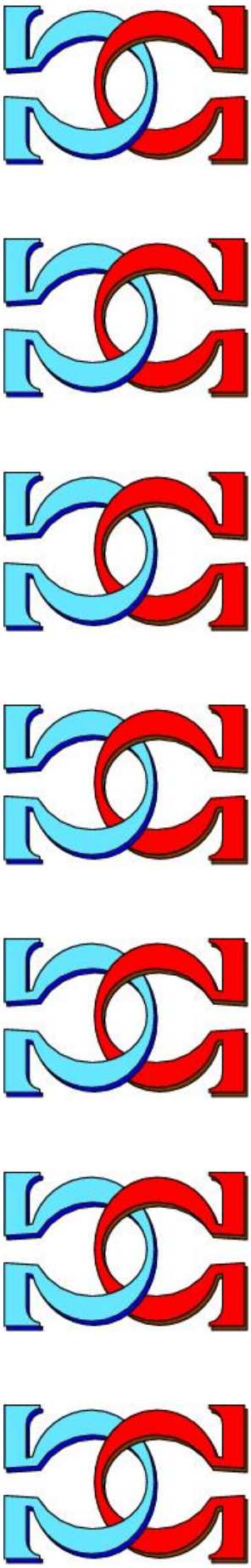


CDMTCS
Research
Report
Series

A New Linear-Time Dominating Number Algorithm for Graphs of Bounded Pathwidth

Michael J. Dinneen
Aisha J.L. Fenton
Department of Computer Science,
University of Auckland,
Auckland, New Zealand

CDMTCS-329
July 2008

Centre for Discrete Mathematics and
Theoretical Computer Science

A New Linear-Time Dominating Number Algorithm for Graphs of Bounded Pathwidth

Michael J. Dinneen and Aisha J.L. Fenton
Department of Computer Science
University of Auckland
Auckland, New Zealand

Abstract

We present a simple algorithm for determining the minimum dominating number of any graph of bounded pathwidth. The algorithm has running time $O(3^{t+1}n)$ where n is the size of the graph and t is a fixed pathwidth bound. We also present an implementation in Python of this algorithm extended to handle graphs of bounded treewidth.

1 Introduction

This paper describes a simple linear-time algorithm for determining the minimum dominating number of a graph of bounded pathwidth. The minimum dominating number problem asks for a graph G, what is the minimum $S \subseteq V(G)$ such that every vertex is either in S or a neighbour of a vertex in S. The minimum dominating number of a graph is a well known problem in graph theory and has applications in many common engineering problems.

In general the minimum dominating set problem has been shown to be $\mathcal{NP}$-complete [8]. As many $\mathcal{NP}$-hard problems have wide applicability and underlie many important real-world engineering and optimization problems, tractable solutions have been sought for restricted problem domains. A common approach for $\mathcal{NP}$-hard problems within the graph theory setting has been to restrict the input type to graphs of some fixed property or parameter; the aim being to produce solutions with polynomial running time for the restricted class of graphs. This is the approach we follow here.

Arnborg and Proskurowski gave a linear-time algorithm for determining the minimum dominating number for graphs of bounded treewidth [1]. The algorithm

we present here shares some similarities with those presented by Arnborg and Proskurowski but we believe our algorithm is easier to use and in practice will be more efficient for small pathwidths. Additionally the algorithm presented in this paper is fully specified.

The rest of this paper is organized as follows: Section 2 introduces the graph theory setting; Section 3 provides some results and notations that we use throughout the rest of this paper; Section 4 presents a high-level sketch of the Minimum Dominating Set (MDS) algorithm with the aim to illustrate its key concepts; Section 5 provides the full details of the MDS algorithm and an example of its operation.

2 Background

In this section we provide a short overview to those areas of the graph theory setting that are used throughout the paper. We assume familiarity with common graph-theoretic notations; reference can be made to [5] for an overview of the graph theory setting and common notations if required.

2.1 Minimum dominating number

The minimum dominating set problem has a long history and can been seen as a generalization of the classic chess problem, Queens Domination, that dates back to at least the publication of [9] in 1862.

Definition 1. A *dominating set* of a graph G is a set $S \subseteq V(G)$ such that for every $v \in (V(G) \setminus S)$, v is in the neighbourhood of a member of S. The *dominating number* of a graph G is the minimum cardinality over all S. Let $\gamma(G)$ denote the dominating number of G, and $\Lambda(G)$ denote the set of dominating sets of G that are minimal.

Example 2. The following finite graph G has the dominating set $S = \{a, e\}$. Observe that S is minimal as no smaller dominating set exists for G. Therefore we say that the minimum dominating number of $\gamma(G) = 2$.

2.2 Graphs of bounded pathwidth

The important notions of *pathwidth* and *treewidth* were introduced by Robertson and Seymour [3, 4]; they codify the notion of how closely a graph's structure resembles that of a path and tree respectively. We use pathwidth within this

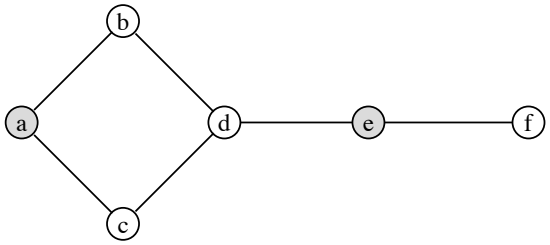


Figure 1: G with dominating set

paper to restrict the type of graphs accepted by our algorithm to those graphs with pathwidth at most k for some $k \in \mathbb{N}$.

Definition 3. A *path decomposition* of a graph G is a sequence $\{X_1, X_2, X_3, ..., X_r\}$ of subsets of $V(G)$ such that the following are satisfied:

1. $\bigcup_{1 \leq i \leq r} X_i = V(G)$
2. For every edge $uv \in E(G)$ there exists an $X_i, 1 \leq i \leq r$ such that $u \in X_i$ and $v \in X_i$
3. For $1 \leq i < j < k \leq r$, $X_i \cap X_k \subseteq X_j$

Definition 4. The *pathwidth* of a path decomposition $[X_1, X_2, X_3, ..., X_r]$ is equal to $max_{1 \leq i \leq r}|X_i| - 1$. And the pathwidth of a graph G is defined as the minimum pathwidth over all G's path decompositions.

Example 5. Figure 2 depicts a graph with one of its path decompositions, $\{X_1, X_2, X_3, X_4, X_5\}$. This path decomposition has a maximum cardinality of 3 over all X_i. As the path decomposition is minimal, we conclude that the graph has pathwidth is 2.

2.3 Graph t-parse encoding

The input to our algorithm (henceforth called the 'MDS algorithm') is encoded as a t-parse; a t-parse is an encoding of a bounded pathwidth graph into a string representation [6, 7]. Although not strictly necessary for the operation of the MDS algorithm it has the convenient property of ordering the elements of a graph into a sequence $[e_1, e_2, ..., e_l]$ where $e_i \in V(G) \cup E(G)$ and consequently orders the set of partial solutions emitted by the MDS algorithm.

It is shown in [6] that a graph G has a t-parse encoding if and only if G has pathwidth at most t.

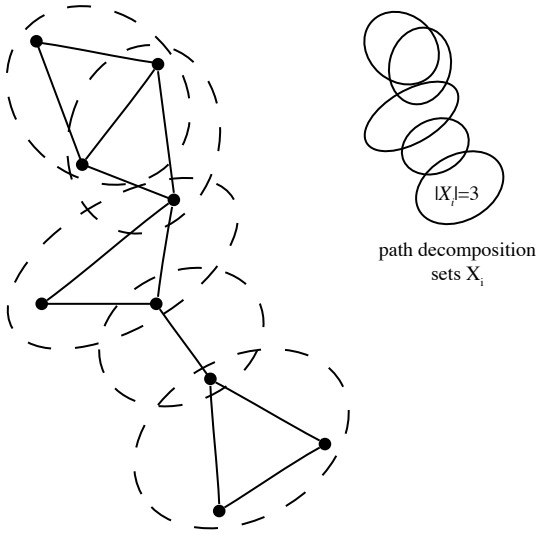


Figure 2: Graph G and path decomposition

Definition 6. We start by defining the language Σ_t^*.

$$\Sigma_t = V_t \cup E_t$$
$$V_t = \{\textcircled{0}, \textcircled{1}, ..., \textcircled{t}\}$$
$$E_t = \{\boxed{i\ j} : i, j \in V_t, i \neq j\}$$

Each character of Σ_t represents a distinct operation that is performed against a given graph. These operations are defined as follows for the graph G:

$\textcircled{i}$ $G = G + u$ and give vertex u the label i. If there is an existing vertex $v \in V(G)$ such that v is labeled i, then remove the label from v first (leaving v unlabeled).

$\boxed{i\ j}$ $G = G + ij$ for $i, j \in V(G)$ that are labeled i, j respectively.

Definition 7. A *parse* is a string of characters in Σ_t^* such that any edge operator $\boxed{i\ j}$ does not appear before the vertex operators $\textcircled{i}$ and $\textcircled{j}$ within the string (that is an edge does not appear before its corresponding vertices have been defined for the first time).

Definition 8. A t-parse is a parse $[o_1, o_2, ..., o_l]$ such that all vertex operators $\textcircled{0}, \textcircled{1}, \ldots, \textcircled{t}$ appear at least once within $[o_1, o_2, ..., o_l]$.

The bounded pathwidth graph that is represented by a t-parse is called its underlying graph. The underlying graph of a t-parse can be found by using the following procedure: start with the empty graph $G = \emptyset$, scan left to right along the t-parse performing each operation as represented by each character T_i. The final state of G, after all operations have been performed is the t-parse's underlying graph.

Definition 9. For a given t-parse T and integer $1 \leq i \leq |T|$, let G_i represent the underlying graph of the t-parse given by the prefix T_1^i.

Informally, a t-parse can simply be understood as a data structure that represents a bounded pathwidth graph. For simplicity of notation, henceforth, we will use G_i to denote both the string T_1^i (that is the data structure) and the underlying graph G_i – which exact meaning will be clear from the context of its use.

Observant readers will have noticed that as a t-parse's underlying graph is built that only the last $t + 1$ vertices are labeled and therefore 'visible' to subsequent operations (e.g. edge operations). We define the active vertex set of a graph as follows:

Definition 10. For a given t-parse G let the active boundary of G, denoted $\partial(G)$, be equal to the currently labeled vertices of G. In other words the last $t+1$ vertices that were added to the graph.

Example 11. The following example depicts G_5 and $G_{|T|}$ for the t-parse $T = [\textcircled{0}, \textcircled{1}, \textcircled{2}, \boxed{0\ 1}, \boxed{0\ 2}, \textcircled{2}, \boxed{0\ 1}, \boxed{0\ 2}\ ]$. In each case the vertices of the graph's currently active boundary are shown in grey.

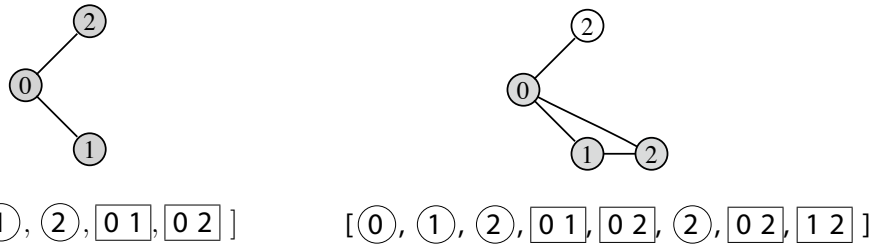


Figure 3: G_5 and $G_{|T|}$

3 Dominating Set Properties

In this section we introduce some properties of minimum dominating sets along with some results that we'll use throughout the rest of the paper.

Definition 12. For a given dominating set $D \in \Lambda(G)$ and vertex $v \in D$ let $\mathcal{U}(v)$ equal the number of vertices that are uniquely dominated by v. In other words $\mathcal{U}(v)$ equals the number of v's neighbours that have no dominators in their neighbourhood, except for v. If v does not uniquely dominate any of its neighbours then $\mathcal{U}(v) = 0$.

Lemma 13. *If a vertex $v \in D$, for some $D \in \Lambda(G)$, has $\mathcal{U}(v) = 0$, then none of v's neighbours are within D, that is $N(v) \cap D = \emptyset$.*

Proof. Assume that v does have a dominator $x \in D$ in its neighbourhood. By definition the vertices $V(G) \setminus \{v\}$ are dominated by the set $D \setminus v$. Now if the dominator x is in v's neighbours then x can also dominate v. Therefore we are able to construct a smaller dominating set $D \setminus \{v\}$ to dominate G, thus contradicting the minimality of D. □

Lemma 14. *If a vertex $v \in D$, for $D \in \Lambda(G)$, has $\mathcal{U}(v) = 0$ then the dominating set $D' = D \setminus \{v\}$ is minimal for $G - v$ (that is $D' \in \Lambda(G - v)$).*

Proof. By the definition of $\mathcal{U}(v) = 0$, the vertices $D \setminus v$ dominate $V(G) \setminus \{v\}$, so D' is a dominating set of $G - v$.

We now show that D' is also minimal. Assume for sake of contradiction that there exists a dominating set $C \in \Lambda(G - v)$ with $|C| < |D'|$. Consequently we are able to construct a dominating set $C' = C \cup \{v\}$ for the graph G. Now observe that $|C'| = |C| + 1$ giving us $|C'| \leq |D'| < |D|$. As we have contradicted the minimality of D, we conclude that $D' \in \Lambda(G - v)$. □

Corollary 15. *If a vertex $v \in D$, for $D \in \Lambda(G)$, has $\mathcal{U}(v) = 0$, then $\gamma(G - v) = \gamma(G) - 1$.*

Definition 16. For a given subset of vertices $s \subseteq V(G)$, let $\gamma_s(G)$ denote the cardinality of the smallest dominating set of the graph G that includes s as a subset. Likewise, let $\Lambda_s(G)$ denote the set of smallest dominating sets of a graph G that include s as a subset.

Lemma 17. *For a given vertex $v \in V(G) | v \notin s$, $\mathcal{U}(v) = 0$ if and only if $\gamma_s(G) > \gamma_s(G - v)$.*

Proof. That $\mathcal{U}(v) = 0 \implies \gamma_s(G) > \gamma_s(G - v)$ follows from Corollary 15.

Conversely we show $\gamma_s(G) > \gamma_s(G - v) \implies \mathcal{U}(v) = 0$ by construction. Given $v \in V(G)$ such that there exists a $D \in \Lambda_s(G - v)$ with $D < \gamma_s(G)$, we are able to construct a minimum dominating set for the graph G with $\mathcal{U}(v) = 0$ as follows: let $D' = D \cup \{v\}$. Since D dominates all vertices $V(G - v)$ and $s \subseteq D$ we conclude $D' \in \Lambda_s(G)$ with $\mathcal{U}(v) = 0$. □

Lemma 18. *For a given subset of vertices $r \subseteq V(G)$, $|\gamma_{s\cup r}(G)| = |\gamma_s(G)|$ if and only if there exists a $D \in \Lambda_s(G)$ such that $r \subseteq D$.*

Proof. If there exists a $D \in \Lambda_s(G)$ such that $r \subseteq D$ then observe that D is also a member of $\Lambda_{s\cup r}(G)$, implying that $|\gamma_{s\cup r}(G)| = |\gamma_s(G)|$. Likewise if $|\gamma_{s\cup r}(G)| = |\gamma_s(G)|$ then for $D' \in \Lambda_{s\cup r}(G)$, D' is a member of $\Lambda_s(G)$. □

4 Algorithm Sketch

We start by providing a high-level sketch of our MDS algorithm. We deliberately leave out some details and make several assumptions at this stage with the aim to provide a clear illustration of the key concepts used by the algorithm.

The MDS algorithm follows a dynamic programming approach. A bounded pathwidth graph, encoded as a t-parse, is provided as input. The algorithm scans through the t-parse from left to right, producing a partial solution for each underlying graph G_i. State information is maintained for the previous partial solution and is used to calculate the next partial solution. The final solution is given after processing the last operator within the t-parse.

4.1 Partial solution procedure

We now describe the procedure used by the MDS algorithm to derive the minimum dominating number of a graph G_{i+1}. The procedure uses the state information from the previous partial solution (of graph G_i) to calculate the minimum dominating number for G_{i+1}. We leave it to the next section to fully detail how the required state information for each partial solution is gathered; in particular we assume knowledge of G_i's dominating sets.

Let o_{i+1} be the operation at index $i + 1$ within the t-parse G.

1. **If** o_{i+1} is a vertex operator $\textcircled{v}$ **then** $\gamma(G_{i+1}) = \gamma(G_i) + 1$.
2. **If** o_{i+1} is an edge operator $\boxed{u\ v}$, such that for all $D \in \Lambda(G_i)$, $\{u, v\} \cap D \neq \{u, v\}$ is true **then** $\gamma(G_{i+1}) = \gamma(G_i)$.

3. **If** o_{i+1} is an edge operator $\boxed{u\ v}$, such that there exists a $D \in \Lambda(G_i)$ with $\{u, v\} \cap D = \{u, v\}$ **then**

   (a) **If** $\mathcal{U}(v) = 0$ or $\mathcal{U}(u) = 0$ within a dominating set D **then** $\gamma(G_{i+1}) = \gamma(G_i) - 1$.

   (b) **Otherwise**, $\gamma(G_{i+1}) = \gamma(G_i)$

The above procedure is now re-stated as a series of lemmas so that we may present the corresponding proofs of the statements.

Lemma 19 (Rule 1)**.** *If $G_{i+1} = G_i + v$ where v is an isolated vertex then $\gamma(G_{i+1}) = \gamma(G_i) + 1$.*

Proof. This is trivially true, as an isolated vertex cannot be dominated by any other vertices. □

Lemma 20 (Rule 2)**.** *If $G_{i+1} = G_i + uv$ such that for all $D \in \Lambda(G_i)$, $D \cap \{u, v\} \neq \{u, v\}$ then $\gamma(G_{i+1}) = \gamma(G_i)$.*

Proof. We start by showing that $\gamma(G_{i+1}) \geq \gamma(G_i)$. Assume for sake of contradiction that there exists a $D' \in \Lambda(G_{i+1})$ with $|D'| < |D|$. It therefore must be possible to construct a dominating set D'' of G_i as follows:

$$D'' = \begin{cases} D' & \text{if } D' \cap \{u, v\} = \emptyset \text{ or } D' \cap \{u, v\} = \{u, v\} \\ D' \cup \{u\} & \text{if } D' \cap \{u, v\} = \{v\} \\ D' \cup \{v\} & \text{if } D' \cap \{u, v\} = \{u\} \end{cases}$$

However examination of each value of D'' shows that we have a contradiction! If $D'' = D'$ then $|D''| = |D'| < |D|$ thus contradicting the minimality of D. Moreover if $D'' = D' \cup \{u\}$ or $D'' = D' \cup \{v\}$ then we have constructed a dominating set such that $D'' \in \Lambda(G_i)$ and both vertices u, v are present together in the dominating set, hence contradicting our definition of how the vertices u, v were to be chosen.

It is trivially the case that $\gamma(G_{i+1}) \leq \gamma(G_i)$ as adding an edge cannot increase the dominating number of a graph. Therefore we conclude that $\gamma(G_{i+1}) = \gamma(G_i)$ □

Lemma 21 (Rule 3.a)**.** *If $G_{i+1} = G_i + uv$ and there exists a $D \in \Lambda(G_i)$ such that $D \cap \{u, v\} = \{u, v\}$ and $\mathcal{U}(u) = 0$ or $\mathcal{U}(v) = 0$ then $\gamma(G_{i+1}) = \gamma(G_i) - 1$.*

Proof. We start by showing $\gamma(G_{i+1}) \leq \gamma(G_i) - 1$ by construction. Without loss of generality assume $\mathcal{U}(v) = 0$, and let $D' = D \setminus \{v\}$. By definition D' must dominate all vertices $V(G_i) \setminus \{v\}$. Recall that $u \in D'$, therefore the addition of the edge between u and v must allow D' to dominate all vertices $V(G_{i+1})$.

We now show the opposite direction: $\gamma(G_{i+1}) \geq \gamma(G_i) - 1$. Assume for sake of contradiction that there exists a $D' = \Lambda(G_{i+1})$ such that $|D'| < |D| - 1$; we are then able to construct a $D'' \in \Lambda(G_i)$ as follows:

$$D'' = \begin{cases} D' & \text{if } D' \cap \{u, v\} = \emptyset \text{ or } D' \cap \{u, v\} = \{u, v\} \\ D' \cup \{u\} & \text{if } D' \cap \{u, v\} = \{v\} \\ D' \cup \{v\} & \text{if } D' \cap \{u, v\} = \{u\} \end{cases}$$

Observe that $|D''| \leq (|D'| + 1) < |D|$, hence we have contradicted the minimality of D and we conclude that $\gamma(G_{i+1}) = \gamma(G_i) - 1$ □

Lemma 22 (Rule 3.b)**.** *If $G_{i+1} = G_i + uv$ and for all $D \in \Lambda(G_i)$ such that $D \cap \{u, v\} = \{u, v\}$ the following is true, $\mathcal{U}(u) > 0$ and $\mathcal{U}(v) > 0$, then $\gamma(G_{i+1}) = \gamma(G_i)$.*

Proof. It is trivially the case that $\gamma(G_{i+1}) \leq \gamma(G_i)$ as the addition of an edge cannot increase the dominating number of a graph. Now consider the opposite direction: $\gamma(G_{i+1}) \geq \gamma(G_i)$. Assume for sake of contradiction that there exists a $D' = \Lambda(G_{i+1})$ with $|D'| < |D|$; we are then able to construct a $D'' \in \Lambda(G_i)$ as follows:

$$D'' = \begin{cases} D' & \text{if } D' \cap \{u, v\} = \emptyset \text{ or } D' \cap \{u, v\} = \{u, v\} \\ D' \cup \{u\} & \text{if } D' \cap \{u, v\} = \{v\} \\ D' \cup \{v\} & \text{if } D' \cap \{u, v\} = \{u\} \end{cases}$$

However examination of each value of D'' shows that we have a contradiction. If $D'' = D'$ then $|D''| < |D|$ thus contradicting the minimality of D. Furthermore if $D' \cap \{u, v\} = \{u\}$ then D' will dominate all vertices $V(G_i) \setminus \{v\}$. Hence $D'' = D' \cup \{v\}$ is a minimum dominating set for G_i with $v \in D''$ and $\mathcal{U}(v) = 0$, thus contradicting that $\mathcal{U}(v) > 0$ must be true. A similar argument can be made for $D'' = D' \cup \{u\}$, hence we conclude that $\gamma(G_{i+1}) = \gamma(G_i)$. □

5 Main Results

We are now able to present the full details of the MDS algorithm, building upon the conceptual framework given in Section 4.

5.1 MDS algorithm

The algorithm accepts a t-parse encoding G of a bounded pathwidth graph as input and returns the minimum dominating number of the underlying graph. The MDS algorithm uses the following procedure to calculate a return value.

For each prefix G_i, for $0 \leq i \leq |G|$, we calculate a set of state information. Let the tuple (s, q, i) index this state information, such that: $q \in Q$, $s \in S_q$

$$Q = \mathcal{P}(\{0, 1, ..., t\}) \qquad\qquad S_q = \mathcal{P}(\{0, 1, ..., t\} \setminus q)$$

Next we assign each value of the tuple (s, q, i) an integer value called its state entry, as follows:

$$(s, q, i) = \gamma_s(G_i - q)$$

In other words, the state information of each partial solution is comprised of the minimum cardinalities of the dominating sets of the graphs $G_i - q$ with the vertices s included within them. Both s and q are subsets of the graph's active boundary.

The state entry values for each new partial solution $i + 1$ is calculated using the *Calculate State Entry* procedure given below. The procedure is provided with the state information of G_i and returns a value for each tuple (s, q).

We now prove Theorem 23 and show that the MDS algorithm correctly calculates the minimum dominating number of a t-parse G. Much of the proof has already been provided in Section 4, so we build upon lemmas 19, 20, 21 and 22 and show how the information we assumed for these lemmas is obtained from the state information we keep for each partial solution.

Theorem 23. *The MDS algorithm gives the minimum dominating number of a graph G, where G has bounded pathwidth.*

Proof. We justify the steps of the algorithm below.

Line 4 Follows directly from Lemma 19.

Line 6 Let j be the label represented by the vertex operator $\textcircled{j}$. As a vertex v already exists with label j, v is removed from the active boundary of G_{i+1} and is therefore not excluded from the graph anymore. Hence our return value is equal to the value indexed by $(s, q \setminus \{j\}, i)$.

Line 11 As either u or v is excluded from the current graph, the edge operation defined by o_{i+1} is ignored.

Algorithm 1 Calculate State Entry (s, q) for $i + 1$

Require: $q \in Q$ {Current boundary vertices removed from graph}
Require: $s \in S_q$ {Current boundary vertices included in dominating set}
Require: o_{i+1} {Current operator under consideration}
Require: $Prev[Q][S_q]$ {State info for partial solution G_i, indexed by Q, S_q}

if o_{i+1} is vertex operator **then**
  Let v equal the vertex label represented by o_{i+1}
  if $v \notin q$ **then**
    Return $Prev[q][s] + 1$
  else
    Return $Prev[q \setminus \{v\}][s]$ {Since v is replaced}
  end if
else if o_{i+1} is edge operator **then**
  Let $\{u, v\}$ equal the edge represented by o_{i+1}
  if $\{u, v\} \cap q \neq \emptyset$ **then** {Is u or v in q?}
    Return $Prev[q][s]$
  else {Therefore neither are in q}
    if $\{u, v\} \cap s = \{u, v\}$ **then** {Are both u,v in s?}
      Return $Prev[q][s]$
    else if $\{u, v\} \cap s = \emptyset$ **then** {Are both u,v not in s}
      if $Prev[q][s \cup \{u, v\}] == Prev[q][s]$ **then** {Are both u,v in some D.S.?}
        if $(Prev[q \cup u][s \cup v] < Prev[q][s \cup v]) \vee (Prev[q \cup v][s \cup u] < Prev[q][s \cup u])$ **then** {Are either $\mathcal{U}(u) = 0$ or $\mathcal{U}(v)$=0 when the other is dominator?}
          Return $Prev[q][s] - 1$
        else
          Return $Prev[q][s]$
        end if
      else {Therefore one of u, v must be dominated by another vertex}
        Return $Prev[q][s]$
      end if
    else {One of $(u, v) \in s$}
      Assume $u \in s$ and $v \notin s$ without loss of generality
      if $Prev[q \cup v][s] < Prev[q][s]$ **then** {Does $\mathcal{U}(v) = 0$?}
        Return $Prev[q][s] - 1$
      else
        Return $Prev[q][s]$
      end if
    end if
  end if
end if

Line 14 From the conditions given on line 13 both $u, v \in s$ and therefore must be included within any dominating set of G_{i+1}. Assume for sake of contradiction that there exists a $D \in \Lambda_s(G_{i+1})$ with $|D| < \gamma_s(G_i)$. However D is also a member of $\Lambda_s(G_i)$ as D dominates all $V(G_i)$ – since $u, v \in D$ – hence we have a contradiction.

Line 18 From the conditions given on lines 16 and 17 – and applying Lemmas 18 and 17 respectively – we conclude that there exists a $D_s \in \Lambda_s(G_i)$ where $u, v \in D_s$ and that either $\mathcal{U}(u) = 0$ or $\mathcal{U}(v) = 0$. Applying Lemma 21 directly we conclude that $\gamma_s(G_{i+1}) = \gamma_s(G_i) - 1$.

Line 20 From the conditions given on lines 16 and 17 – and applying Lemmas 18 and 17 respectively – we conclude that for all $D_s \in \Lambda_s(G_i)$ such that $u, v \in D_s$ the following is true, $\mathcal{U}(u) > 0$ and $\mathcal{U}(v) > 0$. Applying Lemma 22 directly we conclude that $\gamma_s(G_{i+1}) = \gamma_s(G_i)$.

Line 23 From the condition given on line 16 – and applying Lemma 18 – we conclude that for all $D \in \Lambda_s(G_i)$, $\{u, v\} \cap D \neq \{u, v\}$. Applying Lemma 20 directly we conclude that $\gamma_s(G_{i+1}) = \gamma_s(G_i)$.

Line 28 From the conditions given on lines 25 and 27 – and applying Lemma 17 – we conclude that there exists a $D_s \in \Lambda_s(G_i)$ where $u, v \in D_s$ and that either $\mathcal{U}(u) = 0$ or $\mathcal{U}(v) = 0$. Applying Lemma 21 directly we conclude that $\gamma_s(G_{i+1}) = \gamma_s(G_i) - 1$.

Line 30 From the conditions given on lines 25 and 29 – and applying Lemma 17 – we conclude that for all $D \in \Lambda_s(G_i)$ that $\mathcal{U}(u) > 0$ and $\mathcal{U}(v) > 0$. Applying Lemma 22 directly we conclude that $\gamma_s(G_{i+1}) = \gamma_s(G_i)$.

□

5.2 Running time of MDS algorithm

Theorem 24. *The running time of the MDS algorithm is $O(n)$.*

Proof. Recall that we have set t equal to the upper bound of the input graph's pathwidth.

A single left to right scan is made of the t-parse by the MDS algorithm to obtain the final solution. As a t-parse encodes the elements (that is both the vertices and edges) of a graph, there will be $O(n + m)$ operations. However as the input graph

has bounded pathwidth the number of edges within the graph are bounded by $m \leq n(t+1) - 1$; hence we can reduce this to $O(n)$.

For each partial solution the algorithm calculates a value for each tuple (s, q). Recall that $Q = \mathcal{P}(\{0, 1, ..., t\})$ and that $S_q = \mathcal{P}(\{0, 1, ..., t\} \setminus q)$ hence there will be $\sum_{i=1}^{t+1} \binom{t+1}{i}$ values of q and $2^{((t+1)-|q|}$ values of s for each value of q. Each value is updated in constant time, as detailed in the *Calculate State Entry* procedure above, therefore we have:

$$\begin{aligned} T(MDS) &= nc_1 \sum_{i=1}^{t+1} \binom{t+1}{i} 2^i + c_2 \\ &= nc_1(2+1)^{t+1} + c_2 && \text{by the binomial theorem} \\ &= nc_1 3^{t+1} + c_2 \\ &= O(n) \end{aligned}$$

□

Although the MDS algorithm is linear in terms of the input graph's size, it is exponential in terms of t with $O(3^{t+1})$ state entry values needing to be updated for each element of the input graph. However, the constants c_1 and c_2 will be very small in practice, consequently allowing the algorithm to be quite practical for small values of t.

6 Extending Algorithm for Graphs of Bounded Treewidth

We now present a concrete implementation of our dominating set algorithm. Actual working Python code is given in Figure 4. We also extend our pathwidth algorithm to handle graphs of bounded treewidth by including a method to update the state tables for the circle plus $\oplus$ operator (see [6]). Here we can glue together two t-parse's by identifying boundary vertices with the same label. Thus a treewidth t-parse is a tree of unary (vertex and edge) operators and binary circle plus operators. We summarize our MDS algorithm procedure below in terms of our dominating set state representation of a t-parse:

For $0 \leq x \leq t$ of the boundary. let $f : \{0, 1, \ldots, t\} \rightarrow \{0, 1, 2\}$ be an index denoting a dominating set cover D with $x \in D$ (if $f(x) = 0$), not in D but covered but by a t-parse vertex of D (if $f(x) = 1$), or not covered yet (assumed to be covered by an extension when $f(x) = 2$).

```
#!/usr/local/bin/python

import sys, re, array

class tOp:
  "A t-parse Operator"
  def __init__(self,tok): self.tok=int(tok)
  def __str__(self): return str(self.tok)
  def isEdgeOp(self): return (self.tok > 9)
  def isVertexOp(self): return (self.tok <= 9)
  def v1(self): return self.tok % 10 # vertex 1
  def v2(self): return self.tok / 10 # vertex 2 if EdgeOp

def rankDS(f):
  "produce dominating state from boundary states"
  " 0= 'in DS', 1= 'dominated but not in DS', 2= 'future dominated'"
  num=0
  for i in range(t+1): num=num*3+f[t-i]
  return num

def unrankDS(num):
  "extract boundary vertex dominating state from index"
  f=[]
  for i in range(t+1):
    f.append(num % 3); num /= 3;
  return f

def pwDS(pwTokens,state):
  "dynamic program for Pathwidth Dominating Set"
  for op in pwTokens:
    o=tOp(op); # print o
    stateOld=state[:]
    if o.isEdgeOp():
      v1=o.v1(); v2=o.v2()
      for i in xrange(3**(t+1)): # update all state boundary combinations
        f=unrankDS(i)
        if (f[v1]==0 and f[v2]==2) or (f[v1]==2 and f[v2]==0): state[i]=1<<30
        elif f[v1]==0 and f[v2]==1:
          fnew=f[:]; fnew[v2]=2; inew=rankDS(fnew)
          state[i]=min(stateOld[i],stateOld[inew])
        elif f[v1]==1 and f[v2]==0:
          fnew=f[:]; fnew[v1]=2; inew=rankDS(fnew)
          state[i]=min(stateOld[i],stateOld[inew])
    else: # isVertOp()
      v1=o.v1()
      for i in xrange(3**(t+1)):
        f=unrankDS(i)
        if f[v1]==0:
          fnew=f[:]; fnew[v1]=1; inew=rankDS(fnew)
          state[i]=min(stateOld[i],stateOld[inew])+1
        elif f[v1]==1: state[i]=1<<30
        elif f[v1]==2:
          fn1=f[:]; fn1[v1]=0; in1=rankDS(fn1)
          fn2=f[:]; fn2[v1]=1; in2=rankDS(fn2)
          state[i]=min(stateOld[in1],stateOld[in2])
  return state
```

Figure 4: Dominating set algorithm implemented using Python.

```
# Dominating Set (cont.)

def init_count(n):
  "how many in DS without internal DS vertices"
  cnt=0
  for i in unrankDS(n):
    if i==1: return 1 << 30 # no minimum since not dominated
    elif i==0: cnt+=1       # count this vertex
  return cnt

def twDS(G):
  "dynamic program for Treewidth Dominating Set"
  G=G.strip(); #print "G=",G
  state=array.array('i', map((lambda n:init_count(n)), range(3**(t+1))))
  if len(G)==0: return state # state of empty t-parse
  if G[0]!='(': return pwDS(re.findall('\d+',G),state)
  level=1
  for i in range(1,len(G)): # doing a circle plus operator
    if G[i]==')': level-=1
    elif G[i]=='(': level+=1
    if level==0:
      state1=twDS(G[1:i-1]) # strip a level of ()'s
      while 1:
        k=G[i+1:].find('('); # level will imeadiately be set to 1 below
        if k==-1: return pwDS(re.findall('\d+',G[i+1:]),state1)
        for j in range(i+1+k,len(G)): # get 2nd (next) argument to circle plus
          if G[j]==')': level-=1
          elif G[j]=='(': level+=1
          if level==0:
            state2=twDS(G[i+2+k:j-1]); #print "circle plus\n"
            state=array.array('i', map((lambda n:1<<30), range(3**(t+1))))
            for x in xrange(3**(t+1)): # now update state for circle plus
              for y in xrange(3**(t+1)):
                f1=unrankDS(x); f2=unrankDS(y);
                f=[]; common=0;
                for z in range(t+1):  # compute new boundary f() conditions
                   if f1[z]==0 and f2[z]==0: common+=1
                   if f1[z]==0 or f2[z]==0: f.append(0)
                   elif f1[z]==2 and f2[z]==2: f.append(2)
                   else: f.append(1)
                r=rankDS(f);
                s1=state1[rankDS(f1)]; s2=state2[rankDS(f2)]
                if s1 < 1<<30 and s2 < 1<<30: state[r]=min(state[r],s1+s2-common)
            if j+1 < len(G):
              state1=state; i=j+1
              break  # to next while iteration to look for another circle plus
            else: return state

# main program
#
s=sys.stdin.read().strip()              # read input graph
i=s.find('('); t=int(s[0:i-1]); s=s[i:] # set global treewidth t
best=1<<30; state=twDS(s)
for i in xrange(3**(t+1)):
  flag=1
  for f in unrankDS(i):
    if f==2: flag=0; break;
  if flag: best = min(best,state[i])
print best
```

Figure 4: (cont.) Dominating set algorithm implemented using Python.

For the 3^{t+1} possible functions f, we set $T[f]$ to be the minimum dominating set of the current t-parse (or undefined if one isn't possible). That is, we don't count vertices that will occur in an exentsion.

$\diamond$ For initial empty t-parse:

$$T[f] = \left\{ \begin{array}{l} \text{undef if } f(x) = 1 \text{ for any } x \\ \text{the number of } x \text{ such that } f(x) = 0, \text{ otherwise} \end{array} \right\}$$

$\diamond$ Vertex operator $\textcircled{x}$ update rule:

case 1: $f(x) = 0$, $T'[f] = \min(T[f], T[f \text{with} f(x) = 1]) + 1$

case 2: $f(x) = 1$, $T'[f] = \text{undef}$

case 3: $f(x) = 2$, $T'[f] = \min(T[f \text{ with } f(x) = 0], T[f \text{ with } f(x) = 1])$

$\diamond$ Edge operator $\boxed{u\ v}$ update rule:

case 0: $f(u) = 2$ and $f(v) = 0$, $T'[f] = \text{undef}$
$f(u) = 0$ and $f(v) = 2$, $T'[f] = \text{undef}$

case 1: $f(u) = f(v)$, $T'[f] = T[f]$

case 2: $f(u) = 0$ and $f(v) = 1$, $T'[f] = \min(T[f], T[f \text{ with } f(v) = 2])$
$f(u) = 1$ and $f(v) = 0$, $T'[f] = \min(T[f], T[f \text{ with } f(u) = 2])$

case 3: $f(u) = 1$ and $f(v) = 2$, $T'[f] = T[f]$
$f(u) = 2$ and $f(v) = 1$, $T'[f] = T[f]$

$\diamond$ Circle plus operator $G_1 \oplus G_2$ update rule:

We need to combine various substate cases to get the state for each index $f()$. Consider all $f_1()$ and $f_2()$ for t-parses G_1 and G_2. Let common(f) be the number times $f_1(x) = f_2(x) = 0$. We can combine a $f_1(x) = 1$ and $f_2(x) = 2$ (or vise versa) to get $f(x) = 1$.

If one of $f_1(x) = 0$ or $f_2(x) = 0$ then $f(x) = 0$. If both $f_1(x) = 2$ and $f_2(x) = 2$ then $f(x) = 2$. Otherwise $f(x) = 1$.

$T'[f] = \min(T1[f_1] + T2[f_2] - \text{common}(f))$.

6.1 Input format

Our Python program reads input as a sequence of tokens (printable ASCII characters separated by white space) from the *keyboard/stdin/console*. The first token will be a positive integer $t \leq 9$ denoting the pathwidth/treewidth of the graph (i.e., it indicates a t-parse will follow). The interpretation for the remaining tokens are given in the next table.

token	**semantic meaning**
i	a vertex operator $\textcircled{i}$, represented as an integer, where $0 \leq i \leq t \leq 9$
ij	an edge operator $\boxed{i\ j}$ where $t \geq i > j \geq 0$ (note: no space between i and j)
(	a begin marker for a pathwidth t-parse (can be nested for tree branches)
)	an end marker for a pathwidth/treewidth t-parse

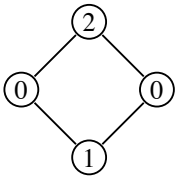
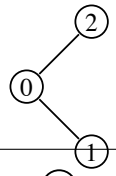
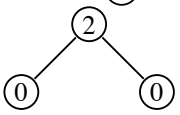
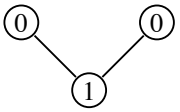
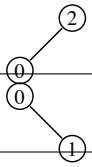
Thus, we read/encounter the two tokens ')' and '(' in sequence then this denotes a circle plus $\oplus$ operator. Note that it is guaranteed that each right token '(' will have a matched left token ')'. We assume boundary vertices $0, 1, \ldots, t$ precede the first token of a pathwidth t-parse (and are not given in the input format). Assume *left-associativity* for all operators and at least one space between each of them, including the parentheses.

6.2 Examples

Some concrete examples of the MDS algorithm, using the Python input format, are given in the next table.

input	**output**
2 (10 21 1 10 21 0 10 20 2 20 21)	2
2 ((10 1 10 21 1 21 10) (21 2))	2
2 ((10 1 10 21 1 21 10 0) ((10 1 10 21 1 21 10 0) (10 1 10 21 1 21 10 0)))	5
3 (32 31 30 20 21 10 0 10 20 21 1 10 21 1 10 21 1 10 0 10 30 20)	2
2 (21 2 21 2 21 20 1 10 21 1 10 21 1 10 21 1 10 21 1 10 21 2 21 1 10 20)	2
3 ((30 32 21 10 1 10 2 32 1 21 10 31 1 21 10 31) (2 20 21 32 2 20 21 32) 0 1 10 1 10 0 10 1 10)	5
4 (10 20 30 40 32 2 20 21 42 0 10 2 21 3 30 31 4 41)	2

The following example shows the state information for each stage of the MDS algorithm for input $T = [\,(0), (1), (2), \boxed{0\ 1}, \boxed{0\ 2}, (0), \boxed{0\ 1}, \boxed{0\ 2}\,]$

(0), (1), (2)			$\boxed{0\ 1}$	$\boxed{0\ 2}$	(0)	$\boxed{0\ 1}$	$\boxed{0\ 2}$	
Q	S	$\gamma(G_i - q)_s$						
∅	∅	3	2	1	2	2	2	
	{0}	3	2	1	2	2	2	
	{1}	3	2	2	3	2	2	
	{2}	3	2	2	3	2	2	
	{0,1}	3	3	2	3	3	2	
	{0,2}	3	2	2	3	2	2	
	{1,2}	3	2	2	3	2	2	
	{0,1,2}	3	3	3	3	3	3	
{0}	∅	2	2	2	1	1	1	
	{1}	2	2	2	2	2	2	
	{2}	2	2	2	2	2	2	
	{1,2}	2	2	2	2	2	2	
{1}	∅	2	2	1	2	2	1	
	{0}	2	2	1	2	2	2	
	{2}	2	2	1	2	2	1	
	{0,2}	2	2	2	2	2	2	
{2}	∅	2	1	1	2	1	1	
	{0}	2	1	1	2	2	2	
	{1}	2	1	1	2	1	1	
	{0,1}	2	2	2	2	2	2	
{0,1}	∅	1	1	1	1	1	1	
	{2}	1	1	1	1	1	1	
{0,2}	∅	1	1	1	1	1	1	
	{1}	1	1	1	1	1	1	
{1,2}	∅	1	1	1	2	2	2	
	{0}	1	1	1	2	2	2	
{0,1,2}	∅	0	0	0	1	1	1	

7 Conclusion

In this paper we have presented an algorithm for determining the minimum dominating number for graphs of bounded pathwidth. It has running time $O(n3^{t+1})$ for graph's of order n and pathwidth upper bound t.

The algorithm has several advantages over previous bounded pathwidth algorithms for obtaining minimum dominating numbers. These include: the algorithm is simple and will have a straight forward implementation; it has fast running time

for small values of t; and it can be extended in constant time, allowing graphs to be interactively grown. Additionally the algorithm presented here is fully specified.

References

[1] S. Arnborg and A. Proskurowski. Linear-Time algorithms for NP-hard problems on graphs embedded in k-trees. Discrete Applied Mathematics, 23:11–24, 1989.

[2] Neil Robertson and Paul D. Seymour. Graph Minors - A survey. In Surveys in Combinatorics, volume 103, pages 153171. Cambridge University Press, 1985.

[3] Neil Robertson and Paul D. Seymour. Graph Minors I. Excluding a Forest. Journal of Combinatorial Theory, Series B, 35(1):39–61, 1983.

[4] Neil Robertson and Paul D. Seymour. Graph Minors III. Planar treewidth. Journal of Combinatorial Theory, Series B, 36:49–64, 1984.

[5] Reinhard Diestel, Graph Theory, Electronic Edition. Springer Verlag, New York, 1997.

[6] Michael J Dinneen, Practical Enumeration Methods for Graphs of Bounded Pathwidth and Treewidth. CDMTCS Research Report Series. http://www.cs.auckland.ac.nz/CDMTCS//researchreports/055mjd.pdf

[7] Kevin Cattell and Michael J. Dinneen. A characterization of graphs with vertex cover up to five. In Vincent Bouchitte and Michel Morvan, editors, Orders, Algorithms and Applications, ORDAL'94, volume 831 of Lecture Notes on Computer Science, pages 86–99. Springer Verlag, July 1994.

[8] Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP–completeness. W. H. Freeman and Company, 1979.

[9] C.F. de Jaenisch, Applications de l'Analyse Mathematique au Jeu des Echecs, Appendix, pages 244 (Petrograd, 1862).