CDMTCS
Research
Report
Series

Branchwidth, Branch Decompositions and *b*-parses

Andrew Probert
Michael J. Dinneen

Department of Computer Science,
University of Auckland,
Auckland, New Zealand

CDMTCS-437
April 2013

Centre for Discrete Mathematics and
Theoretical Computer Science

Branchwidth, Branch Decompositions and b-parses

Andrew Probert and Michael J. Dinneen

Department of Computer Science, University of Auckland, Auckland, New Zealand

apro002@aucklanduni.ac.nz mjd@cs.auckland.ac.nz

Abstract

In this paper we present an easy-to-use data structure for representing graphs of bounded branchwidth, called b-parses. Many hard problems that can be represented as graph problems can be more easily solved when a decomposition of the graph is taken into account. This is particularly true where the input graph can be seen to be treelike in some form. An example of such a treelike structure is branch decomposition, were the edges of a graph are arranged as leaves around a tree and the internal nodes of the tree represent connectivity between subsets of the edges of the original graph. This is similar in concept to the idea of tree decomposition which views the input graph vertices as forming a treelike structure of bounded-sized vertex separators. However branch decompositions may be simpler to work with than tree decompositions for appropriate problems because of the structure (and possibly smaller width) of the tree that is formed. In this paper an algebraic representation of branch decompositions (b-parse) is proposed as an alternative to the t-parse representation for tree decompositions. An application of this data structure to the known hard problems Minimum Vertex Cover and 3-Coloring is examined. Finally, possible benefits of using b-parses from the parallelism perspective is given.

1 Introduction

Problems that can be expressed in terms of graphs are often more easily solved if the structure of the graph is taken into account. This is particularly so where the graph can be viewed as a tree or treelike. The branch decomposition and its associated parameter or metric branchwidth is a way of viewing a graph as treelike in terms of its edges, where we want to minimize the shared incident vertices with respect to small local subtrees of the branch decomposition. This technique has certain advantages over other techniques such as tree decomposition. These include the following:

- For planar graphs it has been proven that one can show that a graph does or does not have a branch decomposition of less than or equal to a given branchwidth in true polynomial time (no such algorithm is known for treewidth).

- Branch decompositions can be represented as rooted trees in which case they are always binary. This gives a much simpler structure to work with than the average tree decomposition (though one can modify tree decomposition trees to make them nice and better balanced see [DT06]).
- Many graph problems are inherently concerned with the edges of the graph (minimum vertex cover for example asks about edge coverage).
- The simpler binary structure of the rooted branch decomposition tree may imply a higher degree of potential parallelism.

In the early 1980's Robertson and Seymour [RS83, RS84, RS91] began a series of papers in which they popularized the concept of k-trees developing the notion of path/tree decompositions with bounded width. In essence this was a generalization of trees. Informally put, graphs of width at most k are subgraphs of tree-like or path-like graphs where the vertices form cliques of size at most $k+1$ (e.g. see [Din97]). Alongside this they developed the concept of branch decomposition which describes the way in which the edges of a graph interconnect with the focus being on the minimum level of connectivity between subsets of edges (or branchwidth) (see [RS91]).

It has been known for some time that problems that are otherwise $\mathcal{NP}$-complete or $\mathcal{NP}$-hard when used with general graphs as input can be easily (in polynomial time) solved when computed when restricted to trees. For example, the minimum vertex cover problem whereby we seek the minimum number of vertices such that every edge in a graph has at least one vertex in this set is known to be $\mathcal{NP}$-complete. However, if the input is a tree the solution is simpler: we choose a leaf edge $\{u, v\}$ where if v is the leaf we add u to the vertex cover then remove u and repeat recursively on the remaining forest until we reach single node trees. This runs in linear time in the number of edges. A similar benefit can be obtained when tree-like input structures are used. This paper will focus mostly upon branch decompositions of bounded width.

In order to use branch-decompositions of bounded width to solve $\mathcal{NP}$-hard problems we are faced with the problem of finding or constructing the decomposition itself. As with the problem of finding tree decompositions this has been a very busy area of research since the pioneering work of Robertson and Seymour. The problem of finding branch-decompositions of small width is in general $\mathcal{NP}$-hard however, if the width is limited to be less than or equal to a fixed value b then the problem can be solved in polynomial time (though with a large fixed constant in the exponent).

Another problem with these structures is how best to represent them. For example a number of logical and algebraic representations have been used to represent tree decompositions (see [Din97]).

An elegant algebraic representation for tree decomposition is the t-parse operator set. This proposes representing a bounded treewidth structure as a sequence of (graph building) parsable operators which consist of a vertex operator, an edge operator and a boundary join operator (for joining paths into a tree). Algebraic representations of branch decomposition are not so common in the literature, though input structures exist (see Christian [Chr02]).

As mentioned above, branch decompositions of bounded width are 'common denominators' leading to efficient algorithms for bounded parameter values for many natural input restrictions of $\mathcal{NP}$-hard problems [Din97]. For many of these restricted input algorithms the underlying technique used is dynamic programming. There are many examples employing this technique for example see [Chr02] for a set of problems solved in this way.

The outline of this paper, based mainly from [Pro13], is as follows. In the next section we formally define both tree and branch decompositions and mention some of their history. Then in Section 3 we focus on the properties of graphs with low branchwidth, including finding branch decompositions and solving graph problems using them. Next in Section 4 we give our main contribution of an algebraic representation, called b-parse, as a practical data structure for graphs of bounded branchwidth. In Section 5, we give several examples of b-parses for graphs of small branchwidth b. Next in Section 6 we give a general algorithm template for using b-parses to solve (in dynamic programming style) various graph problems, with vertex cover and 3-coloring as concrete examples. We then briefly discuss in Section 7 how to use b-parses in parallel computations, including CUDA/GPU programs. Finally, we end with a short concluding section and open problems.

2 Tree and Branch Decompositions

Robertson and Seymour [RS83, RS84] introduced the concept of bounded width decompositions in the early 1980s in their proof of the Graph Minors Theorem. Their focus was primarily on the use of such structures in finding obstruction sets with respect to graphs under the minor partial order. However, these path/tree decomposition structures have proved useful as 'common denominators' leading to efficient algorithms for bounded parameter values for many natural input restrictions of $\mathcal{NP}$-complete problems. In general many problems can be solved in linear time when the input also includes a bounded width decomposition of the graph.

Hicks points out "the origins of branchwidth and treewidth... are deeply rooted in the proof of the Graph Minors Theorem, formally known as Wagner's conjecture." ([HKK05]). This theorem states that in an infinite list of graphs there exist two graphs H and G such that H is a minor of G. Tree and branch decompositions have since proved important because they allow graphs to decomposed into more manageable pieces in such a way that we know how to reconstruct (and process) the graph.

Courcelle's Theorem [Cou90] as extended by Arnborg et al. [ALS91] states that certain problems (those expressible in terms of monadic second-order logic can be solved in linear time given a branch decomposition. This result has made both tree and branch decompositions very useful as a technique for solving various hard problems.

Branchwidth has some advantages over treewidth as a fixed parameter. As we will see, while finding optimal branchwidth is $\mathcal{NP}$-hard for general graphs (as is finding the optimal treewidth) it has been proven that one can find the branchwidth and indeed a branch decomposition for a planar graph in true polynomial time (i.e. it seems feasible in practice for planar graphs with 100,000 vertices). No such equivalent algorithm exists for treewidth. Moreover, when rooted, branch decompositions always form binary trees giving a much simpler structure to work with (though tree decompositions can be made "nice" or "smooth" [DT06]).

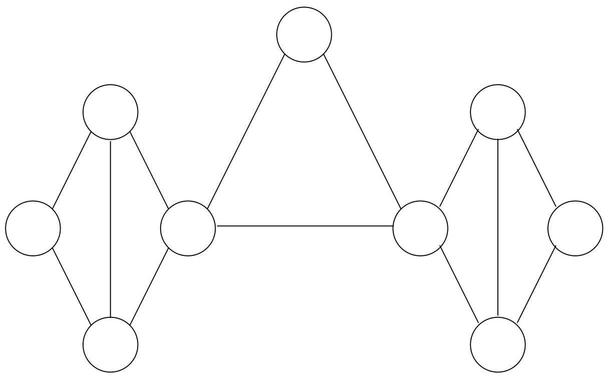


Figure 1: An example graph of treewidth 2.

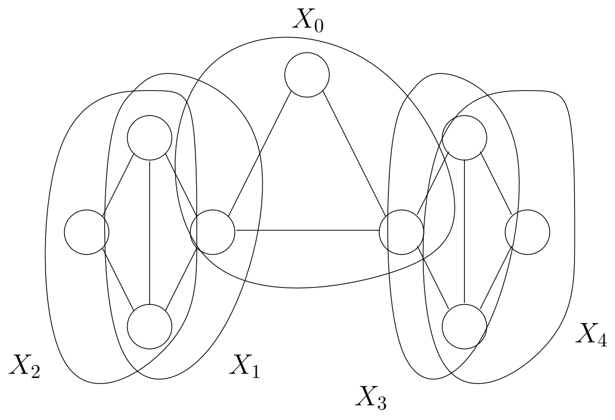


Figure 2: A possible tree decomposition of width 2.

2.1 Tree Decompositions

A tree decomposition is a way to represent a graph G as a set of interlocking subgraphs induced by subsets of the vertices of G (see [Bod93b] for a survey of the concept). These interlocking subsets called bags can be drawn in a pathlike or treelike manner. To be a tree decomposition we require that the union of all these subsets consist of all the vertices of G and that these subsets satisfy some specific requirements that govern how the interlocking behaviour works. See Figures 1, 2 and 3 for an example. Formally we have ([RS84])

Definition 1: A **tree-decomposition** of a graph G is a pair (T, X) where T is a tree

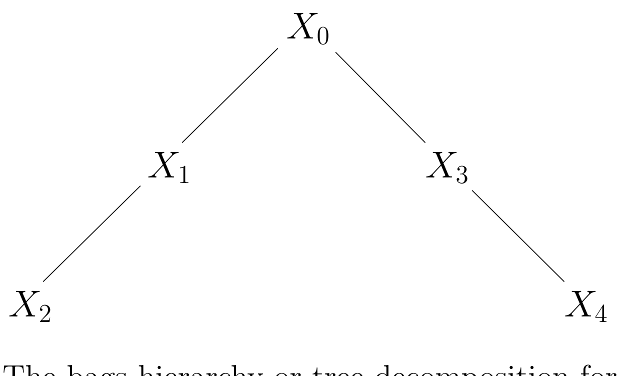


Figure 3: The bags hierarchy or tree decomposition for Figure 2.

and $X = \{X_t \mid t \in V(T)\}$ is a family of subsets of $V(G)$ with the following properties:

1. The union of all the X_t for t in $V(T)$ is $V(G)$.
2. For every edge e of G there exists t in $V(T)$ such that e has both ends in X_t.
3. For every t, t', t'' in $V(T)$ if t' is on the path of T between t and t'' then $X_t \cap X_{t''} \subseteq X_{t'}$.

The key metric or width for tree decomposition is a measure of how close this tree is to a real tree. Indeed treewidth is defined in such a way that a real tree will always have width 1. If we consider the number of vertices for any given bag we can ask what is the maximum size of any bag for a given decomposition. By convention we subtract 1 from this maximum and call it the width of the decomposition. If we then consider all possible tree decompositions for a given graph we call the minimum width across these decompositions the treewidth of the graph. Thus treewidth is a measure of how treelike a graph is—that is, if we view the graph as a tree we can ask how wide is its trunk at the widest point.

Definition 2: The **width** of the tree decomposition is the maximum across all t of $|X_t| - 1$. The graph G has **treewidth** w if w is the minimum such that G has a tree decomposition of width w.

2.2 Branch Decompositions

A branch decomposition is a way to represent a given graph G as a set of interconnected subgraphs induced by subsets of the edges of G. This interconnected set can be drawn as an unrooted tree where the leaves are the original edges of G and the internal nodes represent collections of edges joined together. To be a branch decomposition we require that this tree be ternary for all internal nodes. See Figures 4 and 5 for an example.

Definition 3: Given a graph $G = (V, E)$ a **branch decomposition** is a pair (T, m) where T is a tree with every internal node ternary and m is a bijection $m : E \to \lambda(T)$ from E to the leaves of T (denoted by $\lambda(T)$).

The key metric or width for branch decomposition is a measure of the connectivity between any one of these interconnected edge-induced subgraphs and the other subgraphs. For any two subgraphs (induced by deleting a tree edge of the branch decomposition) we can ask what is the set of vertices that are common to both. We call this set the middleset (see below for a definition). The size of this middleset tells us how connected a subgraph is with the other subgraphs.

The maximum size of any middleset for a given decomposition tells us how interconnected the decomposition is. We call this measure the width of the decomposition. We can then ask what decomposition out of all the possible decompositions of a graph has the smallest width. We call this smallest width the branchwidth and any decomposition whose width is equal to this branchwidth is called optimal. Thus branchwidth can be

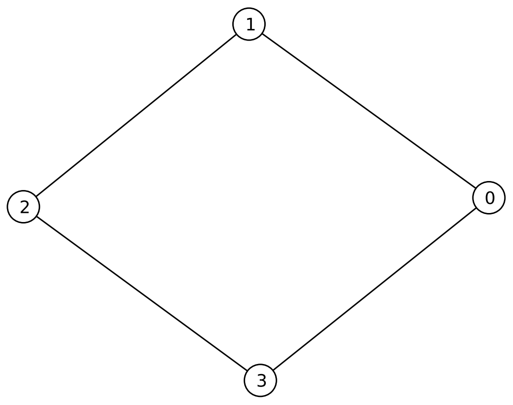


Figure 4: An example graph of branchwidth 2.

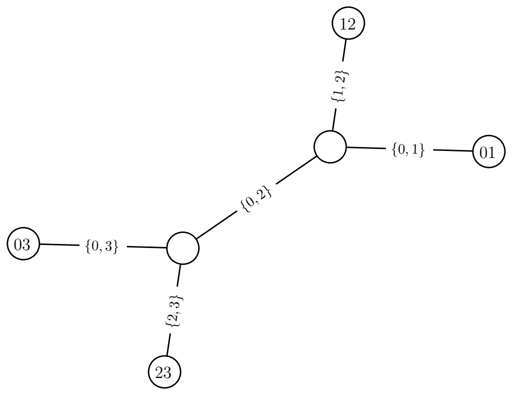


Figure 5: A possible branch decomposition of width 2.

seen as a measure of the best way of decomposing a given graph G. This leads to the following definitions.

Definition 4: For a branch decomposition (T, m) of a graph G if we cut any edge of T we separate $E(G)$ into two disjoint subsets. The set of vertices of $V(G)$ that belong to edges in both of these subsets is called the **middleset**.

Definition 5: The size of the maximum middleset for a branch decomposition is called its **width** and the **branchwidth** of G is the minimum of these widths across all the branch decompositions of G.

We add a **root** to the tree T at an internal edge $\{a, b\}$ by adding a new node r to T and edges from a to r and b to r and removing the edge $\{a, b\}$. The edges $\{a, r\}$ and $\{b, r\}$ incident to the root r will, because of this construction, have the same middleset.

Every internal node of T has three middlesets associated with it. If we root T we can talk about an **outer middleset** along the edge leading from the internal node towards the root and two **inner middlesets** leading into the internal node from "below". All three are "active" middlesets. A "join" at this internal node consists of a composition of the two inner middlesets to obtain the outer middleset.

2.3 How Branchwidth and Treewidth Interrelate

Treewidth and branchwidth are closely related as are tree decompositions and branch decompositions. Robertson and Seymour [RS91] have shown that where $b(G)$ and $w(G)$ are the branchwidth and treewidth respectively, for a graph G it follows that

$$b(G) \leq w(G) + 1 \leq \tfrac{3}{2}\, b(G) \quad .$$

Nevertheless, as we have seen, branchwidth and treewidth are subtly different. In particular treewidth is about how the vertices of a graph interconnect to give a tree-like structure. Branchwidth is about how the edges of a graph interconnect in some (unrooted) binary tree hierarchy with leaves being the edges of G, where the goal is to minimize the shared/separated vertices when cutting the tree's edges. In both cases we are interested in decomposing a graph into a set of interconnected subgraphs but with tree decomposition these subgraphs are induced by subsets of vertices and with branch decomposition they are induced by common vertices of subsets of edges.

Unlike tree decomposition, branch decomposition lends itself well to matroid theory in particular cycle or graph matroids. In this case the elements are the original edges of the graph and the bases are the spanning forests of the graph. The minimal dependent sets of the elements are the circuits or cycles. Tree decomposition, being vertex-based, is not a natural fit with matroid theory (though see Hlineny [Hli06]).

Branchwidth and tangle number are also related as has been shown by Robertson and Seymour [RS91]. (Informally put, for any simple graph the tangle number for the graph is equal to the branchwidth for that graph except when the branchwidth is less than 2.)

3 Characterizing Graphs with Low Branchwidth

In understanding how branchwidth applies to graphs it can be useful to examine graphs with known low branchwidth. Branchwidth has been characterized at low values for a number of categories of graph. Robertson and Seymour [RS91] showed that a graph has branchwidth of 0 if every component is an isolated vertex or edge; that a graph has branchwidth no greater than 1 if every component has at most one vertex with degree greater than 2 (i.e. the family of stars) and that a graph only has branchwidth at most 2 if it does not contain K_4 as minor [RS91].

In addition, Robertson and Seymour have proven that the family of n-grid graphs always has branchwidth n [RS91] and the complete graphs with n vertices have branchwidth $\lceil \frac{2}{3}n \rceil$ [RS91]. This means that the series parallel graphs and outer-planar graphs always have branchwidth at most 2 since they do not contain K_4 as a minor and all forests that are not the union of one factors and stars have branchwidth equal to 2 [HKK05]. It has been further shown (see Hicks [HKK05]) that the chordal graphs have branchwidth bounded by $\lceil \frac{2}{3}cl(G) \rceil \leq b(G) \leq cl(G)$ where $b(G)$ is the branchwidth of G and $cl(G)$ is the maximum clique number for G. Finally, Hicks has shown in his addendum to his doctoral thesis that the Petersen graph has branchwidth equal to 4 [Hic06].

We can also understand the branchwidth of graphs in terms of obstruction sets. As Hicks further observes [HKK05], "the only completely known obstruction sets are for graphs with branchwidth or treewidth 2 and 3. The obstruction set for both graphs with branchwidth at most 2 and graphs with treewidth at most 2 is K_4. Bodlaender and Thilikos [BT99] proved that a graph has branchwidth at most 3 if and only if it does not have K_5, Q_3, M_6, and M_8 as minor." (See also [Rig01] for an analysis of graphs of branchwidth at most 4.)

3.1 Finding Decompositions of Low Width

Before we can analyze or use tree and branch decompositions for a given graph we need to first find them. Ideally, we want to find the optimal decomposition or at least one of relatively low width. In this section we briefly focus on tree decompositions and the next section we look at branch decompositions.

3.1.1 Finding Tree Decompositions

There have been a number of well-known attempts at finding a general purpose algorithm for this, based usually on fixing k. That is all these algorithms work by finding a tree decomposition with less than or equal to fixed k or saying that no such decomposition exists. Early in the history of this concept Robertson and Seymour proposed a treewidth finding algorithm [RS84]. This algorithm either determined that a given graph G had treewidth greater than the given value k or found a tree decomposition of width $4k$. Unfortunately this algorithm took such a long time the exponent was not even computed (see [Ree92]). Later Robertson and Seymour developed a polynomial time $O(n^2)$ algorithm though with again a large constant in the exponent in terms of k [RS86]. There have subsequently been other polynomial time algorithms with more reasonable constants. In the 1990's there appeared $O(n \log n)$ and $O(n)$ algorithms such as those of Reed [Ree92] and of Bodlaender [Bod93a].

3.1.2 Reed

In 1991 Reed proposed results that he claimed could be used to develop a $O(n \log n)$ time algorithm to find tree decompositions in a graph [Ree92]. He builds upon a result given by Robertson and Seymour ([RS90]) which gives a canonical tree decomposition of any graph and in the process gives a min-max theorem which indicates a relationship between treewidth and separators. The ultimate algorithm proposed called "k-tree finder" outputs either a tree decomposition or a subset S of G such that G has no S-separator of order k or less. Reed claims that the algorithm for finding the approximate separators is linear time with the resulting k-tree finder running in $O(n \log n)$ time [Ree92].

3.1.3 Bodlaender

In 1993 Bodlaender proposed a linear time algorithm that given a constant k and graph G determines whether the treewidth of G is at most k and if so finds a tree-decomposition of G with treewidth at most k [Bod93a]. The algorithm does have a reasonably large constant exponent. As Bodlaender points out for $k = 1, 2, 3$ linear time algorithms already existed. There have also been recognition algorithms for graphs with treewidth at most k that use linear time but polynomial space. The main idea of the algorithm proposed by Bodlaender is that vertices are partitioned into two sets—one with vertices of 'low degree' and one with vertices of 'high degree'. It can be shown that for graphs with treewidth at most fixed constant k, there are only few high degree vertices.

3.2 Finding Branch Decompositions

As with tree decompositions having a way to find branch decompositions of low width for a given graph is important. Unfortunately, it has been shown that it is $\mathcal{NP}$-hard to compute the branchwidth of a graph in general (see Seymour and Thomas [ST94] and Hicks [HO10]). That is "we can not hope to have a polynomial-time algorithm to test whether branchwidth is at most k for an input k" [HO10]. However, as with treewidth, it has been shown that if we fix k, as a constant, a polynomial time algorithm can be constructed to test whether a branch decomposition of width k does or does not exist (see [HO10]). For example Oum and Seymour [OS07] have proven that for fixed k one can answer whether the branchwidth is at most k in time $O(m^{8k+c})$ where c is an independent value (see [HO10]). Robertson and Seymour [RS04] have also offered a polynomial time algorithm to approximate the branchwidth of a graph within a factor of 3 [HKK05]. Moreover see Bodlaender [Bod93a] and Reed [Ree97], though as Hicks observes these results are "only of theoretical importance" [HKK05].

3.2.1 An Exception for Planar Graphs

As Hicks points out "when we restrict inputs, the branchwidth can sometimes be computed efficiently. Branchwidth can be computed in polynomial time for circular arc graphs and interval graphs" [HO10]. Most famously, it has been shown by Seymour and Thomas [ST94] that branchwidth and its related variant carving width can be decided for a planar graph in true polynomial time namely $O(n^2)$ time. A carving decomposition is similar to branch decomposition in that it is a ternary tree except that with carving decompositions we arrange the vertices of the original graph as leaves around the tree not the edges as with branch decomposition. We then ask for a given carving decomposition what is the maximum size cutset achieved when we cut any of the internal edges of the tree (where cutset is the set of edges belonging in common to two subsets of vertices). The minimum value for this maximum cutset size across all possible carving decompositions is called the carving width.

Seymour and Thomas [ST94] propose an algorithm that indirectly determines if a low carving width exists for a planar graph by searching for objects called antipodalities that prohibit the existence of small cutsets. They call this the Rat Catcher algorithm after the game of the same name (see [ST94]). They then relate this back to branchwidth by showing that the carving width of the medial of a planar graph is twice the branchwidth of the original graph [ST94]. The medial of a graph is similar to the dual in that it is an inversion of a graph where the edges become vertices and edges exist where two edges in the original graph are incident at a vertex. Cycles exist to show rotation of edges in the original graph around a given vertex.

In fact what Seymour and Thomas [ST94] show is an algorithm that tests whether carving width of size less than or equal to k does not exist for a given medial of an input planar graph [HKK05]. They also offer an algorithm to construct the branch decomposition for low width in $O(n^4)$ time. This has since been improved to $O(n^3)$ by Gu and Tamaki ([GT05] and see [HKK05]). Hicks has pointed out that the original algorithm to compute low branchwidth of Seymour and Thomas is quite space-hungry and offers a slight variation that is more memory friendly at the expense of running in time $O(n^3)$ [HKK05].

3.2.2 General techniques for Forming Branch Decompositions of Low Width

If a graph is not planar or not one of the known categories of bounded branchwidth, we are left with a $\mathcal{NP}$-hard problem to determine if a graph has a branch decomposition of low width. While some exact techniques have been proposed for when k is fixed these have generally not proven to be practicable (see [HKK05]). So the usual strategy followed is heuristic and approximative. As Hicks points out the general approach "to construct a branch decomposition..." is to "start with a partial branch decomposition and refine this decomposition until the tree is ternary. The underlying structure used in constructing a branch decomposition is the separation..." [HKK05]. This separation is a separation of edges into disjoint subsets. Splitting the internal nodes of a branch decomposition causes separations. The usual starting partial branch decomposition is a star with the edges of the original graph arranged as leaves about the star in some order. We then proceed to split the internal nodes in such a way that the separation is minimized.

Hicks, following Cook and Seymour [CS03] and Robertson and Seymour [RS04], defines a partial branch decomposition as extendible if the branchwidth for every subgraph induced by a node in the decomposition is at most the branchwidth of the original graph (see [HKK05]). Hicks then defines a separation as "greedy" or safe "... if the next partial branch decomposition created by the use of the separation in conjunction with a ... split is extendible if the previous partial branch decomposition was extendible" (see [HKK05]). That is a separation is greedy if its use does not lead to worse or higher branchwidth. We then seek algorithms which use greedy separations. As Hicks points out most of these algorithms are heuristic [HKK05].

For example, Cook and Seymour [CS03] give a heuristic algorithm to find branch decompositions of low width based on spectral graph theory and eigenvectors (see also [HKK05]). Hicks has also proposed another heuristic algorithm that finds separations by minimal vertex separators between diameter pairs (see [HKK05]). Hicks also offers another algorithm based on a concept of tangle bases. As he states "the algorithm will either find a branch decomposition whose width is at most $k-1$ or find a tangle basis of order k. This algorithm is utilized repeatedly in a practical setting to find an optimal branch decomposition of a connected graph G whose branchwidth is at least 2, given an input branch decomposition of width at least 3 for G by a heuristic" [Hic05].

3.3 Solving Problems Using Branch Decomposition

"The algorithmic importance of the branch decomposition and tree decomposition was not realized until Courcelle [Cou90] and Arnborg et al. [ALS91] showed that several $\mathcal{NP}$-hard problems posed in monadic second-order logic can be solved in polynomial time using dynamic programming techniques on input graphs with bounded treewidth or branchwidth" [HKK05]. Since this there has been extensive literature on the application of tree decompositions in solving hard problems (see for example [Bod87]). The literature is not so extensive with respect to branchwidth and indeed as Hicks states "overall research in the area has been relatively ignored and vastly unexplored compared to research in tree decompositions" [HKK05].

The usual procedure in using branch decomposition to solve problems is first to find a branch decomposition of small width, then to root it forming a rooted binary tree, and finally to visit all the nodes of the tree in post-order traversal until one is back at the root [HKK05]. While work in this area is not as extensive as with tree decomposition there has been some research. For example, Cook and Seymour [CS03] have offered a branch decomposition based algorithm for solving the ring-routing problem which "arises in the design of reliable cost effective synchronous optical networks" ([HKK05]) and has been incorporated into commercial software. Hicks has also offered algorithms for the Steiner Tree Problem [HKK05] and Maximum Clique [HO10]. Perhaps the best and most extensive survey of problems is that offered by Christian [Chr02].

3.3.1 Christian's Work on Linear Time Algorithms

Christian [Chr02] focuses on problems that can be expressed in terms of monadic second order logic (MSOL) such as Independent Set and Hamiltonian Cycle. Courcelle's Theorem guarantees that these problems can be solved in linear time in terms of the size of the input graph. Christian explains MSOL by first describing first order logic for graphs. First-order logic consists of the usual logic operators plus an operator for whether or not an edge exists between two vertex variables. Second order logic allows operations on sets of vertex and edge variables and allows quantification over these variables (as later defined).

Christian initially describes the concept of the decomposition tree as a formal mechanism on which his algorithms can act (see [Chr02]). The decomposition tree is a generic concept (see [Sze08] and [Lau91]) used also with tree decomposition based algorithms. In essence a decomposition tree is a variety of parse tree with leaf and internal node composition or join operators. Any branch decomposition can be converted into a decomposition tree as Christian shows [Chr02]. One can process a decomposition tree from the leaves upwards in a dynamic programming approach. By following this process one can always reconstruct the original graph [Chr02]. Christian also introduces a lemma which can be summarized in the phrase “once a vertex leaves the middleset we never have to worry about it again” [Chr02].

Lemma 6 (Christian [Chr02]) *Given a branch decomposition (T, m) of a graph G, let $v \in V(T)$ and let T be a rooted tree. If vertex i is in the middleset of the edge (v, l) and not in the middleset of (p, v) for a child l and parent p of v, then i is not in the middleset of any edge in $E(\bar{T_p})$. Moreover, i must be in the middleset of (v, r) where r is the other child of v. (Here $\bar{T_p}$ is all parts of the tree T not including the subtree T_p.)*

Proof: See [Chr02]. □

This in turn means that for a given node of the decomposition tree only the information in the active middlesets is important (where the active middlesets are the middlesets associated with the up to three edges coming out of a current node).

Christian next outlines a framework for his algorithms based on the idea that solutions and their values at the leaves of the decomposition tree plus the join operations are problem specific but one can have a generic algorithm for how these relate. Basically we use an algorithm (see [Chr02]) where if a node of the decomposition tree is a leaf we generate the set of leaf partial solutions and if a node is internal we compare the two sets of incoming solutions and if they are compatible we join them [Chr02]. Christian discusses ways to store the solutions proposing both “fixed partitioning” where the number of solution states is known in advance and “isomorphic partitioning” where we do not know the number of states in advance but we can characterize the solution by membership to a solution state.

An example of a problem which uses the first method is Independent Set and an example which can use isomorphic partitioning is Graph or k-Vertex Coloring. For isomorphic partitioning to work we need to rewrite the partial solutions before testing compatibility [Chr02]. Christian also introduces the idea of dominance or rank as a way of winnowing out similar solutions and finally shows that his framework will run in linear time [Chr02].

Three classes of problem are examined: Vertex Partitioning problems such as Independent Set where the join operation is always the same but the leaf solutions are different; graph coloring using both fixed and isomorphic partitioning; and path partitioning problems such as Hamiltonian Cycle. For the last he requires a third method of solution storage for describing the allowable paths covered by a set of vertices. Christian first states that he generates the input branch decompositions using the algorithms proposed by Hicks (see [Chr02, Hic00]).

To examine Independent Set, Graph Coloring and Hamiltonian Cycle we consider the join operation. For Independent Set this involves including all the vertex values in each incoming solution, provided these are also in the outer middleset and they have the same value across the two incoming solutions for each vertex in the intersection of the vertex sets. A similar approach is followed for the Graph Coloring problem except that we are comparing and joining colorings. On the other hand, the join operation for Hamiltonian Cycle involves “sequentially comparing connectivity at each vertex in the intersection of the active middlesets and determining if the join will violate the restriction on subtours or create a vertex whose degree in the solution is greater than two” [Chr02]. At each stage if the (partial) solutions are compatible and do not break the rules we try to extend them to a full solution.

4 Algebraic Representations of Decompositions

Graphs have been represented in many different ways. Two well-known ways of representing graphs as input are the *adjacency list(s)* and *adjacency matrix*. With the adjacency list format we represent the graph as a list of lists where for each vertex in the graph we list its neighbor vertices. With the adjacency matrix format we form a square matrix indexed by the vertices such that an entry is 1 if there is an edge between two vertices and 0 otherwise. There are many other ways of representing graphs as input data structures including the DIMACS format [DIM12] which explicitly lists the edges or arcs for a graph.

More specifically tree and branch decompositions of graphs have been represented in a number of ways. One of the authors surveys (in [Din97]) a number of algebraic operator sets used to represent tree decompositions. While the t-parse operator set is possibly the most elegant one examined in some detail, two other algebraic representations for tree decompositions—one for which only a constant number of operators is needed (independent of the treewidth) and one which uses a polynomial number of operators per boundary size or width but generates t-boundaried graphs unlike the t-parse which generates $(t+1)$-boundaried graphs (see below for definitions). There do not appear to be any algebraic representations proposed for branch decomposition however we must note the branch decomposition input format of Christian (see above and [Chr02]).

4.1 The t-parse

As mentioned one of the more elegant representations predicated on representing a tree decomposition and its underlying graph as a series of algebraic operators is the t-parse. First we define the $(t+1)$-boundaried graph upon which the t-parse is predicated. See [Din97] for more detail on the following concise presentation.

To generate graphs of bounded path/treewidth we build from graphs that have a distinguished set of labeled vertices.

Definition 7: Formally for a positive integer k, a k-simplex S of a graph $G = (V, E)$ is an injective map $\partial_s : 1, 2, \ldots, k \to$ V. A k-boundaried graph $B = (G, S)$ is a graph G together with a k-simplex S for G. Vertices in the image of ∂_s are called **boundary** vertices (often denoted by ∂). The graph G is called the underlying graph of B.

Given this we can define the t-parse as follows.

Definition 8: The **t-parse operator set** Σ_t consists of the following. There are two unary path operators and one binary tree operator: namely vertex operators $V_t = ⓪, \ldots, ⓣ$, edge operators $E_t = \{\boxed{i\ j} \mid 0 \leq j < i \leq t\}$ and a boundary join operator $\oplus$.

It is known that (see [Din97]) $(t+1)$-boundaried graphs of pathwidth at most t are generated exactly by strings of unary operators from the following operators $V_t \cup E_t$.

A path decomposition can then be represented as for example in Figure 6. The initial $t+1$ boundary is assumed to be present (i.e. the above in Figure 6 should be read as in

$\boxed{1\ 0}\ \boxed{2\ 1}\ \textcircled{1}\ \boxed{1\ 0}\ \boxed{2\ 1}$

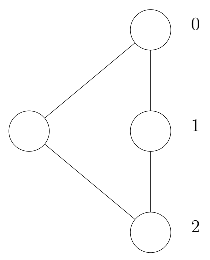


Figure 6: A basic t-parse and its boundaried graph.

$[\ \textcircled{0}\ \textcircled{1}\ \textcircled{2}\ ]\ \boxed{1\ 0}\ \boxed{2\ 1}\ \textcircled{1}\ \boxed{1\ 0}\ \boxed{2\ 1}$

Figure 7: A t-parse including the optional initial boundary vertices.

Figure 7).

To generate the graphs of treewidth at most t the additional binary operator $\oplus$ is added to the unary operators.

The semantics of these operators on $(t+1)$-boundaried graphs G and H are listed in the following table:

Operator	Operation
$G\ \textcircled{i}$	Add an isolated vertex to the graph G and label it as the new boundary vertex i.
$G\ \boxed{i\ j}$	Add an edge between boundary vertices i and j of G (ignore if the edge already exists).
$G \oplus H$	Take the disjoint union of G and H except that boundary vertices of G and H that are labelled the same are identified.

Definition 9: A **parse** is a sequence of operators $[g_1, g_2, \ldots, g_n] \in \Sigma_t^*$ that has vertex operators $\textcircled{i}$ and $\textcircled{j}$ occurring in $[g_1, g_2, \ldots, g_n]$ before the first edge operator $\boxed{i\ j}$, $0 \leq j < i \leq t$.

Definition 10: A ***t*-parse** is a parse $[g_1, g_2, \ldots, g_n] \in \Sigma_t^*$ where all the vertex operators $\textcircled{0}$, $\textcircled{1}$, ..., $\textcircled{t}$ appear at least once in $[g_1, g_2, \ldots, g_n]$. That is a t-parse is a parse with $t+1$ boundary vertices. (see [Din97])

We finally mention the following theorem.

Theorem 11 *The set of treewidth t-parses represents the set of graphs of order at least $t+1$ and treewidth at most t.*

Proof: See [Din97]. □

Note that a t-parse can be represented in a special format for use as programmatic input. The above example $\boxed{1\ 0}\ \boxed{2\ 1}\ \boxed{2\ 0}\ \textcircled{1}\ \boxed{1\ 0}\ \boxed{2\ 1}$ would be in this format represented as

2 (10 21 20 1 10 21)

where $t = 2$. The brackets enclosing the paths and boundary joins occur where a close bracket is adjacent and precedent to an open bracket. For example,

2 ((10 21) (20 21) 0 20)

has one boundary join. Vertex operators are single digit and edge operators are double digit. The above example is the same, assuming left associativity, as writing

$$(\boxed{1\ 0}\ \boxed{2\ 1} \oplus \boxed{2\ 0}\ \boxed{2\ 1}\ )\ \textcircled{0}\ \boxed{2\ 0}$$

4.2 The *b*-parse

It would be useful to have an algebraic representation for branch decomposition that is as elegant and simple as the *t*-parse. In this section a possible algebraic representation for branch decomposition called the *b*-parse is presented. First, a very informal description of the process is given followed by a formal definition.

4.2.1 An Informal Description

As we have seen a branch decomposition of a graph is an un-rooted tree with the edges of a graph as its leaves and where every internal node has degree 3. If we root the tree by splitting at an internal edge we obtain a binary tree with the edges of the original graph as the leaves. At the root we have split at a particular middleset. One way we can represent this tree is to use a variant on newick notation [MSW10] with '+' symbols used in place of commas. So say we have a square C_4 graph as in Figure 8 with possible decomposition as in Figure 9.

We can represent this parse tree as ((01 + 12) + (03 + 23)) where 01, 12, 03 and 23 are edges. This consists of a series of edges and internal nodes connected by composition operations represented by the + operators. The edges 01 and 12 are joined, then 03 and 23 are joined then the resulting internal node (the root in this case) is joined. We can alternately eliminate the need for the brackets by using post-fix or reverse polish notation giving the example as 01 12 + 03 23 + +.

We now consider the middlesets. The middlesets tells us how to correctly reconstruct the graph from the decomposition. For each edge of the tree there is an associated middleset from leaf level up, which we call an active middleset. In addition for every joined pair of leaves or outlier single leaf there is an associated active outer middleset (where outer is pointing towards the root). This outer middleset either consists of all the vertices in the graph induced by the edge or some subset. At the leaf level this middleset consists of both vertices in the leaf (graph edge) or just one or neither. So for simplicity we annotate each leaf and composition operator with the relevant middleset information.

In the above example we have rooted the tree at the point where the middleset is {0, 2} (see Figure 10) so the outer middleset for each subtree from the root is {0, 2}. This means that once we have used the labels 1 and 3 we do not need them any further to represent the tree. This then means that the middleset on each side of the root becomes {0, 2} as required.

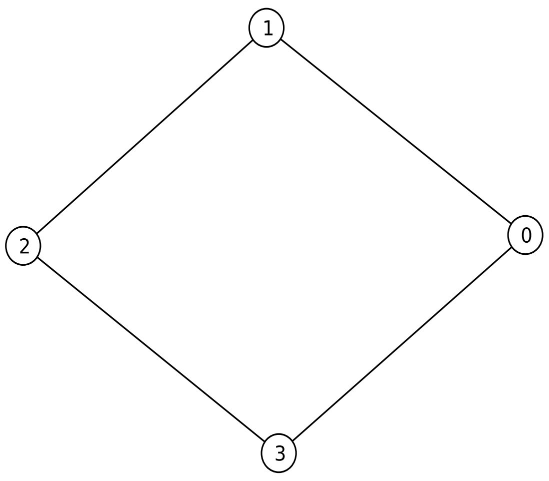


Figure 8: A simple C_4 or square cycle graph with four vertices and four edges.

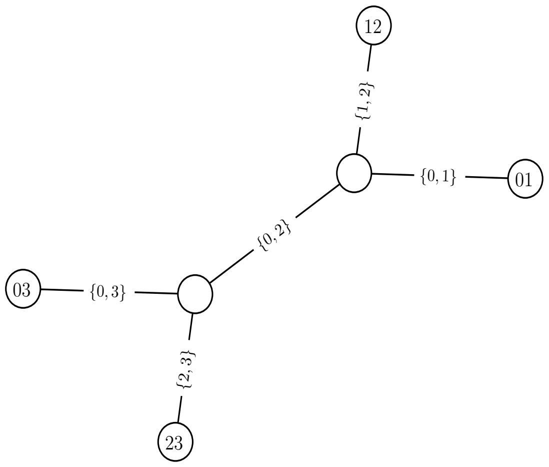


Figure 9: A possible branch decomposition for the C_4 graph with middlesets given as edge labels.

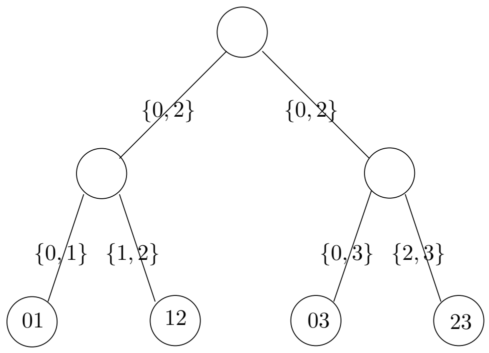


Figure 10: A possible rooted branch decomposition for the C_4 graph with middlesets given as edge labels.

Finally we can label the vertices in each edge/leaf as we read the rooted decomposition from left to right adding new labels by incrementing as needed. Given this and given that the 1 and 3 are not in the final middle set we can represent the decomposition numerically as follows preceding the parse with the width of the decomposition:

2 01 12 $+_{02}$ 03 23 $+_{02}$ $+$ (following the convention that for labels ij, $i < j$)

Furthermore, note some of the vertex labels of C_4 can be replace/reused, such as label 1 in place of label 3.

2 01 12 $+_{02}$ 01 12 $+_{02}$ $+$

Middleset information for the leaves is given in the leaf labels themselves with special labels used if the leaf middleset is empty or has only one member (see below). Any vertex labels not used in the current middleset operator may be subsequently reused (see Lemma 6). Observe that the final join operation always has an empty middleset.

4.2.2 A Formal Definition

The underlying structure for the b-parse is the decomposition tree. In the same way as we build graphs of bounded treewidth from $(t+1)$-boundaried graphs we can build graphs of bounded branchwidth from **k-terminal decomposition trees** (see [Chr02]). While the decomposition tree is a generic concept that can be applied to tree decompositions as well (see [Sze08] and [Lau91]) the following is intended to be specific to branch decompositions.

Definition 12: A **k-terminal** graph $G = (V, E, X)$ is a graph $G' = (V, E)$ with an ordered set $X = \{v_1, \ldots, v_t\}$ of distinguished vertices, called **terminals**, such that $t \leq k$. The set of terminals is called the **boundary**. A k-terminal graph composition function f of arity c is written $G = f(g_1, \ldots, g_c)$ where g_i for $1 \leq i \leq c$ are distinct k-terminal graphs. For any k-terminal graph G, a **k-terminal decomposition tree** is a rooted tree with node properties f_v and g_v such that

- $g_v = G$ if v is the root;
- g_v has only terminal vertices if v is a leaf;

- f_v is a graph composition if v is an interior node; and
- $g_v = f_v(g_{w_1}, \dots, g_{w_c})$ if v is an interior node and $w_1, \dots, w_c$ are its children.

Given this, we can define a **k-decomposition tree of arity** c as a decomposition tree where every subelement g_v is a k-terminal graph and every composition function f_v has arity c (see Christian [Chr02]). In terms of the branch decomposition, the boundary is given by the relevant middleset, the maximum number of terminal vertices will always be $k = b$ (where b is the branchwidth) and the arity will be 2 since a rooted branch decomposition is always binary.

Formally we want to define an algebra or language over a set of decomposition tree operators in a manner similar to the t-parse with $(t+1)$-boundaried graphs (see [Din97]).

Definition 13: The **b-parse operator set** is as follows:

Consider an operator set Σ_b. This operator set is composed of the following operators (where b is the branchwidth or maximum width of the given decomposition): edge operators L (defined below) and composition operators $+_{ij\dots k}$ for $0 \leq i < j < \cdots < k < \left\lfloor \frac{3}{2}b \right\rfloor$. Let G and H be b-terminal binary decomposition trees or subtrees (see definition above) then the operators (read left to right) are defined as:

$G\ L$: Add a new edge to the decomposition where L is one of three possible values depending on the size of the middleset B for this edge:

- If $|B| = 0$ then $L = K_2$ where this is a disconnected edge and its vertices will not be seen again anywhere in the graph.
- If $|B| = 1$ then $L = P_i$ ($0 \leq i < \left\lfloor \frac{3}{2}b \right\rfloor$) where this is a pendant edge one of whose vertices will not be seen again and so only the vertex labelled i is in the outer middleset for this operator.
- If $|B| = 2$ then $L = ij$ ($0 \leq i < j < \left\lfloor \frac{3}{2}b \right\rfloor$) where both of the vertices i and j of this edge will be seen again and are thus in the outer middleset for this operator.

$G +_{ij\dots k} H$ ($0 \leq i < j < \cdots < k < \left\lfloor \frac{3}{2}b \right\rfloor$): Join two subtrees with an outer middleset consisting of vertices with labels $\{i, j, \dots, k\}$ such that $|\{i, j, \dots, k\}| \leq b$.
(Note in postfix notation this operator is written $G\ H\ +_{ij\dots k}$.)

The b-parse operators can be given in reverse polish notation which means that the composition operator will always operate on the previous two subtrees and if a stack is used on the previous two values on the stack. The middleset $\{i, j, \dots, k\}$ must always be a subset of the union of the middlesets of the two preceding entries and the size of this outer middleset must be at most b.

If there is no composition or $+$ operator then there must be at most only one K_2 edge operator. Moreover, the final join operation must have an empty middleset because by this point there are no vertices left which connect to future edges. Finally, in the decimal representation generally only b-parses of width $b \leq 7$ are allowed because as mentioned the labels can be as high as $\lfloor \frac{3}{2}b \rfloor - 1$, numerically. The exception to this is when one can ensure that none of the labels exceed 9. Actually, only in rare cases do we believe that this upper-bound on required labels is reached. The following table shows the maximum number of labels needed for small b.

b	1	2	3	4	5	6	7	8	9	10
$\lfloor \frac{3}{2}b \rfloor$	1	3	4	6	7	9	10	12	13	15

Some further definitions are as follows.

Definition 14: The **composition** (or **join**) operator $+_X$ joins two k-terminal graphs $G_1 = (V_1, E_1, X_1)$ and $G_2 = (V_2, E_2, X_2)$ at their boundaries X_1 and X_2 (i.e. glues together $X_1 \cap X_2$) given a new k-terminal graph $G = (V_1 \cup V_2, E_1 \cup E_2, X)$, where $X \subseteq X_1 \cup X_2$. It identifies at the common vertices for each boundary taking into account the outer boundary X.

Definition 15: A b**-parse** is a sequence $[g_1, g_2, \ldots, g_n]$ in Σ_b^* where all the vertices for a given boundary X appear in the two operand k-terminal graphs, earlier in the decomposition, before each join $+_X$ operator and the size of the largest boundary is b, where labels are taken from the range $0 \ldots \lfloor \frac{3}{2}b \rfloor$.

For computer representations of a b-parse we precede the sequence of operators with an integer b, just like we added t for our computer representation of pathwidth/treewidth t-parses. In most cases we use the reverse polish format of operators so to avoid needing to add and match parentheses to give the tree structure.

A b-parse can be defined as one of Christian's b-decomposition tree of arity 2, which is traversed in post-order form (see next lemma). In turn such a tree represents a rooted branch decomposition.

Lemma 16 *Every b-parse $G_n = [g_1, g_2, \ldots, g_n]$ represents a graph with a branch decomposition of width at most b.*

Proof: This can be proved by induction on the number of '+' operators. If G_n has no '+' operators then by definition G_n has at most one edge operator. This will form exactly one possible branch decomposition. Because the edge is not joined it must be disconnected hence form a single matching. This means the branchwidth for this underlying graph must be 0.

If instead there is one '+' operator then there must be exactly two edge operators. Again there can be only one possible branch decomposition. Moreover these edge operators will be either matchings or pendants meaning that the branchwidth for the underlying graph must be at most 1. This is because if the two edges are joined they can join at most at one vertex. In other words any b-parse with only one '+' operator is represented by exactly one possible branch decomposition and must have width at most 1.

Now suppose G_n has at least one '+' operator. Then we can represent G_n as $G_n = G_1 + G_2$ (or as $G_n = G_1\ G_2\ +$ in post-fix notation) where G_1 and G_2 are both subparses (where a subparse is a valid fragment of a b-parse). We can always do this since the rooted decomposition forms a binary rooted tree. Suppose by hypothesis that G_1 and G_2 both represent branch decompositions. Then G_n must also represent a branch decomposition.

If (T_1, m_1) and (T_2, m_2) are the branch decompositions for G_1 and G_2 respectively we can form T by rooting T_1 and T_2 then adding an edge linking the root nodes of T_1 and T_2 and m will simply be the union of m_1 and m_2. T will be ternary and m will map the edges of the underlying graph to the leaves of T_1 and T_2 and hence T. Thus we are able to form a branch decomposition representing $G_n = G_1 + G_2$. The outer middleset for this decomposition will be, by definition of b-parse, at most b elements of the union of the two inner middlesets. Thus, the width of the decomposition is at most b. □

Lemma 17 *Every branch decomposition of width at most b is represented by some b-parse.*

Proof: The converse that any branch decomposition has an associated b-parse can be shown by construction. Any branch decomposition can be converted into a b-decomposition tree of arity 2. By definition any branch decomposition (T, m) is ternary for its internal nodes. If we root this decomposition we obtain a binary tree.

We give an recursive construction of a b-parse from a rooted branch decomposition. For convenience, we will temporarially keep a record of the outer middlesets, denoted by M, as we construct the final b-parse from subtree b-parses. For the base cases, the leaves, are initially set to be b-parses by the following simple rules:

- If the leaf node represents an isolated edge, we use K_2 as the b-parse and $M = \emptyset$.
- If the leaf node represents a pendent edge, we use P_0 as the b-parse and $M = \{0\}$.
- If the leaf node represents an edge with both ends incident to other edges, we use 01 as the b-parse and $M = \{0, 1\}$.

The labels of these leaf node b-parses, will change as we build the final b-parse by combining subtree b-parses. We first show that at most $\lfloor \frac{3}{2}b \rfloor$ labels are need to be active at a given join. Consider three incident edges in the branch decomposition, with two inner middlesets, M_1 and M_2, and an outer middleset M_3 in the rooted version. We know that $|M_i| \leq b$, for $i = 1, 2, 3$ and $M_i \subseteq M_j \cup M_k$ for all distinct i, j, k. The largest number of labels occur when $M_i \cap M_j \cap M_k = \emptyset$ (i.e. each vertex label is only on two of the three incident edges). Thus, we have at most $3b$ labels and each share two middlesets for a maximum total of $\lfloor \frac{3}{2}b \rfloor$ active labels.

Now consider the recursive construction where we have left and right subtree b-parses, (T_1, m_1) and (T_2, m_2), respectively, where m_1 and m_2 are their outer middlesets and mapped labels—these will be inner middlesets for the join. We assume that we can extract the mapping between the graphs vertices to the label sets m_i. We now create a mapping (of vertices to labels) for the outer middleset of the join. Again, with M_1, M_2 and M_3 as the sets of middleset vertices we need to give a bijective map m_3 from $M_1 \cup M_2 \cup M_3$ to labels $\{0, 1, \ldots |M_1 \cup M_2 \cup M_3| - 1\}$. For convenience, we assume the vertices are linearly ordered and the mapped labels are in same order. The b-parse of the join will be $\texttt{relabel}(T_1, m_1, m_3)$ $\texttt{relabel}(T_2, m_2, m_3)$ $+_{m_3(M_3)}$, where the function `relabel` recursively changes the subtree b-parses to end with their outer middlesets to be consistant with the mapping m_3. This proceedure is illustrated in the next example. □

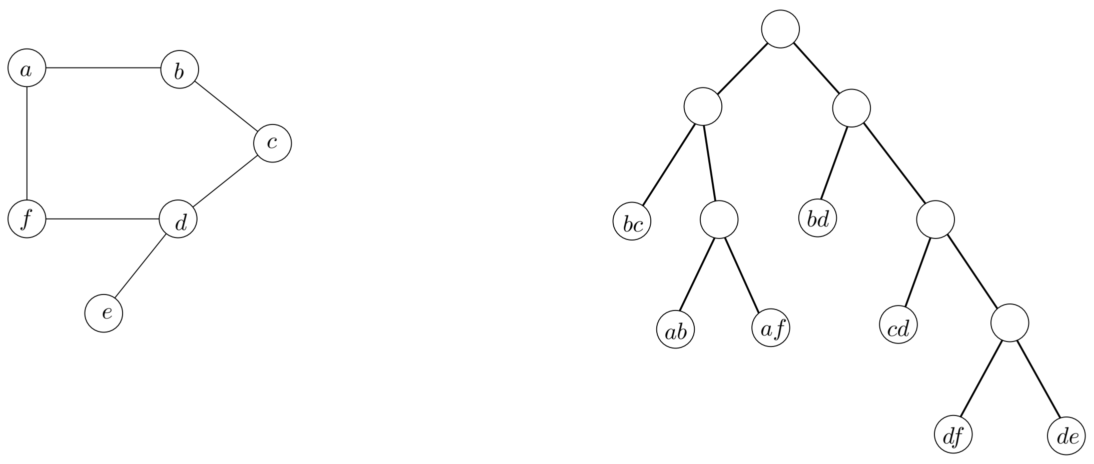


Figure 11: A graph and a branch decomposition of width 3.

Example 18 For the branch decomposition given in Figure 11 we have the following construction of a b-parse from leaves upwards.

(bc) (ab) (af) (bd) (cd) (df) (de)

(01) (01) (01) (01) (01) (01) (P0)

(01) (01 02 +12) (01) (01) (01 P0 +01)

(01 01 12 +02 +012) (01) (01 12 P1 +12 +012)

(01 01 12 +02 +012) (02 12 23 P2 +23 +123 +013)

01 01 12 +02 +012 03 13 23 P3 +23 +123 +013 +012 +

Combining the above two lemmata we obtain justification for our b-parse representation.

Theorem 19 *The set of b-parses represents the set of graphs of branchwidth at most b.*

Proof: Combining the above Lemmata 16 and 17. □

As a result we can be sure that any b-parse is a representation of a valid branch decomposition. From now on we can talk in terms of b-parses with underlying branch decompositions.

5 Examples of b-parses

As mentioned earlier, graphs of bounded branchwidth have been characterized up to branchwidth 4 by forbidden minors. In addition we can obtain branchwidth for graphs higher than 4 by examining the complete and grid-graphs (among others). In this section we will give some examples of b-parses for graphs of known branchwidth.

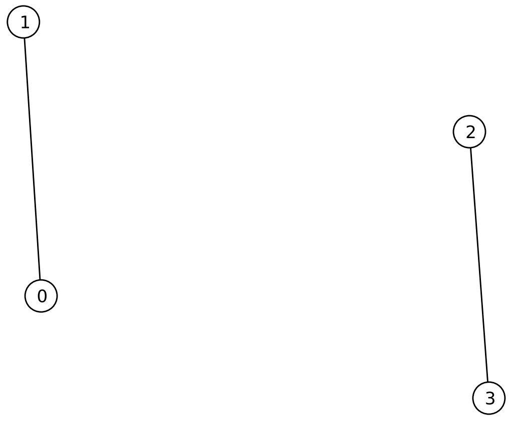


Figure 12: A pair of matchings or K_2.

5.1 Branchwidth 0

We know that the only graphs with branchwidth of 0 are those whose components consist of isolated vertices or disjoint edges. For example a pair of disjoint edges (see Figure 12) is represented by the b-parse (see Figure 13).

$0\ K_2\ K_2\ +$

5.2 Branchwidth 1

Branchwidth 1 graphs are those whose components consist of stars. For example the star S_4 (see Figure 14) is represented by the b-parse (see Figure 15).

$1\ P_0\ P_0\ +_0\ P_0\ P_0\ +_0\ +$

5.3 Branchwidth 2

Branchwidth 2 graphs are defined to be all those graphs which do not contain K_4 as a minor. This includes the cycle graphs, the series-parallel graphs and the outer-planar graphs as well as path graphs with more than three vertices. The cycle graphs are interesting since apparently the width of every middleset in a branch decomposition of a cycle graph is always exactly 2. This may be because with cycle graphs every edge is connected to exactly 2 other edges. For example consider C_4 the square graph (see Figure 16) with b-parse representation (see Figure 17).

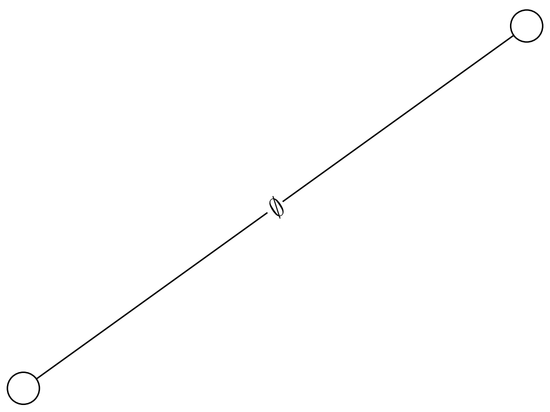


Figure 13: The branch decomposition for two matchings with middlesets given as edge labels. In this case there is only one middleset and it is empty.

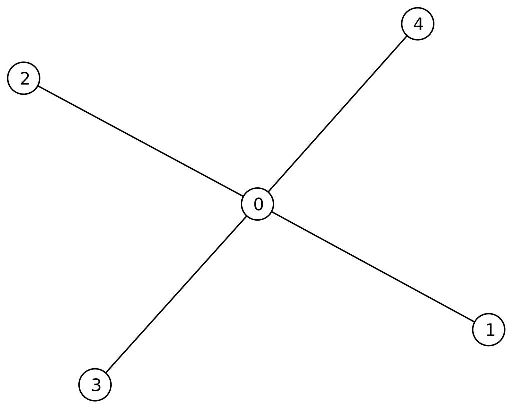


Figure 14: A star or S_4.

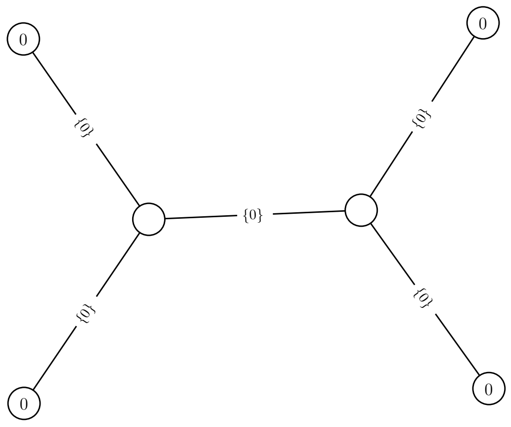


Figure 15: A possible branch decomposition for S_4 with middlesets given as edge labels.

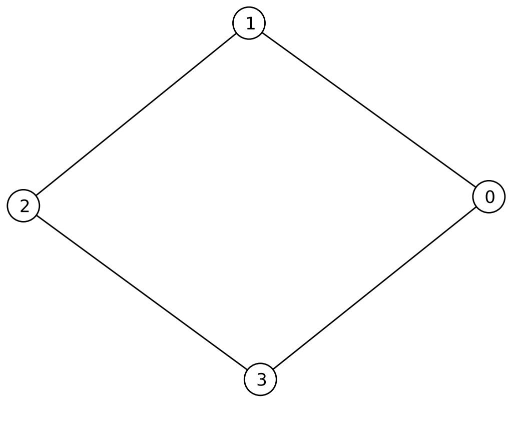


Figure 16: C_4 the square graph.

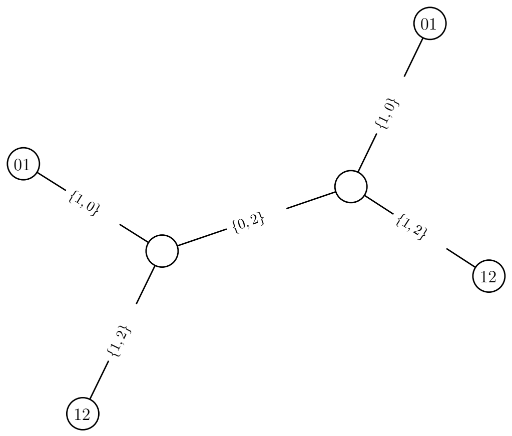


Figure 17: A possible branch decomposition for C_4 with middlesets given as edge labels.

2 01 12 $+_{02}$ 01 12 $+_{02}$ $+$

As a slightly more complicated example, we also see that the cycle C_6 can be represented as the following *b*-parse.

2 01 12 $+_{02}$ 12 $+_{01}$ 02 01 $+_{12}$ 02 $+_{01}$ $+$

5.4 Branchwidth 3

The graphs of branchwidth three have been characterized by Bodlaender and Thilikos (see above and [BT99]). In particular the graphs of branchwidth 3 include K_4 (see Figure 18) which can be represented by the *b*-parse (see Figure 19).

3 01 12 $+_{012}$ 02 23 $+_{023}$ $+_{013}$ 03 13 $+_{013}$ $+$

5.5 Branchwidth 4

To identify graphs with branchwidth greater than three we have some formulae provided by Robertson and Seymour [RS91]. In particular they proved that the complete graph K_n with n vertices has branchwidth $\lceil \frac{2}{3}n \rceil$ (see [RS91]). In addition, Hicks has shown that the Petersen graph has branchwidth of 4 [Hic06]. For example, K_5 (see Figure 20) is represented by the following *b*-parse (see Figure 21).

4 01 12 $+_{012}$ 13 14 $+_{134}$ $+_{0234}$ 03 04 $+_{034}$ 24 02 $+_{024}$ $+_{0234}$ 23 34 $+_{234}$ $+_{0234}$ $+$

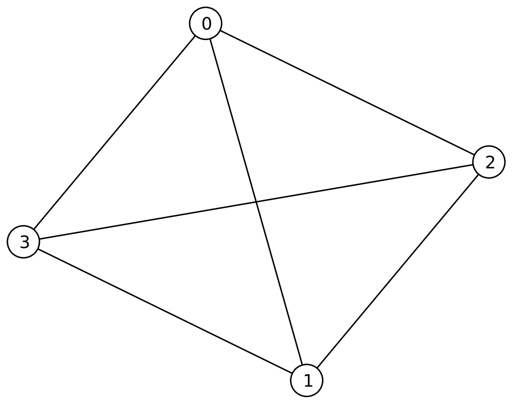


Figure 18: The complete K_4 graph.

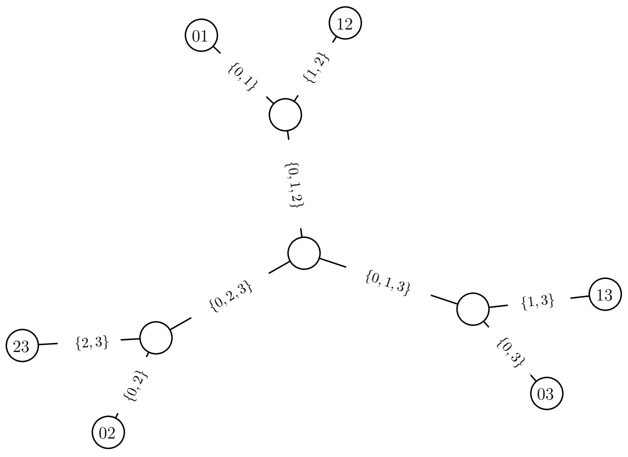


Figure 19: A possible branch decomposition for K_4 with middlesets given as edge labels.

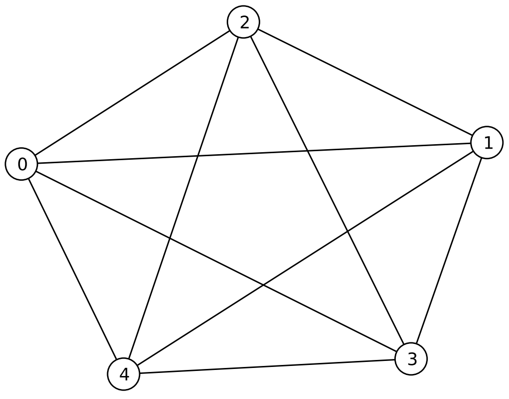


Figure 20: The complete K_5 graph.

5.6 Branchwidth ≥ 5

To find branch decompositions and so b-parses of width greater than four we need only examine the complete graphs. For example, K_7 will have branchwidth 5, K_8 and K_9 will have branchwidth 6 and so on. In the absence of an efficient and optimal b-parse generator (described later) exact b-parses need to be formed manually. This manual method is often prohibitive for graphs of branchwidth greater than four.

6 Solving Problems with the b-parse

As has been mentioned, tree and branch decompositions have proved important because they allow graphs to be decomposed into more manageable pieces in such a way that we know how to reconstruct the graph. Courcelle's Theorem [Cou90] states that graph problems expressible in the language called monadic second order logic are decidable in linear time given a tree decomposition. Arnborg et al. (see [ALS91]) have extended this result to cover a wider range of problems including those using branch decompositions.

These results have made tree and branch decomposition very useful as techniques for solving various hard problems. Indeed this technique is an example of the family of problem-solving techniques known as Fixed Parameter Tractability (FPT) (see [DF99, FG06]). FPT is predicated on the idea that complex data often has structure that, if given as a parameter, can render problems using the data as input much easier to solve.

6.1 The General Approach

The usual strategy for solving graph problems using either branch or tree decompositions, and the one used by Courcelle [Cou90] and Arnborg et al. [ALS91], is based upon tree automata theory and dynamic programming. Tree automata are tree-based variants of

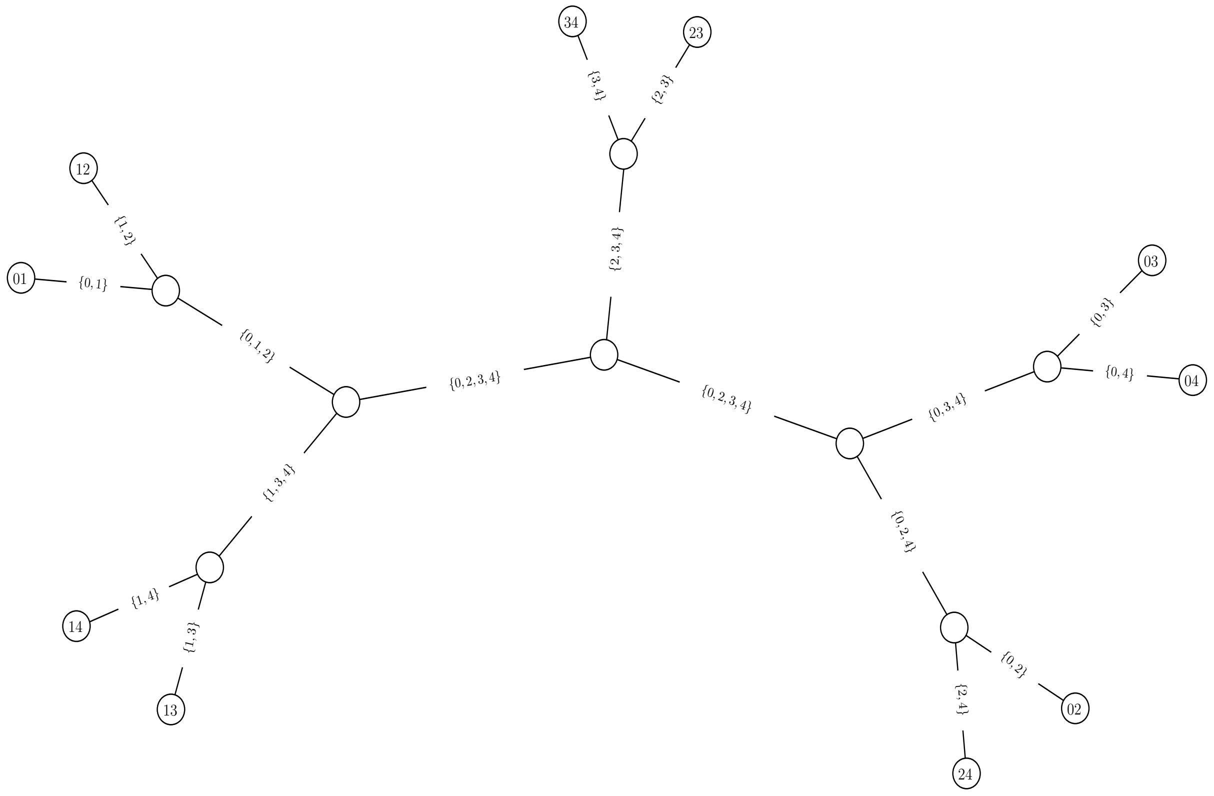


Figure 21: A possible branch decomposition for K_5 with middlesets given as edge labels.

finite automata which are in turn state machines for processing strings over a given alphabet. One can use a parse tree derived from a t-parse as input for a bottom-up leaf to root tree automaton. The Myhill-Nerode Theorem (see [DF99]) can be applied to tree automata to show that for languages of a given type tree automata must exist recognizing them and they can be partitioned into a finite set of equivalence classes which can be mapped to the states of the recognizing tree automaton. This feature of the languages thus recognized is sometimes called right congruence and forms the equivalence relation partitioning the language (e.g. see [Den01]). In turn one can use dynamic programming on the tree to solve the problem.

The basic idea behind the dynamic programming approach is that one can solve a number of otherwise complex problems by implicitly exploring the space of all possible solutions carefully decomposing problems into a series of subproblems and then building up correct solutions to larger and larger subproblems [KT06]. At the core of this approach is the principle of optimality whereby non-optimal subsolutions can never lead to optimal end solutions.

6.1.1 Minimum Vertex Cover and t-parses

Cattell and Dinneen [CD94] give a good example of using bounded pathwidth decompositions to solve intractable problems such as minimum vertex cover. They use the t-parse algebraic representation of the input graph with bounded pathwidth t, as explained in Section 4.1. In this form the t-parse consists of two types of operators—the vertex operator whereby a vertex is added to the boundary of the input graph and an edge operator whereby an edge is added between two boundary vertices. Each time a vertex operator is executed the previous vertex is pushed into the interior of the graph.

The authors [CD94] propose a finite-state algorithm, which is a dynamic program, that makes a single left to right scan of a t-parse $G_n = [g_1, \ldots, g_n]$. The computational process resembles a finite state automaton in that it accepts words over the pathwidth operators of Σ_t. If m is the current scan position of the algorithm on input G_n then the state table at operator g_m is indexed by each subset S of the boundary. In this case there are 2^{t+1} different entries defined as follows:

$$V_m(S) = min\{|V'| : V' \text{ is a vertex cover of } G_m \text{ and } V' \text{ contains } S\}$$

Essentially this “minimal state” algorithm consists of updating the specified state information structure for each parse token operator read. The algorithm begins by setting the sizes for the minimal vertex covers on the empty graph $G_{t+1} = [0, 1, 2, \ldots, t]$ for all subsets S of the initial boundary (that is $V_1(S) = |S|$). The algorithm ends when the last token is read.

In the case of minimum vertex cover the resulting value given pathwidth t is the minimum value across all the entries in the final state structure. This method very much follows the dynamic programming approach. It can be extended as others have shown to process input graphs which have bounded treewidth decompositions. The method can and has also been generalized to process many types of related intractable problem by simply changing the logic used in processing the token operators. Examples include using this method to solve the Minimum Spanning Caterpillar problem (see for example [DK10]).

6.2 Using the Decomposition Tree

As we have seen the underlying algorithmic mechanism for reconstructing graphs from branch decompositions and the b-parse is the decomposition tree (just as the $(t+1)$-boundaried graph is the underlying mechanism for working with t-parses). We can use the decomposition tree as input for a tree automaton, which we traverse leaf to root, in a similar manner to which the implicit parse tree for a tree decomposition and t-parse acts as a input for leaf to root tree automaton.

The decomposition tree for the b-parse is both more complex and simpler than the underlying parse tree for the t-parse. Initialization is more complex since there are three types of leaves. Note we still refer to leaf 'operators' even though strictly speaking we are talking about initialization. Movement up the tree is simpler since there is only the one operation '+'. However, the b-parse '+' operation is more complex than the t-parse boundary join $\oplus$ operation. With boundary join the boundary for both incoming states is always the same so we can do a one-to-one match in many situations. With the b-parse '+' operation the middlesets and so boundaries of the two incoming states are usually different so we need to consider all combinations of the two boundaries. So for example for minimum vertex cover (as we will see) we consider all combinations of the classes of valid vertex cover for each state and for 3-vertex coloring we consider all combinations of valid colorings for each state.

This highlights a further advantage that branchwidth has over treewidth. Branchwidth forms only an upper bound for the sizes or widths of the middlesets in a branch decomposition whereas treewidth is generally a fixed value used as a global bound when processing a graph decomposition. With both branchwidth- and treewidth- based algorithms the amount of processing involved is determined by the size of the boundary as well as the order of the graph. Branchwidth based algorithms for solving graph problems may often involve less processing at any given step than an equivalent treewidth based algorithm.

6.3 A Generic Algorithm

We now show how one can use the b-parse and its underlying decomposition tree to solve hard problems. First we outline a generic algorithm for processing a b-parse for some graph G. This will be similar in some ways to that used in [CD94] but uses a stack rather than a table as the working data storage mechanism since when scanning a b-parse we are in fact traversing the underlying decomposition tree. As with pathwidth t-parses, we scan the b-parse from left to right reading the operators.

We define our values as a collection V, given some boundary B, for some position along the input b-parse. We can call V the state or solution set for its current b-parse operator. In the present case the boundary B will be defined by the relevant middleset. Moreover, for any two value sets V' and V'' the associated boundaries B' and B'' need not be the same. This in turn means that the actual value as well as the position of the boundary members matters. That is middlesets so boundaries $\{a, b\}$ and $\{b, c\}$ have the same number of elements but are not the same. This will have an impact on how the membership of V is handled. Given a b-parse $[g_1, \ldots, g_n]$ our algorithm becomes as follows.

algorithm `processBParse`

```
    Input:   width $b$, parse $[g_1, \dots, g_n]$ and a stack
begin
    initialize solution set $V$ for the first operator $g_1$ and push $V$ onto the stack
    while there are more operators $g_i$ do
    begin
        if $g_i$ is a leaf operator $L$ then
        begin
            initialize solution set $V$
            push $V$ onto the stack
        end
        else if $g_i$ is a join operator $+_B$ where $B$ is the boundary for the join then
        begin
            pop solution sets $V'$ and $V''$ from the stack
            join $V'$ and $V''$ and produce $V$ given the boundary $B$
            push $V$ onto the stack
        end
    end
    pop the last element $V$ from the stack
    return $V$
end
```

Initialization and composition are problem specific as is, of course, the representation of the solution sets. We now use this framework in analyzing two hard problems: minimum vertex cover and 3-vertex coloring. In each case we first outline the problem, then consider how the solution sets can be best represented including how the leaf values will be initialized. We then consider how the join or '+' operation should be applied in each case and what the final result should be. Finally we give an example for each.

6.4 Minimum Vertex Cover

The decision problem vertex cover asks:

Input: Let $G = (V, E)$ be a graph and $k \in \mathbb{N}$
Question: Does there exist $V' \subseteq V$ such that every edge in E has a vertex in V' and $|V'| \leq k$?

The optimization version of this problem (where we ask what is the minimum k such that the above is true) shall be referred to as MINIMUM VERTEX COVER.

It can be shown that MINIMUM VERTEX COVER can be represented in monadic second order logic (see Arnborg et al. [ALS91]). This means that, by application of Courcelle's Theorem [Cou90], we know that a linear time algorithm exists for solving the MINIMUM VERTEX COVER problem using a branch decomposition.

6.4.1 The Solution Sets for Minimum Vertex Cover

To formulate a method for solving this problem using branch decomposition and the b-parse we need first to define what a solution set is and, in particular, to specify the solution values for the base case of an edge. Let H be a non-empty subgraph of some graph G induced by some subset of the edges of G. Let B be the middleset or boundary for H (B can be empty).

For the MINIMUM VERTEX COVER problem a solution set V for H represents the set of all possible valid vertex cover sets for H partitioned by subsets S of the boundary B. The actual solution values for V for each S are defined as the size of the minimum valid vertex cover set such that the set intersected B is S. Thus we define

$$V[S] = \min\{|W| : W \text{ is a valid vertex cover and } S = W \cap B\}$$

There are $2^{|B|}$ entries for V. If no such vertex cover W exists for a given S then we define $V[S] = \infty$ where V, B and S are defined as above.

The smallest non-empty subgraphs of G induced by the edges are the subgraphs consisting of a single edge of G. Consider an edge with vertices a and b. We now explicitly define the solution values for this subgraph depending on the middleset where B represents the middleset (or boundary) for the subgraph consisting of the edge $\{a, b\}$.

A Disconnected Edge or K_2: If the middleset $B = \emptyset$ (so neither a nor b are in the middleset) then there is only one possibility for S namely $S = \emptyset$. So we consider all the vertex covers for this edge $\{a, b\}$ that contain $\emptyset$ which is all of the vertex covers for this edge. The minimum vertex cover for this set of vertex covers must then be the size of a set containing only a or only b and so have the value 1.

A Pendant Edge: If there is one vertex a or b for this edge $\{a, b\}$ call this a in the middleset or boundary B (so $B = \{a\}$) then we have two possible values for S namely $S = \emptyset$ and $S = \{a\}$. The minimum valid vertex cover set for $S = \emptyset$ is a vertex cover containing the other vertex b with a size of 1. The minimum valid vertex cover set for the $S = \{a\}$ is a vertex cover containing only a with a size of 1.

A Fully Connected Edge: If both vertices a and b are in the middleset so $B = \{a, b\}$ then we have four possible values for S: $\emptyset$, $\{a\}$, $\{b\}$ and $\{a, b\}$. However, there are no valid vertex cover sets for this edge which contain neither of its vertices so we set this solution value to ∞. The minimum valid vertex cover for the vertex covers containing only a consists of a on its own with a size of 1. Similarly for the vertex covers containing only b. Finally, the minimum valid vertex cover for the vertex covers that must contain both a and b is the set of a and b itself which has a size of 2 (compare with the initialization of $V_m(S)$ in Cattell and Dinneen [CD94] where $V_1(S) = |S|$).

6.4.2 Defining the Join Operation for Minimum Vertex Cover

Given these base graphs or edges we proceed to join them into composite subgraphs. We then join the resulting compositions until we reach the root of our rooted decomposition tree. Each join will consist of two specified input solution sets and an outer middleset for the join. The root solution set will always have an empty outer middleset since this represents the set of vertex covers for G itself and G does not connect to any other vertices.

As we have seen, each output solution set value is defined by its membership of some subset S of the output solution middleset B at some point during the processing of the input b-parse. If B' and B'' are the middlesets for the two input solution sets we calculate the matching values of $S \subseteq B$ given B' and B'' as:

$S = (S' \cup S'') \cap B$ for some $S' \subseteq B'$ and $S'' \subseteq B''$

We calculate the minimum value for each value of S as follows. If V is the output solution set, V' and V'' are the input solution sets, and solution value $V[S]$ is the solution set V indexed by $S \subseteq B$ then

$$V[S] = min\{V'[S'] + V''[S''] - |S' \cap S''| : \forall S' \subseteq B' \ \forall S'' \subseteq B''\} \text{ s.t. } S = (S' \cup S'') \cap B$$

We have seen that the initial values assigned to the leaves are optimal vertex covers given a particular membership in the current middleset. So we need only show that the join operation will result in an optimal (representative) set of vertex covers for the current subgraph.

Theorem 20 *Given two input solution sets which consist of sizes of minimized valid vertex covers for their associated subgraphs the above join operation will result in an optimal solution set for the joined graph.*

Proof: Consider a subgraph G with outer middleset B and two input solution sets V' and V'' representing two subgraphs G' and G'' with associated middlesets B' and B''. We need to show two things: (1) that the size of any vertex cover W of G, such that $W \cap B = S$, is at least as large as $V[S]$ that is produced by our solution rule and (2) there exists a vertex cover W of G, such that $W \cap B = S$ that is of size $V[S]$.

For our claim (1), let W be any vertex cover of G. Let $S = W \cap G$. We know $W' = W \cap G'$ and $W'' = W \cap G''$ are vertex covers for G' and G'' with boundary vertices $S' = W \cap B'$ and $S'' = W \cap B''$. Also by dynamic programming subproblems, we have $|W'| \geq V'[S']$ and $|W''| \geq V''[S'']$. By observing, we (may) have an overlap of covering vertices $S' \cap S''$, in G'' and G'', we can conclude the following.

$$|W| = |W'| + |W''| - |W' \cap W''| \geq V'[S'] + V''[S''] - |S' \cap S''| \geq V[S]$$

Thus, the first part of the theorem is shown.

For our claim (2), consider a minimum vertex cover W of G such that $S = W \cap B$. We claim that $W \cap G'$ and $W \cap G''$ are minimum vertex covers for G' and G'', respectively, where $S' = W \cap B'$ and $S'' = W \cap B''$. If not, WLOG, assume there is a smaller vertex cover X of G', such that $S' = X \cap B' = W \cap B'$. But this would contradict the representative value of $V'[S']$ of G'. Thus, the solution rule shows that $V[S] = V'[S'] + V''[S''] - |S' \cap S''|$ for this minimum vertex cover W, when combined from the minimum covers $W \cap G'$ and $W \cap G''$ of the two subgraphs. $\square$

6.4.3 Time Complexity for Minimum Vertex Cover

The time complexity for any linear dynamic programming branchwidth based algorithm will be by definition $O(f(b)m)$ where b is the branchwidth of the parse and m is the number of edges (see [Cou90] and [ALS91]). Usually $f(b)$ is some exponential function of b. For MINIMUM VERTEX COVER this will be $O((2^b + 2^{2b})m)$ where for the input graph G, $m = |E(G)|$. For each join we loop twice through the two input solution sets whose size each will be at most 2^b. There will be $O(m)$ joins (at least one half of the number m of edge operators). In addition there will be $O(2^b m)$ leaf operations. The

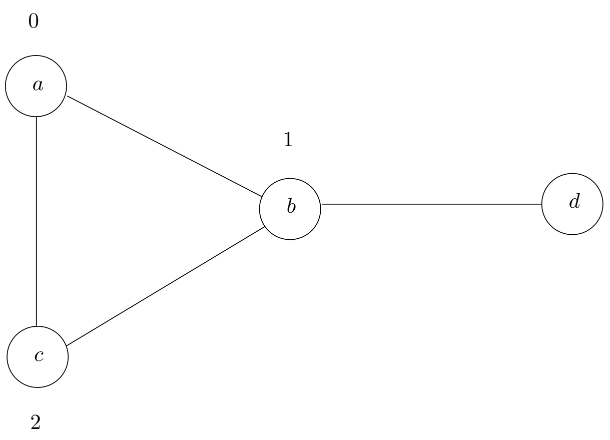


Figure 22: An example graph with matching labels.

space complexity for the MINIMUM VERTEX COVER algorithm will be $O(2^b m)$ since each solution consists of at most 2^b constant sized entries and at any time we need to remember $O(m)$ previous entries.

6.5 An Example for Minimum Vertex Cover

Consider the following graph of branchwidth 2 (see Figure 22). This has a possible b-parse

$2\ 01\ P_1\ +_{01}\ 02\ 12\ +_{01}\ +$

For a possible sequence of solution sets arranged in a tree mirroring the underlying decomposition tree consider the diagram in Figure 23. Each solution set is given with its matching b-parse operator below.

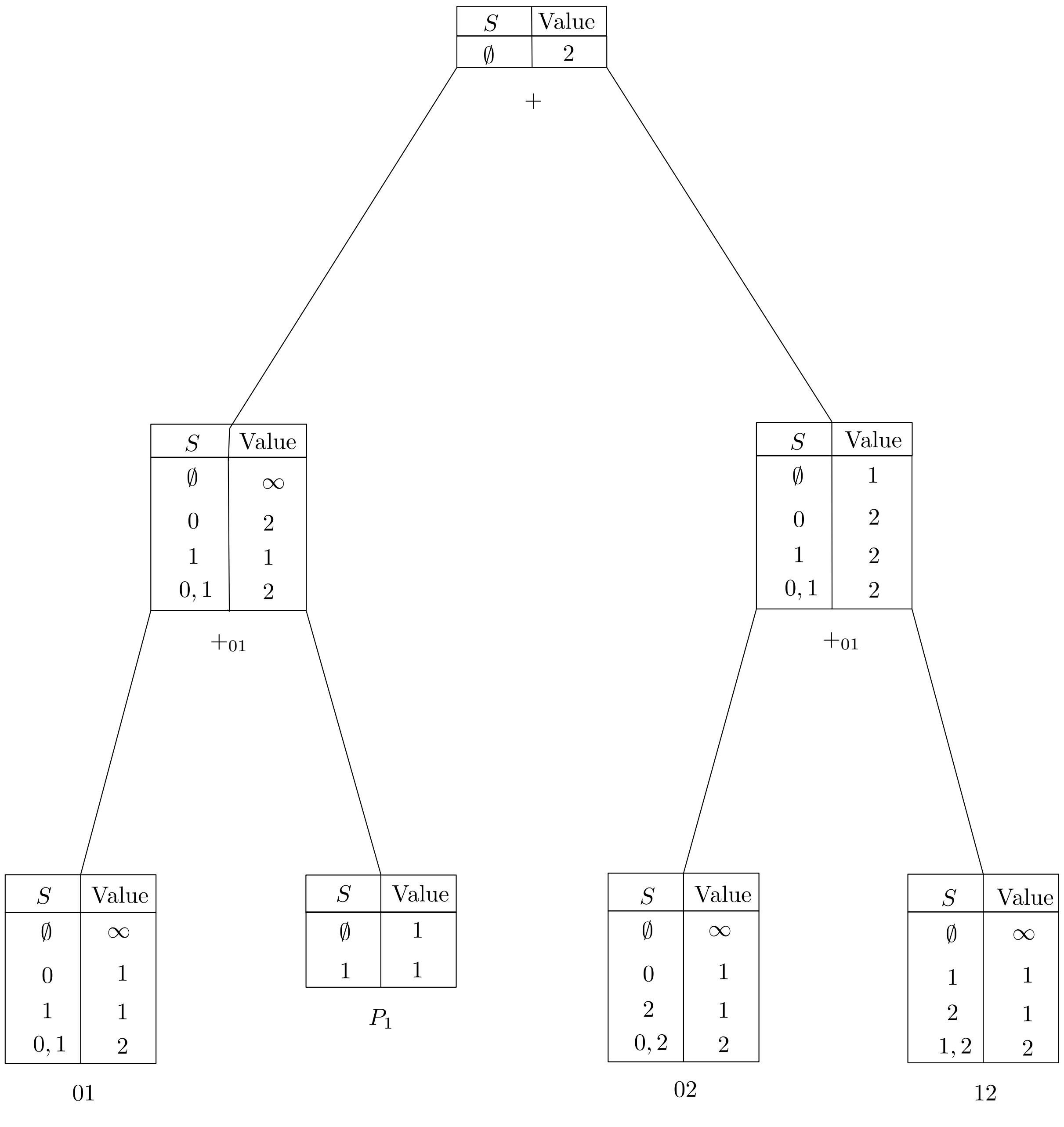
In processing this for the MINIMUM VERTEX COVER problem we start with the two edges 01 and P_1 which can be matched to the edges ab and bd respectively in the sample graph. These have middlesets of $\{0, 1\}$ and $\{1\}$ hence we know the solution sets V' and V'' are as seen in lower left part of the figure.

We now join these two solutions. We know from the parse that the outer middleset for this first join will be $\{0, 1\}$ hence the subsets and so the values of S for the joined solution set will be $\emptyset$, $\{0\}$, $\{1\}$ and $\{0, 1\}$.

We begin by comparing each of the solution values with the other. For example we compare $V'[\{0\}] = 1$ with $V''[\{1\}] = 1$. Now $S = (\{0\} \cup \{1\}) \cap \{0, 1\} = \{0, 1\}$ (see the section above) and the value for $V[\{0, 1\}] = min\{V[\{0, 1\}], V'[\{0\}] + V''[\{1\}] - |\{0\} \cap \{1\}|\} = min\{V[\{0, 1\}], 2\} = 2$ (in this case). We repeat this for all the solution set values for V' and V'' and it can be seen we obtain the resulting solution set V.

This tells us that if we include both 0 and 1 in the vertex cover then the smallest valid vertex cover we can obtain is of size 2. However, if we do not include 0 then we can still include 1 and obtain a vertex cover of size 1 given the subgraph formed by the joining of edges 01 and P_1.

We next join this subgraph with the edges 02 and 12 (which are matched by edges ac and bc in the sample graph) in the same way as above obtaining the solution set V seen in the right part of the figure. Finally, we join these two subgraphs with the solution sets V' and V'' to obtain the result V seen at the top of the figure.



S	Value
$\emptyset$	2

S	Value
$\emptyset$	∞
0	2
1	1
$0, 1$	2

S	Value
$\emptyset$	1
0	2
1	2
$0, 1$	2

S	Value
$\emptyset$	∞
0	1
1	1
$0, 1$	2

S	Value
$\emptyset$	1
1	1

S	Value
$\emptyset$	∞
0	1
2	1
$0, 2$	2

S	Value
$\emptyset$	∞
1	1
2	1
$1, 2$	2

$$2\ 01\ P_1\ +_{01}\ 02\ 12\ +_{01}\ +$$

Figure 23: A possible sequence of solution sets for this *b*-parse for MINIMUM VERTEX COVER arranged in a tree with each *b*-parse operator given below its matching solution set.

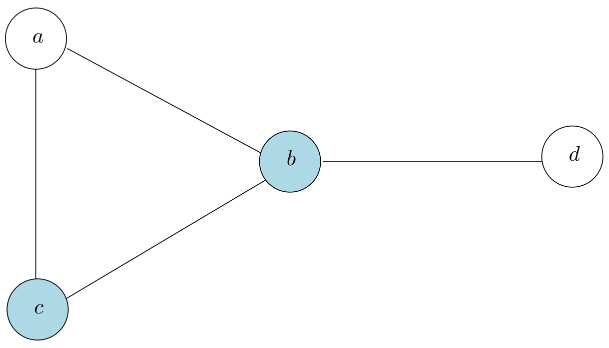


Figure 24: A Minimum Vertex Cover for the example graph with the shaded vertices in the vertex cover.

This tells us that the minimum vertex cover for this graph is of size 2 (see Figure 24 for a possible vertex cover $\{b, c\}$ of size 2 for this graph).

6.6 3-Vertex Coloring

We now consider the 3-vertex coloring problem (sometimes referred to as 3-colorability) where we wish to determine if we can color all the vertices of a graph using only three colors in such a way that no two adjacent vertices have the same color. Formally we define it below.

Input: A graph $G = (V, E)$
Question: Do there exist sets X_1, X_2, X_3 such that $X_1 \cup X_2 \cup X_3 = V(G)$, $X_i \cap X_j = \emptyset$ for $i \neq j$, and the two vertices of any edge do not belong both to X_i for any i?

This problem shall be referred to as 3-VERTEX COLORING. In order to formulate a way to solve this problem using branch decomposition and the b-parse we need to define what a solution set is and specify the values for the base case solutions. Second we need to define the join or composition operation given two input solution sets.

First, we observe that it can be shown that 3-VERTEX COLORING can be represented in monadic second order logic (see Courcelle [Cou08] and Christian [Chr02]). This means we know that a linear time algorithm exists for solving the 3-VERTEX COLORING problem using a branch decomposition.

6.6.1 The Solution Sets for 3-Vertex Coloring

A solution set in the case of 3-VERTEX COLORING is simply a set of valid colorings. A coloring in this case is a way of assigning one of the three possible colors to each vertex label in a set of labels. We represent a coloring as a tuple. In the language of Christian ([Chr02]) we use Fixed Partioning to represent the colorings because the number of color values for each coloring is always exactly the size of the relevant middleset. A solution set for 3-VERTEX COLORING is indexed by the members of the relevant middleset and consists of multiple possible colorings of the members of the middleset. So if V is a solution set then $V[(0, 1)] = \{(1, 2), (2, 1)\}$ means that the middleset for V is $\{0, 1\}$ and that coloring 0 color 1 and 1 color 2, or swapped, is a valid coloring of the subgraph induced below. This can be represented in table form as along the bottom row of Figure 25. The solution sets for the three base cases of an edge or leaf operator are defined as follows.

A Disconnected Edge or K_2: For an edge with an empty middleset—that is a K_2 or disconnected edge—we know we can always color this given 3 colors so we use the empty set to denote a valid coloring exists. That is we let $V[\emptyset] = \{\emptyset\}$, a set of a non-needed coloring. (Later on, if $V[\emptyset] = \emptyset$, then there is no valid coloring.)

A Pendant Edge: For an edge $\{a, b\}$ with just one active middleset member a, which is a pendant or P_a, we can color the one active vertex with the three different colors (see bottom of Figure 25).

A Fully Connected Edge: For an edge $\{a, b\}$ with both vertices in its middleset the solutions are just all the ways of assigning two colors out of three such that the two colors are not the same (see bottom of Figure 25)

6.6.2 Defining the Join Operation for 3-Vertex Coloring

Essentially to join two sets of colorings we need to compare each coloring from each set with the others and if they are compatible join or unite the two colorings. We thus apply the following algorithm where V is the output solution set and V' and V'' are the two input solution sets:

```
algorithm joinSolutions
    Input: solution sets $V'$, $V''$
begin
    for each coloring $s_j \in V'$
        for each coloring $s_k \in V''$
            if $s_j$ is compatible with $s_k$
            begin
                unite $s_j$ and $s_k$ and add a solution for $V$
            end
end
```

Colors $s_j \in V'$ and $s_k \in V''$ are compatible if $\forall c \in B' \cap B''$ $s_j(c) = s_k(c)$ where B' is the middleset for V', B'' is the middleset for V'' and B is the outer middleset for the join. We unite s_j and s_k by forming $s \in V$ as follows:

$$s = \bigcup_{c' \in (B' \cap B) - (B' \cap B'')} [s_j(c')] \quad \cup \quad \bigcup_{c \in B} [s_j(c) = s_k(c)] \quad \cup \quad \bigcup_{c'' \in (B'' \cap B) - (B' \cap B'')} [s_k(c'')]$$

$$\forall c \in B' \cap B'' \; \forall c' \in B' \; \forall c'' \in B'' \text{ s.t. } c', c'' \in (B' \cup B'') \cap B$$

Some special cases occur where the outer middleset is empty or one or more of the input solution middlesets is empty. In these cases we apply the logic:

```
algorithm processEmptySet
    Input: solution sets $V'$, $V''$, middlesets $B'$ and $B''$ and outer middleset $B$
begin
    if $B = \emptyset$ and $B' = B''$ then
    begin
        if $V' \cap V'' \neq \emptyset$ then $V(\emptyset) = 1$
        else leave $V$ empty
    end
    else if $B' \cap B''$ is empty then $V = V' \times V''$
end
```

We continue processing until there are no more operators. We return yes if the last operator value set is non-empty otherwise we return no.

Theorem 21 *Given there is a valid coloring for all the vertices of some subgraph H whenever there is a valid coloring for the members of the middleset for H then joining two compatible colorings will result in a valid coloring for the resulting joined subgraph.*

Proof: Consider a subgraph G with associated middleset B and two input solutions V' and V'' representing subgraphs G' and G'' with associated middlesets B' and B''. Then as we have seen V' and V'' represent sets of valid colorings of their subgraphs expressed solely in terms of colorings of the associated middlesets. As we have seen the leaf (base case) solution sets are defined to be valid colorings of their entire subgraphs (which are of course single edges).

Two colorings s_j and s_k are compatible if the color values for each coloring are the same for the middleset members that are in common. That is in the subgraph G formed from G' and G'' we ask is there a way of validly coloring the (middleset) vertices using s_j and s_k. The vertices that are in common will have to have the same color for this to hold. Given compatibility the new coloring is just the way of coloring the vertices only in B' plus the way of coloring the vertices only in B'' plus the color values for the vertices in common. This will be a valid coloring since s_j and s_k are valid and the vertices in common have a valid color since it is valid for both s_j and s_k. □

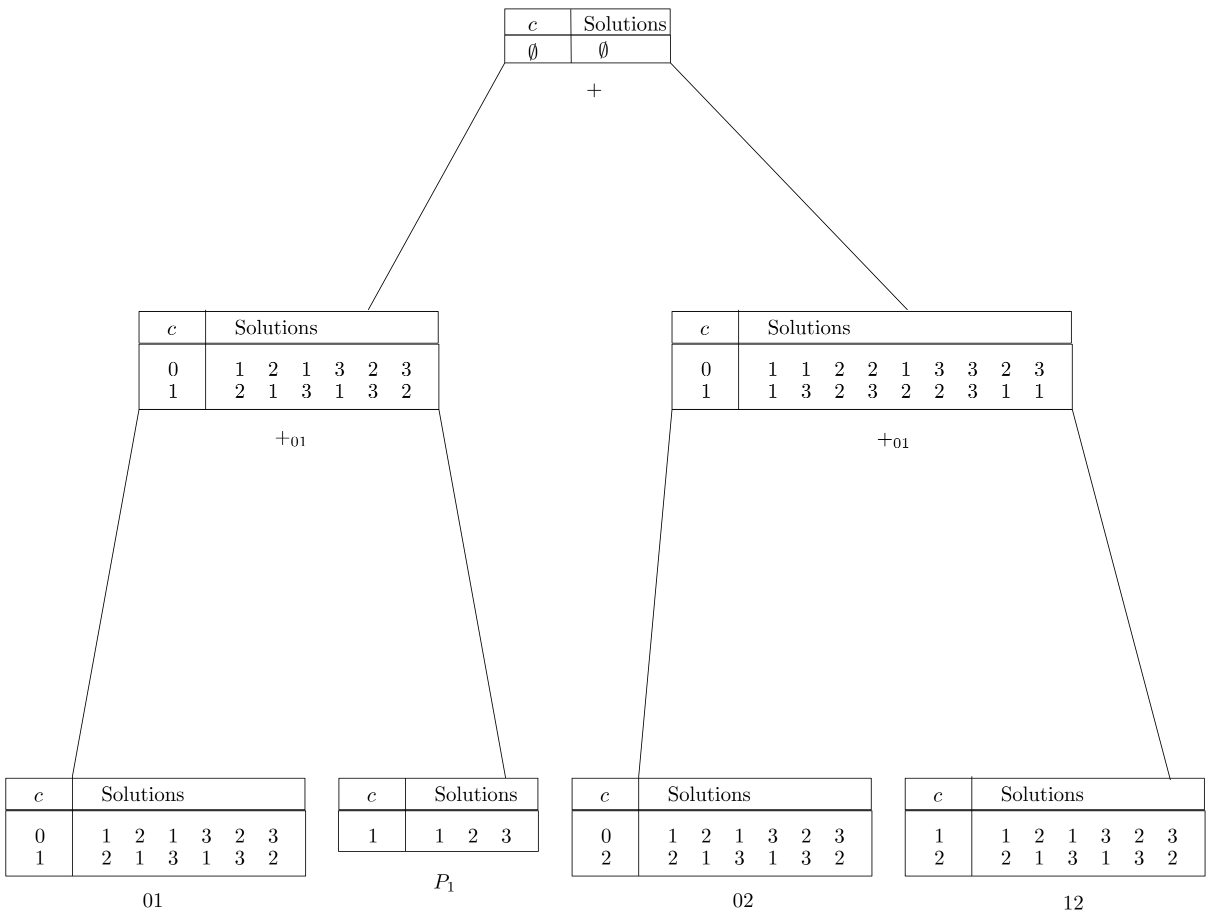
6.6.3 Time Complexity for 3-Vertex Coloring

The maximum size of any solution set for 3-VERTEX COLORING is simply the number of ways to assign 3 colors to b vertices when order matters and repetition is allowed and b is the maximum width for the given decomposition. This maximum size is 3^b. The maximum number of operations there are in forming the leaf solutions is then 3^b and the maximum number of operations in any join must be $3^b 3^b = 3^{2b}$ since with each join we compare every solution value of one solution set with every solution value of the other solution set.

In any b-parse there are $O(m)$ leaf operators by definition where $m = |E(G)|$ and the number of composition operators can not be more than the number of leaf operators. Given this we can see that the worst case running time for the above algorithm for solving the 3-VERTEX COLORING problem using a b-parse as input must be $O((3^b + 3^{2b})m)$. The maximum space used by any solution set in the above representation will be $b3^b$. At any one time $O(m)$ solution sets need to be maintained meaning that the worst-case space complexity for the above algorithm for 3-VERTEX COLORING is $O(b3^b m)$.

6.6.4 An Example for 3-Vertex Coloring

Consider again the graph in Figure 22. As we have seen this has the possible b-parse



$$2\ 01\ P_1\ +_{01}\ 02\ 12\ +_{01}\ +$$

Figure 25: A possible sequence of solution sets for this b-parse for 3-VERTEX COLORING arranged in a tree with each b-parse operator given below its matching solution set.

$2\ 01\ P_1\ +_{01}\ 02\ 12\ +_{01}\ +$

For a possible sequence of solution sets arranged in a tree mirroring the underlying decomposition tree consider the diagram in Figure 25. Each solution set is given with its matching b-parse operator below.

Consider the two edges ab and bd represented by the operators 01 and P_1 with middlesets $\{0, 1\}$ and $\{1\}$. Then the initial edge solutions V' and V'' will be as seen to the left along the bottom level of the tree given in the figure. In joining these solutions we observe that the intersection of $\{0, 1\}$ and $\{1\}$ is $\{1\}$ and the union is $\{0, 1\}$. The outer middleset for this join is $\{0, 1\}$. Then looping through each j and k for the solutions we observe that $V'[(0, 1)] = (1, 2)$ is compatible with $V''[(1)] = (1)$ but not with $V''[(1)] = (2)$ or $V''[(1)] = (3)$. We then form a solution for V by choosing 0 => 1 and 1 => 2 giving a new solution value of $V[(0, 1)] = (1, 2)$. We repeat this for solutions s_j and s_k for all j and k. In fact we see that after all the comparisons the solution set $V = V'$.

We repeat this for edges ac and bc with operators 02 and 12 with middlesets $\{0, 2\}$ and $\{1, 2\}$. Again let V' be the solution set for the left operator 02 and V'' be the solution

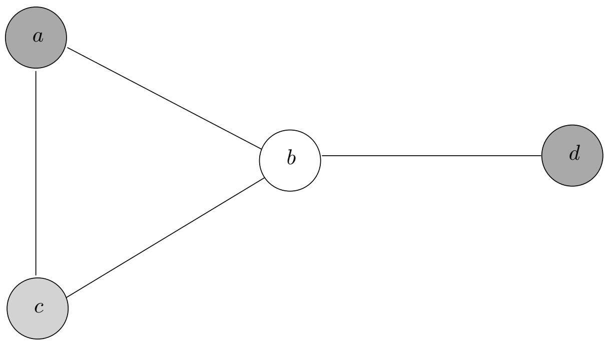


Figure 26: A 3-Vertex coloring for the example graph with colors represented by the three different shades.

set for the right operator 12 (see right side of the figure). In this case in joining these solution sets we observe that the intersection of $\{0, 2\}$ and $\{1, 2\}$ is $\{2\}$ and the union is $\{0, 1, 2\}$. The outer middleset for this join is $\{0, 1\}$. Then looping through each j and k for the solutions we observe that $V'[(0, 2)] = (1, 2)$ is compatible with $V''[(1, 2)] = (1, 2)$ while $V'[(0, 2)] = (1, 2)$ and $V''[(1, 2)] = (2, 1)$ are not. We then form a solution for V by choosing 0 => 1, 1 => 1 and 2 => 2 but we discard the value for 2 because 2 is not in the outer middleset giving a new solution value of $V[(0, 1)] = (1, 1)$. We repeat this for solutions s_j and s_k for all j and k. We obtain the solution set V as seen in the figure. This tells us that in the subgraph formed by the composing of the subgraphs associated with V' and V'' 0 and 1 can take any two color values out the three colors—clearly 0 and 1 are not adjacent in this subgraph.

We finally join the two subgraphs created above with associated solution sets V' and V'' in top of the figure. Because the outer middleset is empty and the two inner middlesets are the same being both $\{0, 1\}$ we apply the algorithm *processEmptySet* and simply take the intersection of V' and V''. This is clearly non-empty hence we set the resulting solution set V to the special value $V[\emptyset] = \{\emptyset\}$ (see top of the figure). Since we have reached the root of the tree and thus the final operator we return yes the underlying graph in this case is 3-vertex colorable (see Figure 26 for a possible coloring of this graph).

7 Algorithms for working with b-parses

In this section a detailed set of algorithms will be proposed for generating b-parses and processing b-parse input in order to help solve hard problems.

7.1 Generating b-parses

In order to generate a b-parse for a given graph we need first to find a branch decomposition of low width. Given this we construct the associated decomposition tree by rooting the branch decomposition then reading the operators of the tree in a post-order traversal producing an intermediary parse. We then relabel the parse in order to ensure that the labelling of the b-parse is correct, namely that the labels, starting at 0, restrictively increase from left-to-right and no label is more than $\left\lfloor \frac{3}{2}b \right\rfloor$.

As we saw above finding a branch decomposition of low width is $\mathcal{NP}$-hard and generally exponential in practice unless the graph is one of a few given types such as planar (see [HKK05, ST94]). Once we have a branch decomposition we can root it in linear time. We consider all the edges of the branch decomposition in turn until an optimal edge is encountered (where optimal is defined as below). Given this rooted tree one can read the tree in post-order traversal in linear time in terms of the number of nodes. Finally, one can relabel the parse in linear time in the number of operators by following a simple one-to-one remapping process (see below) or by using the procedure given in Lemma 17.

The algorithm for generating a b-parse used here does not find optimal branch decompositions. It follows the splitting procedure described by Hicks [HKK05] to build the tree. While splitting it greedily attempts to find a separation that is no worse than those so far encountered except for a few fixed situations such as when an internal node has two leaves and one internal neighbor. In this case a new internal node is attached to this existing node and the two leaves are moved to this new node. This helps ensure that the tree is better balanced.

Constructing a branch decomposition consists of two elements: building a tree and (while building) finding good separations.

7.1.1 Building the Tree

The process for building the branch decomposition is as follows: Given a graph G

1. Form a star T with as many ends as there are edges in G
2. While there are nodes in T whose degree is greater than 3
    (a) choose a node u with leaves whose degree is greater than 3
    (b) if u has more than 2 leaves
        i. find a greedy separation of the leaves (A, B)
        ii. create a new node v
        iii. attach v to u
        iv. detach the leaf set B from u and move them to v
    (c) else if u has 2 leaves
        i. create a new node v
        ii. attach v to u
        iii. move both leaves from u to v

7.1.2 Finding the Separations

Finding good separations is the difficult task. In the algorithm used in this paper this process is simplistic. Given a partial decomposition with tree T and node u with attached leaves C.

1. Calculate the entire set of leaves S including
    (a) leaves, C, directly connected to u
    (b) leaves connected to nodes connected to the subtrees connected to u

2. Find an optimal separation of S
   (a) let $A = C \cup (S - C)$ and $B = \emptyset$
   (b) while A has any leaves that are in C
      i. remove a leaf from A that is in C and move to B
      ii. calculate the width of the separation
      iii. if this separation is minimal record it
3. Return the minimal separation of S namely (A, B)

7.1.3 Rooting the Tree

Next we need to root the branch decomposition. Given the tree T for an unrooted branch decomposition with width b.

1. Find the **optimal** edge $\{u, v\}$ of T to split
2. Add a new node r and edges $\{u, r\}$ and $\{v, r\}$ and remove the edge $\{u, v\}$
3. Recursively traverse the tree adding information about the parent of each node where direction is downward from r

We define an **optimal** edge as an edge of width b that is near the "center" of the tree. Here, a **center edge** of the tree, when removed, divides the number of original leaves evenly into two parts within a given threshold.

7.1.4 Labelling the b-parse

We now give a simple greedy method (that is not optimal) to label the nodes, using a post-order traversal. Given a rooted branch decomposition with tree T and width b we do the following.

1. Recursively traverse T
2. Record each node as it is seen
   (a) if the node is a leaf, record the leaf middleset information
      i. if the middleset is empty, record the leaf as operator K_2
      ii. if the middleset has a single member a, record the leaf as operator P_a
      iii. if the middleset has membership $\{a, b\}$, record the leaf as operator ab
   (b) if the node is internal with middleset $\{a, b, \ldots, c\}$, record the node as operator $+_{ab \ldots c}$
   (c) if the node is the root, record the node as the operator $+$
3. Read the intermediary parse from left-to-right and map each vertex label to a number $0 \leq i < \left\lfloor \frac{3}{2} b \right\rfloor$.
4. As each intermediary parse vertex label is seen for the first time

(a) find the next free label i not currently mapped

(b) if such a label i exists, use this and add it to the map

(c) if no such i exists then increment the highest label i in the map and add the entry to the map

(d) abort if label $i = \left\lfloor \frac{3}{2}b \right\rfloor$ is reached–need to use a slightly more computational expensive labeling algorithm (see Lemma 17).

5. As each vertex label is not included in the next outer middleset remove its entry from the map

Thus a reading of a branch decomposition of width 2 might give an intermediary parse of

$$2\ ab\ bc\ +_{ac}\ ad\ cd\ +_{ac}\ +$$

which would become

$$2\ 01\ 12\ +_{02}\ 01\ 12\ +_{02}\ +$$

7.2 Processing a *b*-parse

Essentially we want algorithms that read the *b*-parse, from left to right, traversing the underlying decomposition tree and processing the operators as they are encountered. We shall examine both sequential and parallel processing algorithms.

7.2.1 A Sequential Algorithm

Initially, a sequential algorithm is required both for its own sake and to provide a control for time tests comparisons with a future parallelized algorithm. The algorithm chosen is based on the generic algorithm given in the previous section. It works by reading through the *b*-parse, pushing operators onto the stack then processing each leaf and composition operator as they are encountered off the stack. This algorithm can be described as follows:

```
algorithm processBParseSequentially
    Input: a valid b-parse, an intermediary stack
begin
    initialize stack
    while there are still more operators in the b-parse do
    begin
        if the operator is a leaf then
        begin
            solution set = generate_leaf_solution_set(operator)
            push solution set onto the stack
        end
        else {this is a composition operator}
        begin
```

```
                pop the last two solution sets (left, right) off the stack
                solution set = *join_solution_sets*(left, right)
                push solution set onto the stack
            end
        end
        pop the solution set off the stack
        display *generate_result*(solution set)
end
```

As we have seen this is often a finite-state algorithm with the operators, read left to right, in a single scan similar to the t-parse processing algorithm of Cattell and Dinneen [CD94] but using a stack as the storage mechanism rather than a table. The program will need to handle the special case when the final solution is the solution associated with K_2. This algorithm can be generalized for many different problem sets by substituting call-back handlers for *generate_leaf_solution_set*, *join_solution_sets* and *generate_result* and by providing a problem specific structure for each solution set. This algorithm is clearly linear in the number of operators which, in turn, a linear function of the number of edges.

7.2.2 A Parallel Algorithm

An alternate parallelized algorithm can also be used for many problems using bounded branchwidth graphs as input. The algorithm is based upon the parallelized t-parse processing algorithm given in [Pro11]. In brief the algorithm is designed for CREW PRAM. It takes as input a graph with a branch decomposition with bounded branchwidth in b-parse format. This algorithm varies from the sequential algorithm by first needing to pre-process the input to convert it into a tree which can be traversed bottom up in parallel.

Thus there are essentially two steps: first the input b-parse is converted into an explicit decomposition tree; second the tree is read from bottom-up with each token on the same level processed in parallel according to its type (that is leaf operator or composition operator). As each token in the tree is processed a solution set is updated for that branch of the tree. The actual processing of each operator is the same as for the sequential algorithm. In detail the parallel algorithm has the following steps:

1. Split the b-parse into its subtokens and reassemble as an explicit decomposition tree keeping track of all the leaves (these are where we will start processing in step 2).

2. Process the tokens from each leaf upwards in parallel. Each branch heading up from each leaf has an associated solution set.

   i If the token is a leaf operator process and update the solution and move up to the parent of the current token node.

   ii If the token is binary (composition) wait until we have reached this node along the other branch. If the node is (arbitrarily) from the left branch then perform the relevant composition operation (given the current middleset information) on the solution set and carry on to the parent. If the node is from the right branch enter a done status. The associated solution structure memory for this done node can be freed at this point.

3. Finish when the last operator has been processed and the root token has been reached.

Synchronization of the threads involved will depend upon the implementation chosen (see Section 7.2.5 below for an example). The end result will be a solution set associated with the root of the decomposition tree. This can be interpreted in exactly the same way as would be the final solution set during standard sequential stack processing of the *b*-parse.

For the first step a linear process will be used which navigates the operators of the *b*-parse input in reverse polish form (an alternate parallelized process is possible for constructing the decomposition tree, as explained below). The second step is inherently able to be parallelized given the decomposition tree. All leaf nodes and associated branches can be processed in parallel. Processors will however have to wait when they reach composition operators. An efficient use of processors would allow recycling of processors once the node branches they have been following finish. The simpler approach to be followed here does not optimize in this way.

Another required feature of this algorithm is that it be generic for a range of problem sets. As with the sequential algorithm, this can be implemented using call-back handlers and a problem defined solution set for each *b*-parse operation including the two operator calls (leaf initialization and composition) and the final interpretation of the result. Note that initialization of the solution set occurs during the leaf operations. Indeed, the same handlers and solution set structure can be used as for the sequential implementation.

As mentioned above there are at least two ways we can construct the decomposition tree.

7.2.3 Sequential Decomposition Tree Construction Process

We read the *b*-parse in the same manner as for the sequential parsing algorithm. As we encounter each operator we push it and associated information onto an operator stack. We then process this stack recursively constructing the parent node information and hence the tree itself. This is the method that will be used in the implementation and experiments included in [Pro13].

7.2.4 Alternate Parallelized Decomposition Tree Construction Process

Alternatively, we can build the decomposition tree from the input *b*-parse in parallel as follows:

1. Read the input into an array of “nodes”.
2. Determine the parent of each node in parallel by
   (a) If an element is a leaf, the parent is the next ‘+’ operator (either one element on or two elements on).
   (b) If the element is a ‘+’ :
      i. If the next element is a ‘+’ that is the parent.
      ii. If the next element is a leaf followed by a ‘+’ that ‘+’ operator is the parent.

iii. If the '+' is followed by two leaves then the parent is the next '+' which follows directly after another '+' operator (not including this current operator).

3. Gather the leaves in parallel.

7.2.5 A CUDA Implementation

CUDA (see `http://www.nvidia.com/cuda/`) is the GPGPU platform provided by Nvidia Corporation that enables software developers to access the low level instructions and memory of the Nvidia GPUs. With respect to the current architecture of GPUs, CUDA follows the Single Instruction with Multiple Threads approach to parallel processing. Note that while CUDA is restricted to the Nvidia GPUs (and currently the most popular), there is a similar OpenCL (see `http://www.khronos.org/opencl`) platform that provides a common interface for heterogeneous and parallel processing for both CPU and GPU based systems on different devices, such as AMD Radeon graphics cards.

A possible CUDA version of the parallelized algorithm is as follows. The host program first sequentially converts an input *b*-parse into its underlying decomposition tree. The host program then calls a kernel function to process the tree. The kernel function enters a loop processing each level of the tree starting from the leaves in parallel until the root level is reached. Note that this same algorithm could be used more generally by treating the kernel call as an ordinary function call.

Each thread available is assigned a branch. If the current branch operator is a leaf the thread processes this immediately and increments its pointer to point to the parent of this operator along the branch. If the current operator is a composition operator then the thread waits until both branches for the operator are ready. Then arbitrarily the thread managing the left side of the operation proceeds to handle the join. The left hand thread then increments its pointer to the next operator along the branch. The right hand thread is done.

This implementation can be described as follows (where *generate_leaf_solution_set*, *join_solution_sets* and *generate_result* are defined for each problem set):

```
algorithm processBParseInParallel
    Input: a valid b-parse
begin
    nodes, leaves = build_parse_tree(bparse)
    call kernel processTree(nodes, leaves, solutions)
    display generate_result(solutions)
end

algorithm processTree
    Input: decomposition tree nodes, leaves, solutions
begin
    set found_root = False
    while not found_root do
    begin
        for branches in parallel
        begin
```

```
            **if** current branch is a leaf **then**
                active_solutions[current branch] = *generate_leaf_solution_set*()
            **else if** this is a composition operator **then**
            **begin**
                **if** current branch is left side of current operation **then**
                    active_solutions[current branch] =
                        *join_solution_sets*(active_solutions[current branch],
                        active_solutions[right branch])
                **else**
                    this thread is done so **exit** thread
            **end**
            increment pointer along current branch
            **if** the pointer points to the root node **then**
                set found_root = True
        **end**
        **synchronize** threads before next while loop
    **end**
    **return**
**end**
```

This implementation is optimized to use a single block of threads. A more general implementation is possible where inter-block synchronization is enabled by having the host program call the kernel repeatedly once for each level of the tree (for the theory behind this see [DKP11]). Locking behaviors is not explicitly supported in CUDA though it is sometimes possible to achieve this effect indirectly [DKP11].

The thesis [Pro13] examined a practical implementation of algorithms both for generating and processing b-parses; implemented both for sequential processing and for parallelized processing on the OpenMP and CUDA platforms. For the most part the results were as expected at least for the sequential and CUDA programs. For low width graphs the performance of the CUDA program generally improves (if only slightly) relative to the sequential program as n increases and the width reduces.

7.2.6 Expected Time Complexity

It should be clear from the above that the sequential algorithm runs in linear time in terms of the number of input b-parse operators given there are n such operators.

As discussed with the parallelized algorithm, the building of the decomposition tree involves scanning the input operators from left to right and should be no worse than linear in terms of the number of those operators. The complexity of the second step for the parallelized algorithm is clearly a function of the height of the decomposition tree and so is a function of $\log n$ (where n is the order of the tree). This value of $\log n$ may however be high depending on how well-balanced the decomposition tree is.

This means that the complexity for the whole of this new algorithm can be no worse than linear. However bear in mind that the second stage is much the more process intensive so the overall timing should benefit from the parallelization of this stage given large enough n and given a large enough underlying graph G.

8 Conclusion and Future Work

We have seen that branch decomposition, which arose out of the graph minors project of Robertson and Seymour, can be a useful tool in solving certain types of graph based problem. However, use of branch decomposition raises other problems in turn. These problems include the question of how to find branch decompositions of low width, which is important since algorithms using branch decompositions are generally exponential in the width. They also include the problem of how best to represent the branch decomposition for data input and ease of processing.

There has been much research into the former question both for particular types of graph such as planar graphs and for graphs in general. There has also been work in the area of how to use branch decompositions to solve graph problems. One strategy involves the use of the concept of the decomposition tree to map to a branch decomposition and be used as input to a dynamic program. There has been less written on efficient ways to represent a branch decomposition, which we hoped to address here.

This paper has proposed the concept of the b-parse as a way of algebraically representing any branch decomposition. It builds upon the concept of the decomposition tree in the form proposed by Christian [Chr02] representing it in post-order form. The b-parse is inspired by the elegant t-parse representation for tree decompositions. Following this presentation of the b-parse, as a concept, we examined some examples of b-parses of low width and then entered a discussion on how to use the b-parse to solve various hard graph problems, such as minimum vertex cover and 3-coloring. We presented a generic framework for solving graph problems using the b-parse and touched upon the use of parallelism.

Areas of future work regarding the b-parse include

- More closely tying together the theory of the b-parse with that of the t-parse. For example, proving that one can easily represent a b-parse as a t-parse and vice versa.
- Developing an efficient b-parse generator for graphs of small branchwith. Also, consider when we can minimize the required labels (in most cases, we think $\lfloor \frac{3}{2}b \rfloor$ is too many for $b \geq 4$).
- Analyzing harder real-world graph problems using branch decomposition and the b-parse, such as the Travelling Salesman Problem.
- Developing a way to enumerate graphs of bounded branchwidth using canonic forms with the b-parse. This would also lead to developing a way to generate random graphs of bounded branchwidth.
- Implementing b-parse generation and processing for much larger graphs and testing parallelization against these.

References

[ALS91] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

[Bod87] Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. *Laboratory for Computer Science, Massachusetts Institute of Technology*, 1987.

[Bod93a] Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *ACM Symposium on the Theory of Computing*, volume 25, 1993.

[Bod93b] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2), 1993.

[BT99] Hans L. Bodlaender and Dimitrios M. Thilikos. Graphs with branchwidth at most three. *Journal of Algorithms*, 32:167–194, 1999.

[CD94] Kevin Cattell and Michael J. Dinneen. A characterization of graphs with vertex cover up to five. *LNCS*, 831:86–99, 1994.

[Chr02] William Christian. *Linear Time Algorithms for Graphs with Bounded Branchwidth.* PhD thesis, Rice University, 2002.

[Cou90] Bruno Courcelle. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Information and Computation*, 85:12–75, 1990.

[Cou08] Bruno Courcelle. A multivariate interlace polynomial and its computation for graphs of bounded clique-width. *Electronic Journal of Combinatorics*, 15(R69), 2008.

[CS03] William Cook and Paul Seymour. Tour merging via branch-decomposition. *INFORMS Journal on Computing*, 15(3):233–248, 2003.

[Den01] Zili Deng. *Exploiting Parse Trees for Graphs of Bounded Treewidth.* Masters, University of Auckland, 2001.

[DF99] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity.* Springer-Verlag, New York, 1999.

[DIM12] DIMACS. Dimacs implementation challenges, 2012. ftp://dimacs.rutgers.edu/pub/netflow/general-info/specs.tex.

[Din97] Michael J. Dinneen. Practical enumeration methods for graphs of bounded pathwidth and treewidth. *Centre for Discrete Mathematics and Theoretical Computer Science*, CDMTCS-055, 1997.

[DK10] Michael J. Dinneen and Masoud Khosravani. A linear time algorithm for the minimum spanning caterpillar problem for bounded treewidth graphs. *Structural Information and Communication Complexity*, pages 237–246, 2010.

[DKP11] Michael J. Dinneen, Masoud Khosravani, and Andrew Probert. Using OpenCL for implementing simple parallel graph algorithms. In Hamid R. Arabnia, editor, *Proceedings of the 17th annual conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11), part of WORLDCOMP'11.* CSREA Press, 2011.

[DT06] Frederic Dorn and Jan Arne Telle. Two birds with one stone: The best of branchwidth and treewidth with one algorithm. In José R. Correa, Alejandro Hevia, and Marcos A. Kiwi, editors, *LATIN-2006*, volume 3887 of *Lecture Notes in Computer Science*, pages 386–397. Springer, 2006.

[FG06] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory.* Springer-Verlag, New York, 2006.

[GT05] Qian-Ping Gu and Hisao Tamaki. Optimal branch-decomposition of planar graphs in $O(n^3)$ time. 2005.

[Hic00] Illya V. Hicks. *Branch Decompositions and their Applications.* PhD thesis, Rice University, 2000.

[Hic05] Illya V. Hicks. Graphs, Branchwidth, and Tangles! Oh My! *Networks*, pages 55–60, 2005.

[Hic06] Illya V. Hicks. Errantum: Petersen graph has branchwidth 4. Texas A&M University, May 2006.

[HKK05] Illya V. Hicks, Arie Koster, and Elif Kolotoglu. Branch and tree decomposition techniques for discrete optimization. *INFORMS*, New Orleans:1–33, 2005.

[Hli06] Petr Hlineny. Branch-width, parse trees, and monadic second-order logic for matroids. *Journal of Combinatorial Theory*, Series B 96:325–351, 2006.

[HO10] Illya V. Hicks and Sang-il Oum. Branch-width and tangles. *Wiley Encyclopedia of Operations Research and Management Science*, 2010.

[KT06] Jon Kleinberg and Eva Tardos. *Algorithm Design.* Addison-Wesley, 2006.

[Lau91] Clemens Lautemann. Tree automata, tree decomposition and hyperedge replacement. In *Graph Grammars and Their Application to Computer Science*, pages 520–537. Springer, 1991.

[MSW10] Suzanne J. Matthews, Seung-Jin Sul, and Tiffani L. Williams. A novel approach for compressing phylogenetic trees. In Mark Borodovsky, J. Peter Gogarten, Teresa M. Przytycka, and Sanguthevar Rajasekaren, editors, *Bioinformatics Research and Applications: 6th International Symposium, ISBRA 2010,* , pages 113–124. Springer-Verlag, 2010.

[OS07] Sang-il Oum and Paul D. Seymour. Testing branch-width. *Journal of Combinatorial Theory Series B*, 97(3):385–393, 2007.

[Pro11] Andrew Probert. A parallelized algorithm for processing graphs of bounded tree-width using *t*-parse representation. Honours dissertation, University of Auckland, Auckland New Zealand, 2011.

[Pro13] Andrew Probert. An algebraic representation of branch decomposition. Msc thesis, University of Auckland, Auckland, New Zealand, 2013.

[Ree92] Bruce Reed. Finding approximate separators and computing tree width quickly. In *24th Annual ACM STOC*, 1992.

[Ree97] Bruce Reed. Tree width and tangles: a new connectivity measure and some applications. *Survey in Combinatorics*, pages 87–162, 1997.

[Rig01] Kymberley Riggins. *On Characterizing Graphs with Branchwidth At Most Four.* Masters, Rice University, 2001.

[RS83] Neil Robertson and Paul D. Seymour. Graph Minors. I. Excluding a Forest. *Journal of Combinatorial Theory*, Series B 35:39–61, 1983.

[RS84] Neil Robertson and Paul D. Seymour. Graph Minors. III. Planar Tree-Width. *Journal of Combinatorial Theory*, Series B 36:49–64, 1984.

[RS86] Neil Robertson and Paul D. Seymour. Graph Minors. II. Algorithmic Aspects of Tree-Width. *Journal of Algorithms*, 7:309–322, 1986.

[RS90] Neil Robertson and Paul D. Seymour. Graph Minors. IV. Tree-Width and Well-Quasi-Ordering. *Journal of Combinatorial Theory*, Series B 48:227–254, 1990.

[RS91] Neil Robertson and Paul D. Seymour. Graph Minors. X. Obstructions to Tree-Decomposition. *Journal of Combinatorial Theory*, Series B 52:153–190, 1991.

[RS04] Neil Robertson and Paul D. Seymour. Graph minors XIII: The disjoint paths problem. *Journal of Combinatorial Theory*, Series B 92(2):325–357, 2004.

[ST94] Paul D. Seymour and Robin Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994.

[Sze08] Stefan Szeider. Monadic second order logic on graphs with local cardinality constraints. *Mathematical Foundations of Computer Science 2008*, pages 601–612, 2008.