A Simple Linear-Time Algorithm for Finding Path-Decompostions of Small Width

Kevin Cattell   Michael J. Dinneen   Michael R. Fellows

Department of Computer Science
University of Victoria
Victoria, B.C. Canada V8W 3P6

Information Processing Letters 57 (1996) 197–203

Outline

1. Introduction
   - Motivation
   - History
2. Preliminary Definitions
   - Boundaried graphs
   - Path-decompositions
   - Topological tree obstructions
3. Pathwidth Algorithm
   - Main result
   - Linear-time algorithm
   - Proof of correctness
   - Other results

Motivation

- Pathwidth is related to several VLSI layout problems:
  - vertex separation link
  - gate matrix layout
  - edge search number
  - . . .
- Usefullness of bounded treewidth in:
  - study of graph minors (Robertson and Seymour)
  - input restrictions for many NP-complete problems
  - (fixed-parameter complexity)

History

- General problem(s) is NP-complete
  Input: Graph G, integer t
  Question: Is tree/path-width(G) $\leq t$?
- Algorithmic development (fixed t):
  - $O(n^2)$ nonconstructive treewidth algorithm by Robertson and Seymour (1986)
  - $O(n^{t+2})$ treewidth algorithm due to Arnberg, Corneil and Proskurowski (1987)
  - $O(n \log n)$ treewidth algorithm due to Reed (1992)
  - $O(2^{t^2} n)$ treewidth algorithm due to Bodlaender (1993)
  - $O(n \log^2 n)$ pathwidth algorithm due to Ellis, Sudborough and Turner (1994)

History

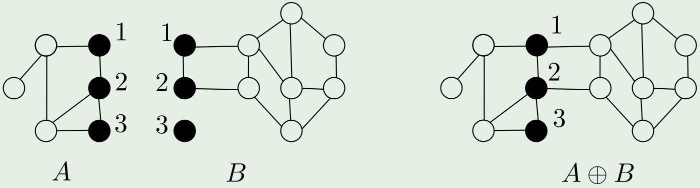
- General problem(s) is NP-complete
  Input: Graph G, integer t
  Question: Is tree/path-width$(G) \leq t$?
- Algorithmic development (fixed t):
  - $O(n^2)$ nonconstructive treewidth algorithm by Robertson and Seymour (1986)
  - $O(n^{t+2})$ treewidth algorithm due to Arnberg, Corneil and Proskurowski (1987)
  - $O(n \log n)$ treewidth algorithm due to Reed (1992)
  - $O(2^{t^2} n)$ treewidth algorithm due to Bodlaender (1993)
  - $O(n \log^2 n)$ pathwidth algorithm due to Ellis, Sudborough and Turner (1994)

Boundaried graphs

- A distinguished set of vertices labeled $1, 2, \ldots, k$, is called the boundary of a (finite simple) graph.
- A boundary size k *factorization* of a graph G is two k-boundaried graphs A and B such that $G = A \oplus B$.

Example

The $\oplus$ operator on two 3-boundaried graphs A and B is illustrated below.



Path-decompositions

Definition

A *path-decomposition* of a graph $G = (V, E)$ is a sequence $X_1, X_2, \ldots, X_r$ of subsets of V that satisfy the following:

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

Definition

The *pathwidth of a path-decomposition* $X_1, X_2, \ldots, X_r$ is $\max_{1 \le i \le r} |X_i| - 1$. The *pathwidth of a graph* G is the minimum pathwidth over all path-decompositions of G.

Path-decompositions

Definition

A *path-decomposition* of a graph $G = (V, E)$ is a sequence $X_1, X_2, \ldots, X_r$ of subsets of V that satisfy the following:

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

Definition

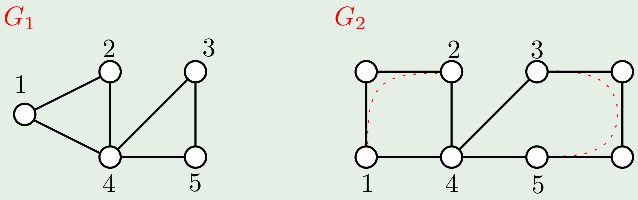
The *pathwidth of a path-decomposition* $X_1, X_2, \ldots, X_r$ is $\max_{1 \le i \le r} |X_i| - 1$. The *pathwidth of a graph G* is the minimum pathwidth over all path-decompositions of G.

Graph embeddings

Definition

An *(homeomorphic) embedding* of a graph $G_1 = (V_1, E_1)$ in a graph $G_2 = (V_2, E_2)$ is an injection from vertices V_1 to V_2 such that the edges E_1 are mapped to disjoint paths of G_2.

Example



Topological order

Definition

The set of homeomorphic embeddings between graphs gives a partial order, called the *topological order*.

Definition

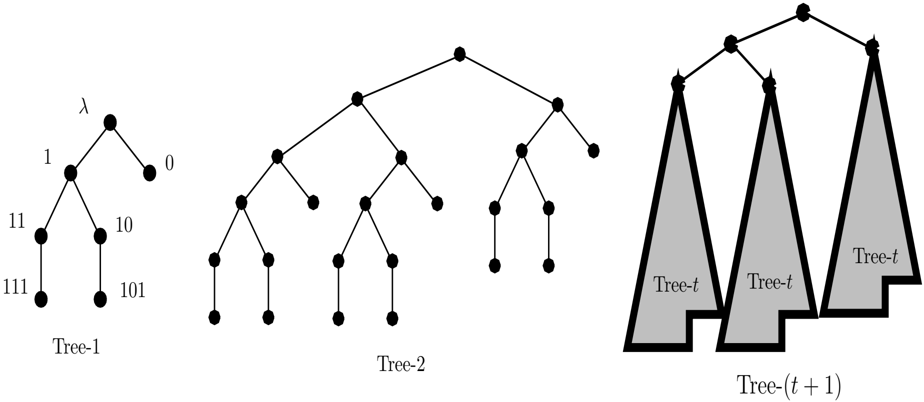
A *lower ideal* $\mathcal{J}$ in a partial order $(\mathcal{U}, \geq)$ is a subset of $\mathcal{U}$ such that if $X \in \mathcal{J}$ and $X \geq Y$ then $Y \in \mathcal{J}$. The *obstruction set* for $\mathcal{J}$ is the set of minimal elements of $\mathcal{U} - \mathcal{J}$.

Recursively generated tree obstructions

Some recursive rules for generating all topological tree obstructions of pathwidth t:

1. The tree K_2 is the only obstruction of pathwidth 0.
2. If T_1, T_2 and T_3 are any 3 tree obstructions for pathwidth t then the tree T consisting of a new degree 3 vertex attached to any vertex of T_1, T_2 and T_3 is a tree obstruction for pathwidth $t + 1$.

Embedding tree obstructions in binary trees.



This shows that the complete binary tree of height $h(t) = 2t + 1$ and order $f(t) = 2^{2t+1} - 1$ has pathwidth greater than t.

Main result

Theorem

Let H be an arbitrary undirected graph, and let t be a positive integer. One of the following two statements must hold:

1. *The pathwidth of H is at most $f(t) - 1$.*
2. *H can be factored: $H = A \oplus B$, where A and B are boundaried graphs with boundary size $f(t)$, the pathwidth of A is greater than t and less than $f(t)$.*

Proof idea of main result

- Assume we can embed a *guest tree* $B_{h(t)}$ in the *host graph* H then we know that the pathwidth of H is greater than t. (e.g. height $h(t) \geq \lg f(t)$)
- Refer to the vertices of $B_{h(t)}$ as tokens, and call tokens *placed* (or *unplaced*) if they are (not) mapped to vertices of H in the current partial embedding.
  A vertex v of H is *tokened* if a token maps to v.
- Let $P[i]$ denote the set of vertices of H that are tokened at time step i.
  The sequence $P[0], P[1], \ldots, P[s]$ will describe either a path-decomposition of H or of a factor A.

Proof idea of main result

- Assume we can embed a *guest tree* $B_{h(t)}$ in the *host graph* H then we know that the pathwidth of H is greater than t. (e.g. height $h(t) \geq \lg f(t)$)
- Refer to the vertices of $B_{h(t)}$ as tokens, and call tokens *placed* (or *unplaced*) if they are (not) mapped to vertices of H in the current partial embedding.
  A vertex v of H is *tokened* if a token maps to v.
- Let $P[i]$ denote the set of vertices of H that are tokened at time step i.
  The sequence $P[0], P[1], \ldots, P[s]$ will describe either a path-decomposition of H or of a factor A.

Proof idea of main result

- Assume we can embed a *guest tree* $B_{h(t)}$ in the *host graph* H then we know that the pathwidth of H is greater than t. (e.g. height $h(t) \geq \lg f(t)$)
- Refer to the vertices of $B_{h(t)}$ as tokens, and call tokens *placed* (or *unplaced*) if they are (not) mapped to vertices of H in the current partial embedding.
  A vertex v of H is *tokened* if a token maps to v.
- Let $P[i]$ denote the set of vertices of H that are tokened at time step i.
  The sequence $P[0], P[1], \ldots, P[s]$ will describe either a path-decomposition of H or of a factor A.

Identifying tokens and tokened vertices

We recursively label the tokens of the guest binary tree by the following standard rules:

1. The root token of $B_{h(t)}$ is labeled by the empty string λ.
2. The left child token and right child token of a height h parent token $P = b_1 b_2 \cdots b_h$ are labeled $P \cdot 1$ and $P \cdot 0$, respectively.

The token placement algorithm is described as follows.

1. Initially consider that every vertex of H is colored blue.
2. A vertex of H has its color changed to red when a token is placed on it, and stays red if the token is removed.
3. Only blue vertices can be tokened, and so a vertex can only receive a token once.

Identifying tokens and tokened vertices

We recursively label the tokens of the guest binary tree by the following standard rules:

1. The root token of $B_{h(t)}$ is labeled by the empty string λ.
2. The left child token and right child token of a height h parent token $P = b_1 b_2 \cdots b_h$ are labeled $P \cdot 1$ and $P \cdot 0$, respectively.

The token placement algorithm is described as follows.

1. Initially consider that every vertex of H is colored blue.
2. A vertex of H has its color changed to red when a token is placed on it, and stays red if the token is removed.
3. Only blue vertices can be tokened, and so a vertex can only receive a token once.

Linear-time algorithm (grow part)

function GrowTokenTree

1 **if** root token λ is not placed on H **then**

arbitrarily place λ on a blue vertex of H

endif

2 **while** there is a vertex $u \in H$ with token T and blue neighbor v, and token T has an unplaced child $T \cdot b$ **do**

2.1 place token $T \cdot b$ on v

endwhile

3 **return** {tokened vertices of H}

Linear-time algorithm (main program)

program PathDecompositionOrSmallFatFactor

1 $i \leftarrow 0$

2 $P[i] \leftarrow$ **call** GrowTokenTree

3 **until** $|P[i]| = f(t)$ **or** H has no blue vertices **repeat**

  3.1 pick a token T with an unplaced child token

  3.2 remove T from H

  3.3 **if** T had one tokened child **then**

    replace all tokens $T \cdot b \cdot S$ with $T \cdot S$

  endif

  3.4 $i \leftarrow i + 1$

  3.5 $P[i] \leftarrow$ **call** GrowTokenTree

enduntil

done

Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

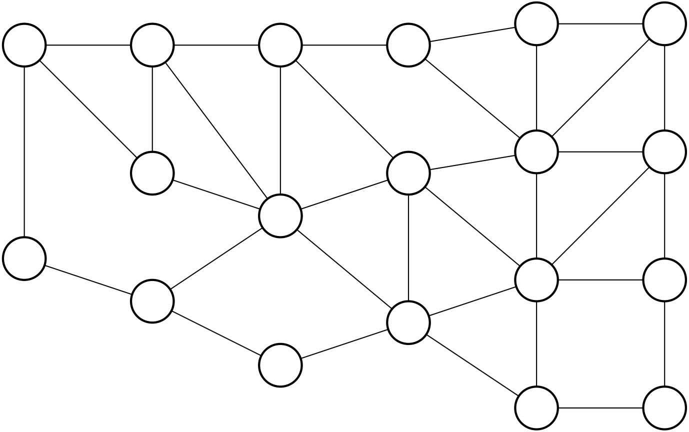


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

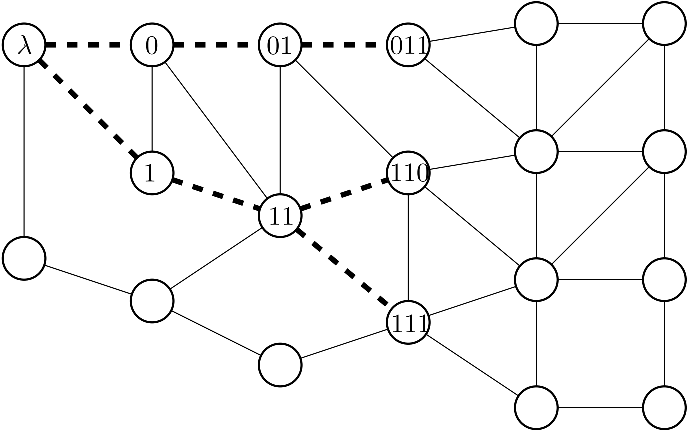


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

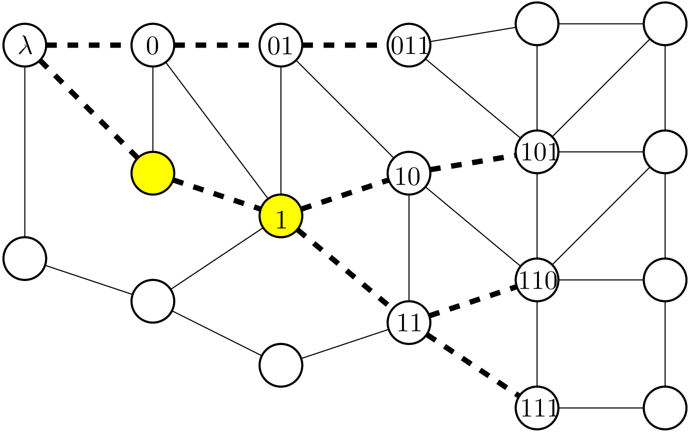


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

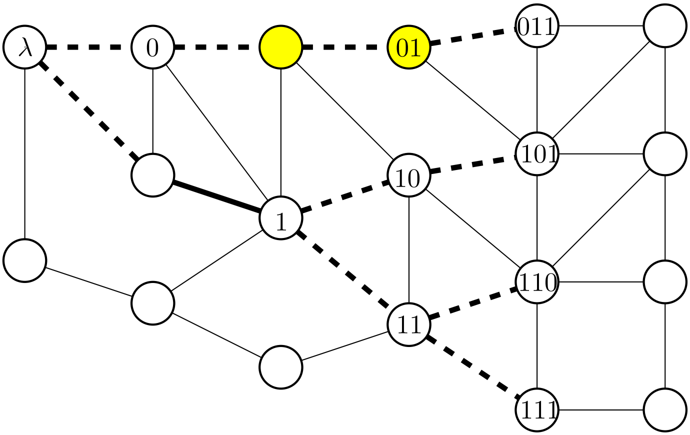


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

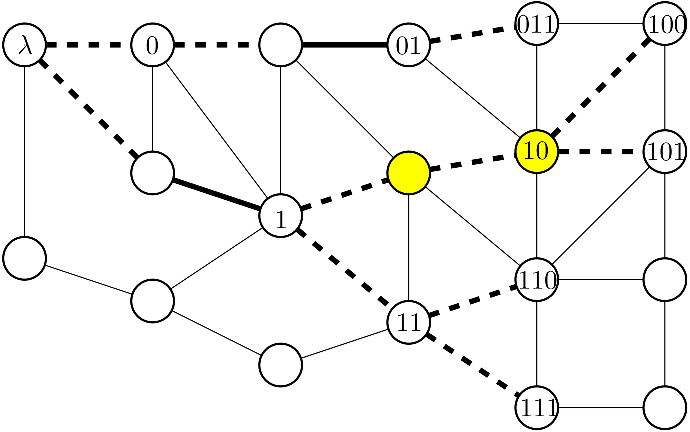


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

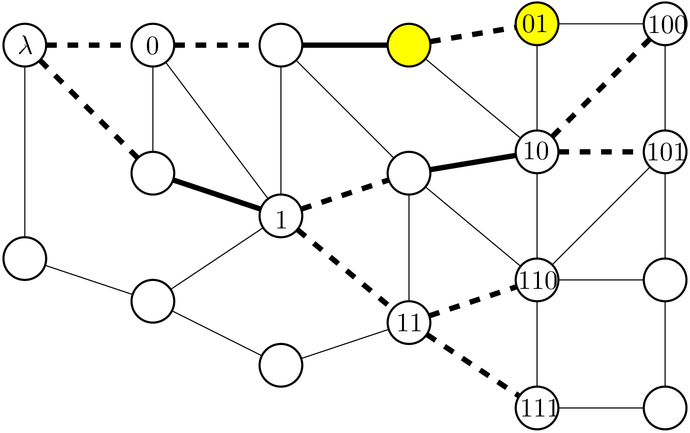


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

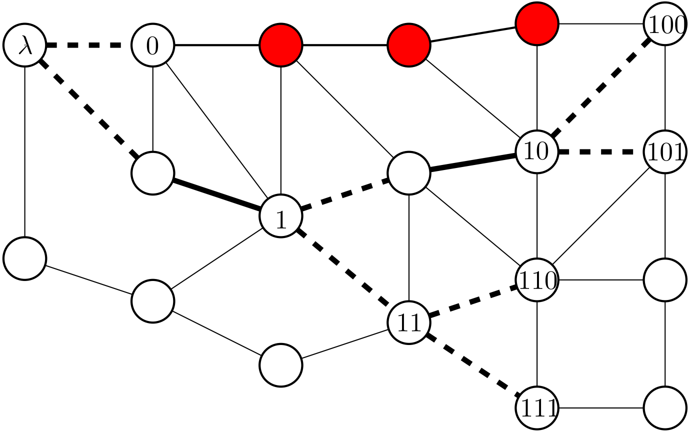


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

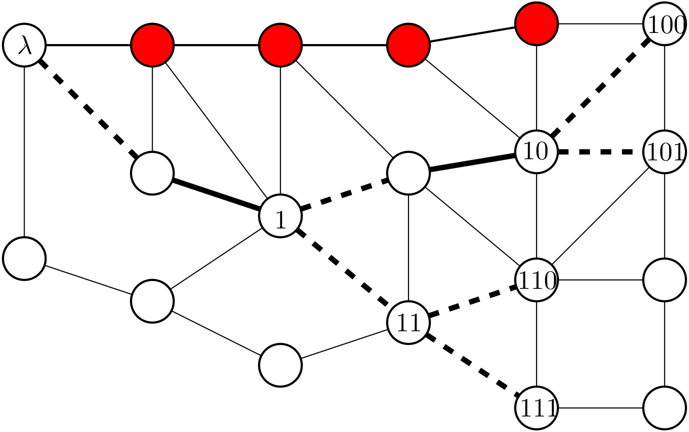


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

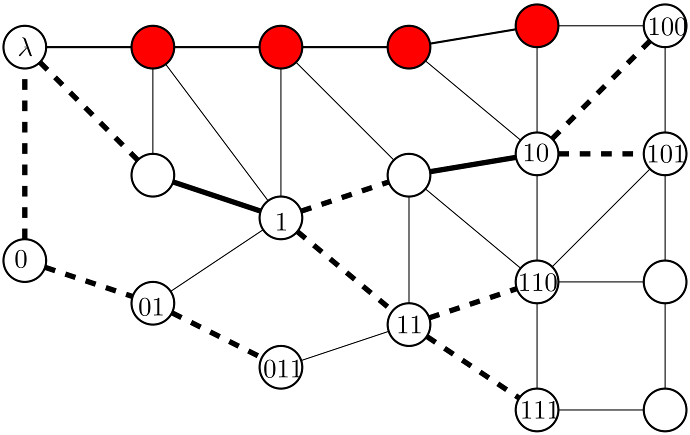


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

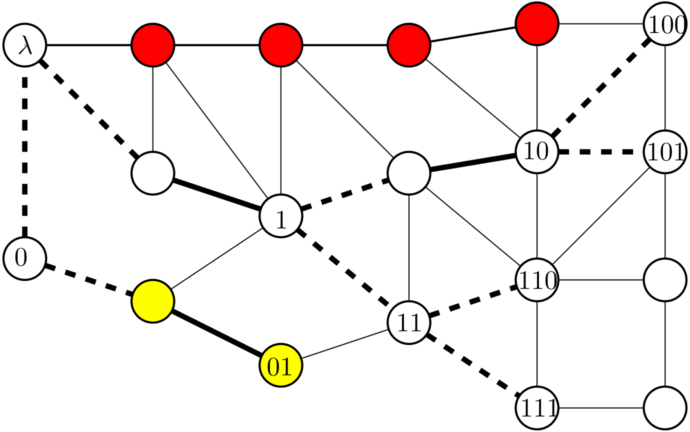


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

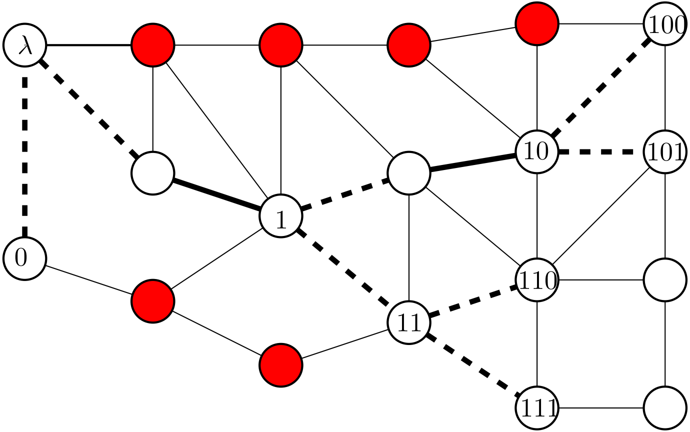


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

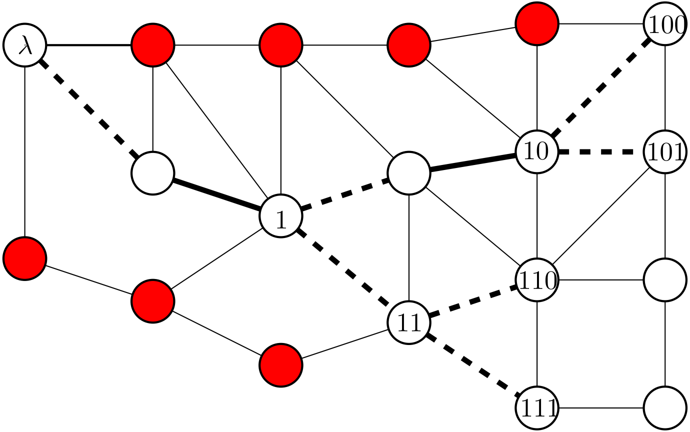


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.

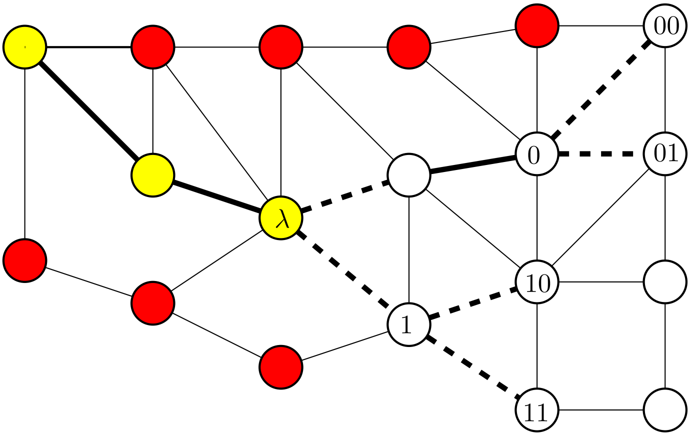
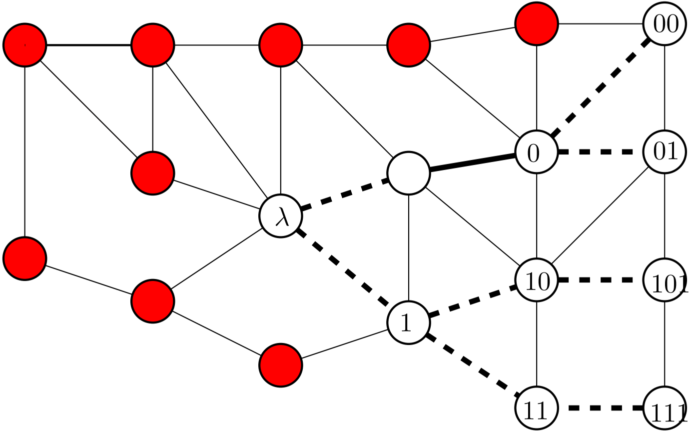


Illustration of algorithm execution

Trying to embed a complete binary tree of height 3.



Why the algorithm is correct (1 of 2)

Some properties of the algorithm:

- The root token λ of $B_{h(t)}$ is placed at most once for each component of H. But can move in steps 3.2-3.3.
- GrowTokenTree only returns when either $B_{h(t)}$ is completely embedded or there are no blue neighbors for the unplaced tokens.
- The algorithm terminates since each iteration of step 3.2 a tokened red vertex becomes untokened. (This can happen at most n times.)

Why the algorithm is correct (1 of 2)

Some properties of the algorithm:

- The root token λ of $B_{h(t)}$ is placed at most once for each component of H. But can move in steps 3.2-3.3.
- GrowTokenTree only returns when either $B_{h(t)}$ is completely embedded or there are no blue neighbors for the unplaced tokens.
- The algorithm terminates since each iteration of step 3.2 a tokened red vertex becomes untokened. (This can happen at most n times.)

Why the algorithm is correct (1 of 2)

Some properties of the algorithm:

- The root token λ of $B_{h(t)}$ is placed at most once for each component of H. But can move in steps 3.2-3.3.
- GrowTokenTree only returns when either $B_{h(t)}$ is completely embedded or there are no blue neighbors for the unplaced tokens.
- The algorithm terminates since each iteration of step 3.2 a tokened red vertex becomes untokened. (This can happen at most n times.)

Why the algorithm is correct (2 of 2)

Why $P[0], \ldots, P[s]$ is a path-decomposition of H or A?

- Since each vertex u is tokened at most once, the interpolation property holds.
- Let (u, v) be an edge and assume vertex u is tokened first. We only untoken a vertex when there is an unplaced child token step 3.2.
  Thus, vertex v will be tokened as a child token of u.
  Therefore, there is some $P[i]$ containing both u and v.

If all tokens of $B_{h(t)}$ are embedded into a subgraph of H we claim that A contains a subdivision of $B_{h(t)}$.
Since GrowTokenTree only attaches pendant tokens to parent tokens we need only observe that the operation in step 3.3 subdivides the edge between T and its parent.

Why the algorithm is correct (2 of 2)

Why $P[0], \ldots, P[s]$ is a path-decomposition of H or A?

- Since each vertex u is tokened at most once, the interpolation property holds.
- Let (u, v) be an edge and assume vertex u is tokened first. We only untoken a vertex when there is an unplaced child token step 3.2.
  Thus, vertex v will be tokened as a child token of u.
  Therefore, there is some $P[i]$ containing both u and v.

If all tokens of $B_{h(t)}$ are embedded into a subgraph of H we claim that A contains a subdivision of $B_{h(t)}$.
Since GrowTokenTree only attaches pendant tokens to parent tokens we need only observe that the operation in step 3.3 subdivides the edge between T and its parent. □

Why the algoritm runs in linear time

- If graph H has more than $t \cdot n$ edges then the pathwidth is greater than t (i.e. input has $O(n)$ edges).
- All operations on $B_{h(t)}$ are constant time.
- In GrowTokenTree step 2 if we find an edge (u, v) where v is a red vertex, we can delete it.
  Also it is safe to remove (u, v) after step 2.1.
  Therefore, across all calls, each edge of H needs to be considered at most once.
- The number of iterations in PathDecompositionOrSmallFatFactor is at most n.

Why the algoritm runs in linear time

- If graph H has more than $t \cdot n$ edges then the pathwidth is greater than t (i.e. input has $O(n)$ edges).
- All operations on $B_{h(t)}$ are constant time.
- In GrowTokenTree step 2 if we find an edge (u, v) where v is a red vertex, we can delete it.
  Also it is safe to remove (u, v) after step 2.1.
  Therefore, across all calls, each edge of H needs to be considered at most once.
- The number of iterations in PathDecompositionOrSmallFatFactor is at most n.

Why the algoritm runs in linear time

- If graph H has more than $t \cdot n$ edges then the pathwidth is greater than t (i.e. input has $O(n)$ edges).
- All operations on $B_{h(t)}$ are constant time.
- In GrowTokenTree step 2 if we find an edge (u, v) where v is a red vertex, we can delete it.
  Also it is safe to remove (u, v) after step 2.1.
  Therefore, across all calls, each edge of H needs to be considered at most once.
- The number of iterations in PathDecompositionOrSmallFatFactor is at most n.

Why the algoritm runs in linear time

- If graph H has more than $t \cdot n$ edges then the pathwidth is greater than t (i.e. input has $O(n)$ edges).
- All operations on $B_{h(t)}$ are constant time.
- In GrowTokenTree step 2 if we find an edge (u, v) where v is a red vertex, we can delete it.
  Also it is safe to remove (u, v) after step 2.1.
  Therefore, across all calls, each edge of H needs to be considered at most once.
- The number of iterations in PathDecompositionOrSmallFatFactor is at most n.

Other results

Corollary

Any subtree of the binary tree $B_{h(t)}$ that has pathwidth greater than t may be used in the pathwidth algorithm.

Corollary

Every graph with no minor isomorphic to forest F, where F is a minor of a complete binary tree B, has pathwidth at most $c = |B| - 2$.

Other results

Corollary

Any subtree of the binary tree $B_{h(t)}$ that has pathwidth greater than t may be used in the pathwidth algorithm.

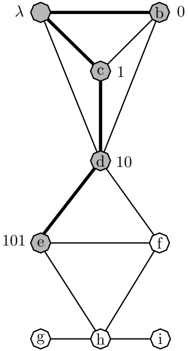
Corollary

Every graph with no minor isomorphic to forest F, where F is a minor of a complete binary tree B, has pathwidth at most $c = |B| - 2$.

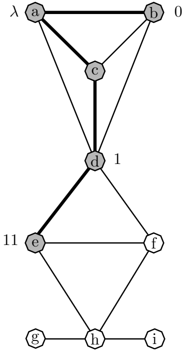
This is basically the main result of Bienstock, Robertson, Seymour and Thomas (1991) that for any forest F there is a constant c, such that any graph not containing F as a minor has pathwidth at most c.

Illustration of algorithm (revised)

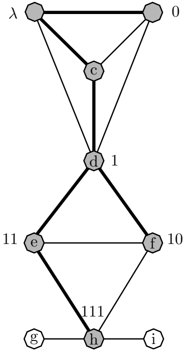
Trying to directly embed the Tree-1 obstruction.



After step 2,
$P[0] = \{a, b, c, d, e\}$



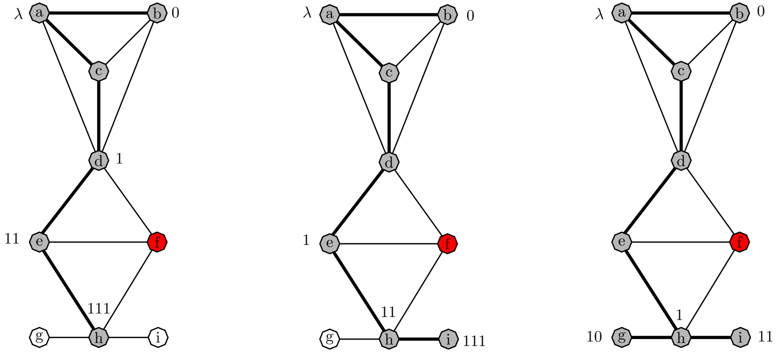
After 3.3, $T = 1$



After 3.5
$P[1] = \{a, b, d, e, f, h\}$

Illustration of algorithm (revised)

Trying to directly embed the Tree-1 obstruction.



$T = 10$, after 3.5

$P[2] = \{a, b, d, e, h\}$

$T = 1$, after 3.5

$P[3] = \{a, b, e, h, i\}$

$T = 1$, after 3.5

$P[4] = \{a, b, g, h, i\}$

Summary

We have presented a simple linear-time algorithm (for each fixed constant t) that either establishes that the pathwidth of a graph is greater than t, or finds a path-decomposition of width at most $O(2^t)$.

- The width is equal to the number of tokens used. In practice this may be smaller than the complete binary tree.
- Can the width of the path-decomposition be bounded to the number of vertices in tree obstructions?
- There may be placement heuristics that can improve our performance on "typical" instances.

Summary

We have presented a simple linear-time algorithm (for each fixed constant t) that either establishes that the pathwidth of a graph is greater than t, or finds a path-decomposition of width at most $O(2^t)$.

- The width is equal to the number of tokens used. In practice this may be smaller than the complete binary tree.
- Can the width of the path-decomposition be bounded to the number of vertices in tree obstructions?
- There may be placement heuristics that can improve our performance on "typical" instances.

Summary

We have presented a simple linear-time algorithm (for each fixed constant t) that either establishes that the pathwidth of a graph is greater than t, or finds a path-decomposition of width at most $O(2^t)$.

- The width is equal to the number of tokens used. In practice this may be smaller than the complete binary tree.
- Can the width of the path-decomposition be bounded to the number of vertices in tree obstructions?
- There may be placement heuristics that can improve our performance on “typical” instances.

Summary

We have presented a simple linear-time algorithm (for each fixed constant t) that either establishes that the pathwidth of a graph is greater than t, or finds a path-decomposition of width at most $O(2^t)$.

- The width is equal to the number of tokens used. In practice this may be smaller than the complete binary tree.
- Can the width of the path-decomposition be bounded to the number of vertices in tree obstructions?
- There may be placement heuristics that can improve our performance on "typical" instances.

Thank you!

Vertex separation

Definition

A layout L of a graph $G = (V, E)$ is a one to one mapping $L : V \rightarrow \{1, 2, \ldots, |V|\}$.

For a graph $G = (V, E)$ we conveniently write a layout L as a permutation of the vertices $(v_1, v_2, \ldots, v_n)$.
For any layout $L = (v_1, v_2, \ldots, v_n)$ of G let

$$V_i = \{v_j \mid j \leq i \text{ and } (v_j, v_k) \in E \text{ for some } k > i\}$$

for each $1 \leq i \leq n$.

Definition

The vertex separation of a graph G with respect to a layout L is $vs(L, G) = \max_{1 \leq i \leq |G|}\{|V_i|\}$.
The vertex separation of a graph G, denoted by $vs(G)$, is the minimum $vs(L, G)$ over all layouts L of G.