type of graph is less or equal to 5 and a polynomial time algorithm was presented that calculates the 5-*expression*.

Additionally Heule et al. [10] present a procedure to transform the *cwd* problem to the SAT problem. Its conversion algorithm is polynomial, however, as is well known, the Propositional Satisfaction problem (SAT) remains *NP*-complete. They calculate the *cwd* of relevant graphs in the literature for which the exact *cwd* was not known, using SAT solvers. The graphs considered have known topologies such as Brinkmann, Dodecahedron, Frutch, Kittell, McGee, Desargues, among others.

In this article we present a polynomial algorithm that approximates the *cwd* of simple graphs based on induced paths and show a comparison with the exact results of Heule et al.

2 Preliminaries

All the graphs in this article are simple, that is, finite, without loops or multiple edges, and without direction. A graph is a pair $G = (V, E)$, where V is a set of elements called vertices and E an unordered set of pairs of vertices called edges. An edge e is denoted as uv where u and v are vertices. As we know $|A|$ is used to denote the cardinality of a set A . The degree of a v vertex in a graph G is the number of edges of G incident at v . Also the maximum degree of the vertices of G is denoted by $\Delta(G)$. In other hand a graph is connected if for every partition of its set of vertices into two non-empty sets X and Y , there is at least one edge with an end in X and another end in Y . Additionally a graph G' , whose set of vertices and edges build a subset of vertices and edges of a given graph G , is called a subgraph of G . An abstract graph represents a class of isomorphic graphs.

Let G be a graph and v, w vertices of G , a path from v to w , denoted by $path(v, w)$, is a sequence of edges: $v_0v_1, v_1v_2, \dots, v_{(n-1)}, v_n$ such that $v = v_0, v_n = w$ and v_k is adjacent to $v_{(k+1)}$, for $0 \leq k \leq n$. The length of the path is n . In other way, a simple path is a path where $v_0, v_1, \dots, v_{(n-1)}, v_n$ are all different. A cycle is a nonempty path such that the first and last vertex are identical, and a simple cycle is a cycle in which there is no repeated vertex, except for the first and last. P_n and C_n denote a path and a simple loop respectively, where n denotes the number of vertices.

Dijkstra's algorithm called minimum paths algorithm determines the shortest path given a starting vertex to the rest of the vertices of a given graph [7], the complexity of this algorithm is $O(n^2)$. In the Algorithm 1 we present the pseudo-code of the procedure of E. Dijkstra (1959) to find the shortest path between a pair of vertices.

In other hand, a spanning tree of a connected graph of n vertices is a subset of $n - 1$ edges that forms a tree. Given a graph $G = (V, E)$, let T_G be one of its spanning trees. The edges in T_G are called tree edges, while the edges $E(G) \setminus E(T_G)$ are called fronds. Let $e \in E(G) \setminus E(T_G)$ be a frond edge, the union of the path in T_G between the end points of e with the same edge e form a simple cycle, such a cycle it is called the fundamental of G with respect to T_G . Each frond $e = vw$ fulfills the maximum path contained in the fundamental cycle of which it is part. The set of fundamental cycles of G will be denoted by C . Let L be a countable set of labels. A labeled graph is a (G, γ) pair where γ is a function that maps $V(G)$ to the set L . A labeled graph can also be defined by a triplet $G = (V, E, \gamma)$ and its labeling function is denoted by $\gamma(G)$. We can say that G is D -labelled if D is finite and $\gamma(G) \subseteq D$.

We now introduce the notion of *cwd* (*cwd*, for short). Let \mathcal{C} be a countable set of labels. A *labeled* graph is a pair (G, γ) where γ maps each element of $V(G)$

Algorithm 1. Dijkstra. Procedure that calculates the shortest path between two vertices given a graph G and two origin and destination vertices.

- 1: **procedure** SHORTEST PATH
 - 2: **let** (v_{ini} an origin vertex and v_{fin} a destination vertex of G)
 - 3: Assign each vertex of $V(G)$ a distance value, zero to the vertices between which the path is sought and infinity to the remaining vertices.
 - 4: The source vertex v_{ini} is identified as the current vertex and the other $v_i \in V(G)$ as unvisited vertices.
 - 5: A set of unvisited vertices is created denoted by B
 - 6: **while** $v_{fin} \in B$ **do**
 - 7: For the current vertex its neighbors are considered and the different routes are saved as P^i .
 - 8: The vertices used are removed from the set B of unvisited vertices.
 - 9: **end while**
 - 10: **return** The shortest path P^i with v_{ini} and v_{fin} as endpoints
-

into \mathcal{C} . A labeled graph can also be defined as a triple $G = (V(G), E(G), \gamma(G))$ and its labeling function is denoted by $\gamma(G)$. We say that G is C -labeled if C is finite and $\gamma(G)(V) \subseteq C$. We denote by $\mathcal{G}(C)$ the set of undirected C -labeled graphs. A vertex with label a will be called an a -port.

We introduce the following symbols:

- a nullary symbol $a(v)$ for every $a \in \mathcal{C}$ and $v \in V$;
- a unary symbol $\rho_{a \rightarrow b}$ for all $a, b \in \mathcal{C}$, with $a \neq b$;
- a unary symbol $\eta_{a,b}$ for all $a, b \in \mathcal{C}$, with $a \neq b$;
- a binary symbol \oplus .

These symbols are used to denote operations on graphs as follows: $a(v)$ creates a vertex with label a corresponding to the vertex v , $\rho_{a \rightarrow b}$ renames the vertex a by b , $\eta_{a,b}$ creates an edge between a and b , and \oplus is a disjoint union of graphs.

For $C \subseteq \mathcal{C}$ we denote by $T(C)$ the set of finite well-formed terms written with the symbols $\oplus, a, \rho_{a \rightarrow b}, \eta_{a,b}$ for all $a, b \in C$, where $a \neq b$. Each term in $T(C)$ denotes a set of labeled undirected graphs. Since any two graphs denoted by the same term t are isomorphic, one can also consider that t defines a unique abstract graph.

The following definitions are given by induction on the structure of t . We let $val(t)$ be the set of graphs denoted by t .

If $t \in T(C)$ we have the following cases:

1. $t = a \in C$: $val(t)$ is the set of graphs with a single vertex labeled by a ;
2. $t = t_1 \oplus t_2$: $val(t)$ is the set of graphs $G = G_1 \cup G_2$ where G_1 and G_2 are disjoint and $G_1 \in val(t_1)$, $G_2 \in val(t_2)$;
3. $t = \rho_{a \rightarrow b}(t')$: $val(t) = \{\rho_{a \rightarrow b}(G) | G \in val(t')\}$ where for every graph G in $val(t')$, the graph $\rho_{a \rightarrow b}(G)$ is obtained by replacing every vertex label a by b in G ;
4. $t = \eta_{a,b}(t')$: $val(t) = \{\eta_{a,b}(G) | G \in val(t')\}$ where for every undirected labeled graph $G = (V, E, \gamma)$ in $val(t')$, we let $\eta_{a,b}(G) = (V, E', \gamma)$ such that

$E' = E \cup \{\{x, y\} | x, y \in V, x \neq y, \gamma(x) = a, \gamma(y) = b\}$, e.g. $\eta_{a,b}(G)$ adds an edge between each pair of vertices a and b in G .

For every labeled graph G we let:

$$cwd(G) = \min\{|C| | G \in val(t), t \in T(C)\}.$$

A term $t \in T(C)$ such that $|C| = cwd(G)$ and $G = val(t)$ is called optimal expression of G [11] and written as $|C|$ -expression.

In other words, the cwd of a graph G is the minimum number of different labels needed to construct a vertex-labeled graph isomorphic to G using the four mentioned operations.

As an example we show the computing of cwd for a simple graph Fig. 1 that consists of a cycle of size 4. Firstable in Fig. 2 shows the creation of vertices 3 and 2 of the original graph by the expression $a(3) \oplus a(2)$, next in Fig. 3 the labels $b(1)$ and $b(4)$ are created. Then, in Fig. 4 the disjoint union of the vertices created in steps 1 and 2 is made using the expression $(b(1) \oplus b(4)) \oplus (a(3) \oplus a(2))$. Finally, in Fig. 5 the edges between the labeled vertices are created, resulting in an abstract graph isomorphic to the original by the expression $\eta_{(a,b)}((b(1) \oplus b(4)) \oplus (a(3) \oplus a(2)))$.

As we can see, 2 labels were used to build the cycle of size 4. It is obvious that a single label cannot be used to build the cycle of 4, since edges cannot be created between the same vertices. Therefore, the cwd of a cycle with size 4 is 2.

3 Shortest Path Procedure

In this section we show a procedure based on shortest paths, which allows, throughout the construction, to use a smaller number of labels, thus optimizing the cwd computation.

Let $G = (V, E)$ be a simple graph and \mathcal{C} be the set of fundamental cycles of G . If $C_i, C_j \in \mathcal{C}$ and $E(C_i) \cap E(C_j) \neq \emptyset$ then $C_i \Delta C_j = (E(C_i) \cup E(C_j)) - (E(C_i) \cap E(C_j))$ forms a compound loop, where Δ denotes the symmetric difference operation between the set of edges in both loops.

Let $G = (V, E)$ be a simple graph and C_1 the smallest fundamental cycle, the result of the decomposition of the graph by a spanning tree and its co-tree.

Looking to find an optimal way to construct a graph from the minimum, element of G , i.e. the smallest fundamental cycle, we have proposed the following inductive construction of subgraphs representing together G :

$$G'_1 = C_1.$$

$$G'_n = G'_{(n-1)} \cup \min\{Dijkstra((V(G), E(G) \setminus \bigcup_{i=1}^{n-1} E(G'_i)), v_i, v_j) | v_i, v_j \in G'_{n-1}, v_i \neq v_j\}.$$

Each graph G'_i is induced from the original graph and is contained as shown by the proposition 1.

Proposition 1. $G'_1 \subset G'_2 \dots \subset G'_n = G$.

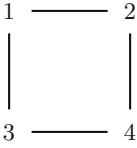


Fig. 1. Simple graph

$b(1)$

$b(4)$

Fig. 3. Step 2: $b(1) \oplus b(4)$

$a(2)$

$a(3)$

Fig. 2. Step 1: $a(3) \oplus a(2)$

$b(1)$

$a(2)$

$a(3)$

$b(4)$

Fig. 4. Step 3: $(b(1) \oplus b(4)) \oplus ((a(3) \oplus a(2)))$

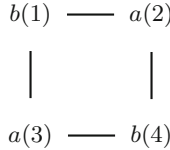


Fig. 5. Step 4: $(\eta_{a,b}(b(1) \oplus b(4)) \oplus ((a(3) \oplus a(2))))$

Proof. $V(G'_i) \subseteq V(G'_{i+1})$ y $E(G'_i) \subseteq E(G'_{i+1}), \forall i, 1 \leq i < n.$ □

Now, to simplify the notation we use:

$$P^i_{v_{ini}, v_{fin}} = \min\{Dijkstra((V(G), E(G) \cup_{i=1}^{n-1} E(G'_i)), v_i, v_j) | v_i, v_j \in G'_{n-1}, v_i \neq v_j\}$$

where v_{ini}, v_{fin} are the initial and final vertex respectively of the shortest path found by Dijkstra's method. Therefore, we can establish the following theorem.

Theorem 1. *Let G a simple graph, and $G'_1 \subset G'_2 \dots \subset G'_n = G$, we can build each $k_{G'_n}$ -expression to calculate the $cwd(G)$, inductively represented by:*

$$G'_1 = C_1.$$

$$G'_n = G'_{n-1} \cup P^{n-1}_{v_{ini}, v_{fin}}.$$

Proof. The proof will be carried out detailing the procedure:

Let L be a set of labels. We construct the k -expression of each induced subgraph G'_i as follows:

- For all $v_i \in G'_1$, take a new $a_i \in L$ and create $a_i(v_i)$.
- The k -expression of the graph G'_1 is: $k(G'_1) = \eta_{a_1, a_1}(\eta_{a_{i-1}, a_i}(a_1(v_1) \oplus \dots \oplus a_i(v_i)))$ where $l = |V(G'_1)|, 1 < i \leq l$.

- Let $K = \{a_i | 1 \leq i \leq l\}$, that is, the labels that have been used to construct the k -expression of the graph G'_{j-1} . The set of labels that can be reused in the construction of the $k_{G'_j}$ -expression starting from the expression $k_{G'_{j-1}}$ are defined as:

$$R = \{a_i \in K | a_i(v_k) \in k_{G'_{j-1}}, \delta(v_k) \in G \setminus E(G'_{j-1}) = 0\}.$$
- Let $a_s \in R$ the label that will be used to rename all other R in the G'_{j-1} graph. The resulting expression after relabeling is: $\rho_{a_t \rightarrow a_s}(k_{G'_{j-1}}) \forall a_t \in R, a_t \neq a_s$.
- For each $v \in P_{v_{ini}, v_{fin}}^{n-1}, v \neq v_{ini} \neq v_{fin}$, take a $a_i \in R$ with $a_i \neq a_s$ if exists, if not, take a new $a_i \in L$ and create $a_i(v_i)$.
- As $P_{v_{ini}, v_{fin}}^{j-1} \setminus \{v_{ini}, v_{fin}\}$ is a path that can be associated with each vertex, considering its adjacency, an index of the sequence $\{1, \dots, r\}$ where r is the number of vertices of the path. With this arrangement create the k -expression: $k_{P^{j-1}} = \eta_{a_{i-1}, a_i}(a_1(v_1) \oplus \dots \oplus a_r(v_r))$ where $1 < i \leq r$.
- Thus $k_{G'_j} = \eta_{a_r, v_{fin}}(\eta_{a_1, v_{ini}}(k_{G'_{j-1}} \oplus k_{P^{j-1}}))$, where $a_1, a_r \in P^{j-1}$ are the two labeled vertices of the ends of P^{j-1} . □

Now for a better understanding, we present the algorithm 2 that constructs the k -expression given a simple graph. As input, the algorithm receives a graph and its decomposition into fundamental cycles. It starts using the smallest size cycle and calculating its k -expression, later the vertices already used in the construction are stored in the set A , using these vertices the shortest path between them is searched, when finding it, compute its k -expression and add the vertices of the path to the set A . This method is repeated until the original graph is built, creating edges between already labeled vertices and releasing labels already covered.

Each step of the 2 algorithm is described as follows. From lines 1 to 2 the algorithm starts with G as input and C_1 is the smallest fundamental cycle of G . In line 3 each vertex of C_1 is added to the set of vertices A . In line 4 the vertices involved in the cycle C_1 are deleted from the original graph G . In line 5 we begin to build the k -expression using different labels for each vertex in C_1 . From lines 6 to 23 we have the main procedure as long as the number of edges in the resulting G is different from 0. In this procedure, in line 7, the shortest path P is found between each vertex of A on the G graph using the well known Dijkstra algorithm, if you have more than one path with the same cardinality then you can choose the last one found. In line 8 the new k -expression of the path P from step 7 is constructed, at this moment we already have the first and last label of P since it is an element of A , each vertex between the first and last on the way must be different. In line 9 the edges involved in the path P mentioned above are deleted from the current G . In line 10 each vertex of the path P found (except the extremes) is added to A .

Now the following condition allows to release labels to be able to reuse them, from 11 to 14 it is compared if the degree of each vertex in A is 0, if it is true, the corresponding labels are released and the vertex of the A set is deleted since has been covered in its entirety. The above is useful to do less operations. The next condition from 15 to 22 is to verify if the elements of A are connected in

the current G , if so, an edge is created using the η operation, which will join the different k -expressions that have already been built. After that, on line 17 the edges found in the last step of the current G are deleted. Finally the same condition is repeated from 18 to 21 as it was done in lines 11 to 14. The *while* instructions will be repeated until all edges are deleted from the graph G .

Furthermore, the complexity of the method presented in this paper is given by two main methods, the first is the well-known Dijkstra Algorithm for simple graphs which is of order $O(n^2)$. In other hand, the proposed method is in the worst case $O(n - 3)$, removing the first minimum cycle of size 3. Therefore, the complexity of these methods is $O(n^3)$.

4 Example

This section shows how our algorithm works when considering the 8-cubic graph, which is illustrated in Fig. 6 also called a trivalent graph whose vertices have degree 3. This graph contains 8 vertices and 12 edges. In Fig. 7 we start with the cycle $C_1 = [4, 6, 5]$ so 3 labels are used and the set $A = \{4, 6, 5\}$. In Fig. 8 the computing of the shortest path is shown with the following steps: the shortest

Algorithm 2. Procedure that calculates a k -expression (G) when G is decomposed into fundamental cycles.

```

1: procedure  $k$ -EXPRESSION( $G$ )
2: let ( $C_i$  a subgraph of  $G$  which is the smallest fundamental cycle of  $G$ )
3: 3. Add each vertex of  $V[C_i]$  in  $\mathcal{A}$ 
4: Delete the edges  $E[C_i]$  from  $G$ 
5: Build the  $k$ -expression of  $C_i$ 
6: while  $|E[G]| \neq 0$  do
7:   Find the shortest path  $P$  using Dijkstra's algorithm between each vertex of  $\mathcal{A}$ 
   in the graph  $G$  {If you have more than one path of the same size, the last one
   found is taken}
8:   Build the  $k$ -expression of  $P$ 
9:   Delete from  $G$  the edges of  $P$ 
10:  Add to each vertex of  $V[P]$  in  $\mathcal{A}$ 
11:  if For each  $a_i \in \mathcal{A}$ , the degree of  $a_i$  in  $G$  is 0 then
12:    Free a label
13:    Delete vertex  $a_i$  of  $\mathcal{A}$ 
14:  end if
15:  if the actual elements of  $\mathcal{A}$  are connected in  $G$  then
16:    Build the  $k$ -expression using  $\eta$  operator
17:    Delete the previous created edges from  $G$ 
18:    if for each  $a_i \in \mathcal{A}$ , the degree of  $a_i$  in  $G$  is 0 then
19:      Free the label
20:      Delete the vertex  $a_i$  of  $\mathcal{A}$ 
21:    end if
22:  end if
23: end while

```

path between the vertices of A is calculated, the resulting path is $P = [5, 1, 7, 6]$ and 2 more labels are used, as a result $A = \{4, 5, 6, 1, 7\}$, now 2 labels can be released, the ones corresponding to vertex 6 and 5, one of them will be left as a residual label for the entire graph and we have a free one, as a result $A = \{4, 1, 7\}$, 5 labels have been used and 1 remains free to use. Later in Fig. 9 the following algorithm computing is shown, the shortest path is found $P = [7, 8, 2, 1]$, the free label is used and a new one, and the vertices are added to the set A , so far $A = \{4, 1, 7, 8, 2\}$, at this moment labels 7 and 1 can be released, so there are 2 free labels left and the set would be $A = \{4, 8, 2\}$. In Fig. 10 the following path found is shown $P = [8, 3, 2]$, one of the two available labels is used and the set is as follows: $A = \{4, 8, 2, 3\}$, labels 8 and 2 can be released, the resulting set would be: $A = \{4, 3\}$. Finally, in Fig. 11 an edge is created between 4 and 3, to later release these two vertices leaving the set as $A = \{\}$, ending the algorithm and resulting in the $cwd(8 \text{ cubic graph } 2) \leq 6$.

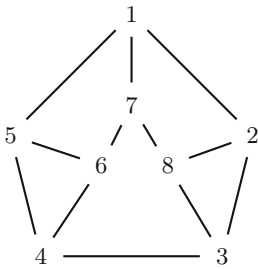


Fig. 6. $8 \text{ cubic graph } 2$

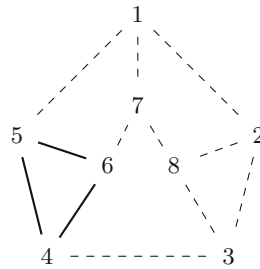


Fig. 7. $C_1 = [4, 6, 5]$

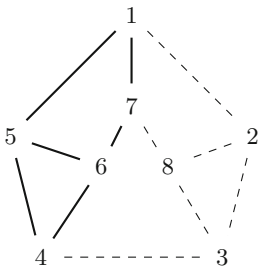


Fig. 8. $P = [5, 1, 7, 6]$

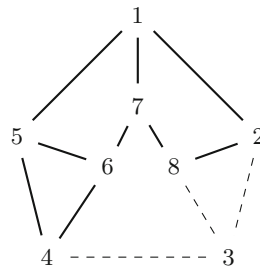


Fig. 9. $P = [7, 8, 2, 1]$

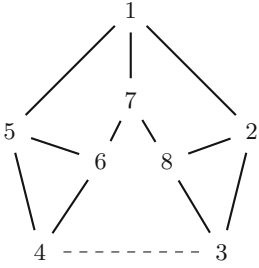


Fig. 10. $P = [8, 3, 2]$

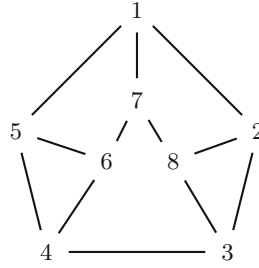


Fig. 11. $\eta_{(4,3)}$

5 Conclusions

This article presents a proposal to approximate the *cwd* of simple graphs. The proposed algorithm is based on the classic Dijkstra algorithm together with the calculation of the shortest paths of induced graphs. To get a conclusion about the efficiency of our algorithm, the results reported by Heule [10] were considered and compared with those generated by our proposal. In Table 1, is shown the name of the graph with its number of vertices and edges, followed by the exact result of Heule et al. and the result of our proposal.

On the other hand, the error between an exact method and an approximation method is given by an equation in [3] as follows: $|A(G) - cwd(G)| \leq n^\epsilon$, where $A(G)$ is the result of the *cwd* using an approximation algorithm (which is our case), the *cwd*(G) is the exact result, n is the number of vertices and finally ϵ is

Table 1. Comparison between approach algorithm and exact *CWD*

Graph	$ V $	$ E $	Heule	Our proposal	Error
Brinkmann	21	42	10	10	0%
Desargues	20	30	8	10	23.13%
Dodecahedron	20	30	8	8	0%
Errera	17	45	8	8	0%
Frutch	12	18	5	7	27.8%
Kittell	23	63	8	9	$\approx 0\%$
McGee	24	36	8	10	21.8%
Paley13	13	39	9	10	$\approx 0\%$
Pappus	18	27	8	9	$\approx 0\%$
Petersen	10	15	5	7	30.1%
Poussin	15	39	7	8	$\approx 0\%$
Robertson	19	38	9	10	$\approx 0\%$
Shirkhande	16	48	9	11	25%

the error, clearing ϵ gives the error of the last column. As can be seen in Table 1, in 8 of the 13 cases the *cwd* with an $\approx 0\%$ of error was obtained, having in the worst case an error of 30.1% and in the average case an error of 9.8%.

The advantage of this algorithm is that its execution time is polynomial so the approximation has a complexity of the order $O(n^3)$ where n is the number of vertices of the input graph.

References

1. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Monographs in Computer Science. Springer, New York (1999)
2. Courcelle, B., Engelfriet, J., Rozenberg, G.: Handle-rewriting hypergraph grammars. *J. Comput. Sys. Sci.* **46**(2), 218–270 (1993)
3. Fellows, M.R., Rosamond, F.A., Rotics, U., Szeider, S.: Clique-width is np-complete. *SIAM J. Discrete Mathe.* **23**(2), 909–939 (2009)
4. Golombic, M.C., Rotics, U.: Graph-theoretic concepts in computer science. In: 25th Proceedings of the International Workshop on Clique-Width of Perfect Graph Classes (WG1999) Ascona, Switzerland, June 17–19, 1999, pp. 135–147. Springer, Berlin (1999). <https://doi.org/10.1007/978-3-642-11409-0>
5. Langer, Alexander, Reidl, Felix, Rossmanith, Peter, Sikdar, Somnath: Practical algorithms for MSO model-checking on tree-decomposable graphs. *Comput. Sci. Rev.* **1314**, 39–74 (2014)
6. Derek, G. Corneil, M.H., Lanlignel, J.-M., Reed, B., Rotics, U.: Polynomial-time recognition of clique-width ≤ 3 graphs. *Discrete Appl. Math.* **160**(6), 834–865 (2012)
7. Bondy, J.-A., Murty, U.S.R.: Graph Theory. Graduate Texts in Mathematics. Springer, New York, London (2007)
8. Leonardo González-Ruiz, J., Raymundo Marcial-Romero, J., Hernández-Servín, J.A.: Computing the clique-width of cactus graphs. *Electronic Notes in Theoretical Computer Science*. In: Tenth Latin American Workshop on Logic/Languages, Algorithms and New Methods of Reasoning (LANMR), vol. 8, pp. 47–57 (2016)
9. Leonardo González-Ruiz, J., Raymundo Marcial-Romero, J., Hernández, J.A., De Ita, C.: Computing the clique-width of polygonal tree graphs. In: Pichardo-Lagunas, O., Miranda-Jiménez, S. (eds.) *Advances in Soft Computing*, pp. 449–459, Springer International Publishing, Cham (2017)
10. Marijn, J., Heule, H., Szeider, S.: A SAT Approach to clique-width. In: *ACM Transactions on Computational Logic* pp. 318–334. Springer, Berlin (2013)
11. Courcelle, B., Olariu, S.: Upper bounds to the clique width of graphs. *Discrete Appl. Math.* **101**, 77–114 (2000)